UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament de Ciències de la Computació

Ph.D. in Computing

Structure discovery techniques for circuit design and process model visualization

Javier de San Pedro Martín

Advisor: Jordi Cortadella Fortuny

Barcelona, May 2017

Abstract

Graphs are one of the most used abstractions in many knowledge fields because of the easy and flexibility by which graphs can represent relationships between objects. The pervasiveness of graphs in many disciplines means that huge amounts of data are available in graph form, allowing many opportunities for the extraction of useful structure from these graphs in order to produce insight into the data.

In this thesis we introduce a series of techniques to resolve well-known challenges in the areas of digital circuit design and process mining. The underlying idea that ties all the approaches together is discovering structures in graphs. We show how many problems of practical importance in these areas can be solved utilizing both common and novel structure mining approaches.

In the area of digital circuit design, this thesis proposes automatically discovering frequent, repetitive structures in a circuit netlist in order to improve the quality of physical planning. These structures can be used during floorplanning to produce regular designs, which are known to be highly efficient and economical. At the same time, detecting these repeating structures can exponentially reduce the total design time.

The second focus of this thesis is in the area of the visualization of process models. Process mining is a recent area of research which centers on studying the behavior of real-life systems and their interactions with the environment. Complicated process models, however, hamper this goal. By discovering the important structures in these models, we propose a series of methods that can derive visualization-friendly process models with minimal loss in accuracy.

In addition, and combining the areas of circuit design and process mining, this thesis opens the area of specification mining in asynchronous circuits. Instead of the usual design flow, which involves synthesizing circuits from specifications, our proposal discovers specifications from implemented circuits. This area allows for many opportunities for verification and re-synthesis of asynchronous circuits.

The proposed methods have been tested using real-life benchmarks, and the quality of the results compared to the state-of-the-art.

Acknowledgments

This thesis would have been impossible without the expertise, guidance, and, sometimes, necessary insistence of my advisor, Prof. Jordi Cortadella. As all his students will attest, including myself, it is quite an opportunity to be able to work with him. I am thus extremely thankful for this opportunity, that has allowed me a firsthand view of both the academic and industrial worlds of Computer-aided design; as well as his ability to make sense even from the most terrible of my explanations.

I would also like to thank every person I have had the pleasure to work with. For the extremely useful guidance during the early days, Nikita Nikitin, Prof. Josep Carmona, and Prof. Jordi Petit. For their insights on Chip multiprocessors and Networks-on-Chip, Francesc Guim, Antoni Roca and Daniel Rivas. For his knowledge on the, for me, novel topic of Process Mining, Jorge Muñoz-Gama. For our discussions on asynchronous circuits and transition systems, Thomas Bourgeat. I hope that I have been as useful to you as you have been to me in, believe me, very stressful times.

And then, I would also like to thank all the people I have not been able to work with, but would have enjoyed to. From my lab colleagues working on similar topics, I would like to thank Alex Vidal, Alberto Moreno, and Lucas Machado. A special gratitude goes to Seppe vanden Broucke, Joos Buijs, and the CoSeLoG, ActiTraC, and 4TU.Datacentrum projects, for providing the benchmarks that have been used to evaluate many of the methods proposed in this thesis. Plus, big thanks to all the excellent people in UPC, including professors, researchers, and administrative staff.

I obviously will not forget about all the support provided by my family, nor from all the friends made during this experience – Daniel Alonso, Sergi Oliva, Adrià Gascón, Ramón Xuriguera, Josep Lluís Berral, Alberto Gutiérrez, Evelia Lizárraga, Eva Martinez, Carles Creus, Albert Vilamala, Alessandra Tosi, and M. Àngels Cerveró. Good luck you all in your future endeavors!

This work has been supported by a scolarship from the Catalan Government (FI-DGR 2013), by the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R), and by a gift from the Intel corporation.

Contents

Chapter 1

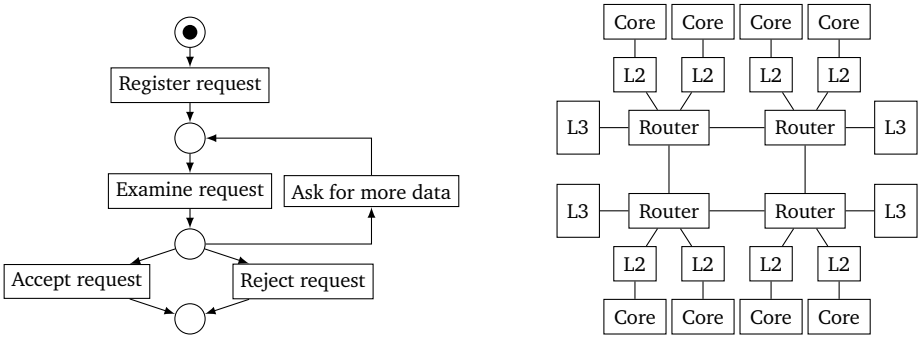
Introduction

The ever increasing amount of information generated and captured during day-to-day activities have firmly entrenched *data mining* as an essential part of almost every sector in the global economy. The goal of data mining is to allow the extraction of useful knowledge from large amounts of data. This data may be generated either as a by-product of other activities, e.g. trails of consumer interactions with both physical and online services, or data expressly captured by all types of sensors.

Data of interest may be represented in various forms. *Unstructured* data is perhaps the most simple, and includes free-form text, untagged audio or video, or any other data that does not reside in fixed fields [105]. On the other hand, *structured* data is organized according to some model. Fixed fields, tags and other types of markers are used to separate the individual data elements.

With the rise of the Internet, relational databases, semantic tagging and other advances, the amount of usable structured data has increased significantly. Furthermore, in many knowledge domains data naturally lends itself to a certain structure. It is common for trails of consumer interactions to be comprised of well-delimited events tagged with timestamps, even if the event information itself is unstructured. It is also common to find data described in terms of relations between objects, such as in relational databases, or any dataset describing a physical structure or a network. *Structure mining*, a subfield of data mining, centers on extracting information from all types of structured and semi-structured data.

Graphs provide an ideal abstraction model for structured data. At its core, a graph represents the relationships between a set entities in a specific knowledge domain. Because of the ease and flexibility by which graphs can represent structured data, they are abundant in many topics of interest. Thus, there has been a growing interest in the structured mining of graph data, an area referred as *graph mining* [81].



(a) A graph representing a process (*Petri net* [112]).

(b) A graph representing a Chip multiprocessor (*block-level netlist*).

Figure 1.1: Examples of graph-based data structures.

The generality of graphs allows graph mining techniques to be used in a variety of different knowledge fields. Figure 1.1 contains two examples of graphs, modeling data sets of the two different domains in which this thesis will primarily operate. However, graph mining is applicable to a myriad of other real-world topics.

Figure 1.1a describes the *control flow* of a business process using a modeling abstraction known as a *Petri net* [112]. Petri nets provide a formalism to describe concurrent systems, effectively describing all possible executions of a system. In the figure, a snippet of the behavior of an online request system is detailed. While the vertices represent the different tasks performed by the system, the edges specify the dependencies of the different tasks, e.g. it is necessary to register the request before examining it.

The area of *Process mining* deals with the discovery of process models, such as the one in Fig. 1.1a, from event logs. Process mining also deals with the analysis and extension of these process models. Process models, Petri nets, and the associated area of Process mining are documented with more detail in Section 2.2.

From a different knowledge field, Fig. 1.1a shows a *netlist*, which describes the physical connectivity of an electronic circuit. In this case, the vertices represent the different high-level functional blocks of a Chip multiprocessor: cores, I/O routers, and L2 and L3 cache memory modules. An edge indicates a physical bus, perhaps comprising thousands of individual on-chip wires, between a pair of components. Connectivity graphs like these are used, for example, as inputs during the design stage of chip multiprocessors, to ensure that highly connected components are kept physically close and thus minimize total wire length. Netlists and the general design process of integrated circuits will be further introduced in Section 2.3.

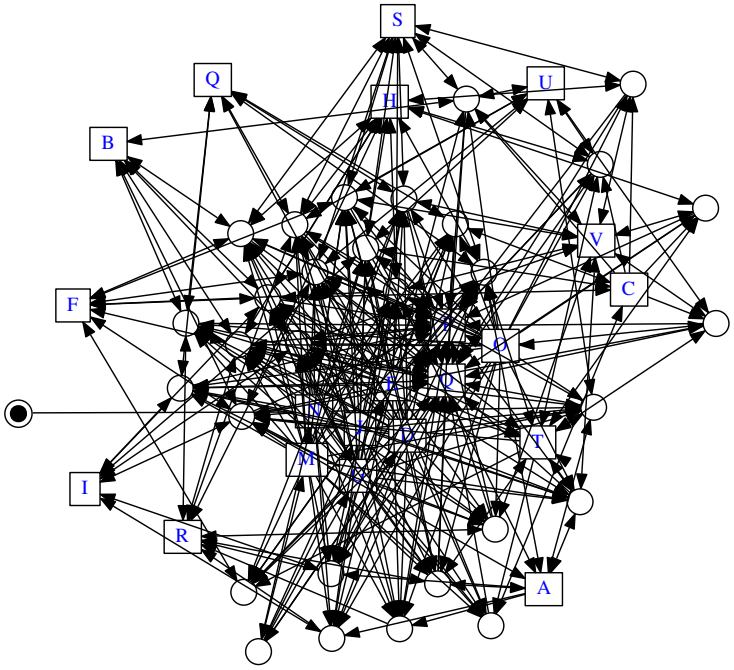


Figure 1.2: Process model extracted from consumer interactions with the helpdesk of a telecommunications company.

As in other related fields of data mining, one of the significant challenges in graph and structure mining is the ever growing amount of available data [105]. More information is being produced and captured than ever, and the trend for the next decade is towards exponential growth. Such growth can provide new opportunities to gather additional insights from the data. Due to the increased complexity of the data, however, more advanced mining tools are required before these benefits can be reaped. We center on two requirements, that we describe with more detail below.

First, the danger of overwhelming data, i.e. too much 'noise' and not enough 'signal' so as to provide useful insight into the data. In Figure 1.2, we show a process model constructed automatically from the event logs of real-life helpdesk consumer interactions of a large telecommunications company [56]. The graph shown is too complex to provide any insight into the underlying process, and thus fails at one of the primary objectives of data mining: providing useful information. Simplifying or entirely avoiding these types of models is one of the desirable objectives.

Second, larger data sets demand more efficient algorithms to process the available data under reasonable time constraints. Algorithms for graph mining should always consider efficiency in terms of the size of the input dataset.

Circuit netlists, for example, typically contain millions of individual vertices. Algorithms should be scalable in the number of vertices and edges, whenever possible. Alternatively, problems should be partitioned into smaller instances to avoid excessive runtimes.

The ability to apply graph mining strategies to a wide range of problems in the areas of circuit design and process mining, and the challenges and opportunities provided by the ever-increasing available data in graph form motivate this thesis. The rest of this chapter will describe the goals of this thesis with more detail.

1.1 Contributions of this thesis

In this thesis, we propose using graph and structure mining methods to solve various real-life problems in the areas of process mining and circuit design. The underlying idea behind all of the proposed methods is structure discovery, i.e., automatically extracting new structures from structured data, usually in the form of a graph. Specifically, the challenges addressed in this thesis are:

1. Finding repeating structures in netlists to optimize regularity in chip floorplans (Chapters 3 and 4).

2. Simplifying process models by discovering visually-friendly structures (Chapters 5, 6 and 7).

3. Reverse engineering asynchronous circuit specifications from implemented circuits (Chapter 8).

The following subsections provide an high-level summary of each one of these challenges.

1.1.1 Physical planning for regular layouts

In circuit design, the floorplanning problem comprises allocating the physical space required to functional blocks in a chip. Structures that are highly connected should be placed close together, so as to minimize the total required length of the wires. The computational complexity of the floorplanning problem highly depends on the number of components of the design.

Regular floorplans, which contain runs of repeating subpatterns, each of them with the same subfloorplan, are well-known to provide multiple benefits. Each of the repeating identical parts needs to be designed only once, thereby exponentially reducing the design cost and search space of the problem.

In Chip multiprocessors (CMPs), a standard industry practice to exploit regularity is via the use of tiles. A tiled CMP is usually entirely comprised of a single tile design that is replicated tens or hundreds of times, usually in a grid-like fashion. Chapter 3 presents a method to perform efficient floorplanning in presence of these tiles, while still guaranteeing the global physical constraints for manufacturing such as routability or abutability. The goal of the proposed method is to be used during the early exploration of CMP architectures, as an efficient estimator of the physical viability of candidate designs.

In Chapter 4, on the other hand, we present an approach that is more generally oriented towards all types of circuits. For this, we propose automatically finding repeating subgraphs in the design netlist using frequent subgraph mining techniques. We introduce a floorplanner, *HiReg*, that can use these frequent subgraphs to accelerate floorplanning runtime and automatically produce regular floorplans, with configurable focus on area or wire length minimization. The problem of frequent subgraph mining will be defined in Section 2.1.2, while the algorithm used will be detailed in Chapter 4.

These chapters are based on the following publications:

- J. de San Pedro, N. Nikitin, J. Cortadella, and J. Petit, *Physical Planning for the Architectural Exploration of Large-Scale Chip Multiprocessors*, in Proceedings of the 2013 IEEE/ACM Seventh International Symposium on Networks-on-Chip, Tempe, Arizona, USA, 2013.

- J. Cortadella, J. de San Pedro, N. Nikitin, and J. Petit, *Physical-aware system-level design for tiled hierarchical chip multiprocessors*, in Proceedings of the 2013 ACM International Symposium on Physical Design, New York, NY, USA, 2013, pp. 3–10.

- J. de San Pedro, J. Cortadella, and A. Roca, *A hierarchical approach for generating regular floorplans*, in Proceedings of the 33th IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, USA, 2014.

1.1.2 Visualization of process models

As with any other discipline in data mining, a significant challenge in process mining is presenting the results in a way such that new insight may actually be gained from the data. While ever increasing computing power combined with huge data sets provide new opportunities, in practice, there is a big gap between what computers can store and what humans can interpret and use. For crucial areas like health care, extracting value from the data is a challenge [7].

The visualization of process models in a understandable way for a human is a crucial step towards this end.

In this thesis we propose a series of methods with the goal of simplifying existing process models as well as process discovery techniques that allow the direct generation of visually-friendly models. The proposed methods involve applying graph mining techniques on top of either the process models themselves, or derived transition systems. While the methods are primarily oriented towards Petri nets, many of the methods can be extended to other Process model formalisms.

The first proposed approach, described in Chapter 5, introduces a series of methods to simplify existing process models. We also propose a metric that is able to rank the importance of the different control flow structures when reproducing the behavior of the original process. This way, parts of a model that have high visual complexity but only specify infrequent behavior can be identified and removed. The understandability of the model can be enhanced with a minimal impact on its fitness and precision.

Chapter 6 proposes an alternate approach in order to allow the simplification of process models without incurring any cost in precision. Representing all the behavior of a process in a single process model may not be possible without sacrificing simplicity, fitness or precision. This is because real-life processes are usually highly unstructured. The proposed approach automatically discovers multiple process models, each of them satisfying certain structural properties while centering on a specific aspect of the behavior of the process. Therefore, the complexity of the obtained models can be kept under check even for complex event logs.

Duplicate tasks [7] are an extension available in many process model formalisms, which allows two or more vertices in a single model to refer to the same event. Duplicate tasks can be used to simplify process models with minimal impact on their accuracy metrics. However, the automatic discovery of duplicate tasks is an open challenge in Process mining. In Chapter 7, we contribute a method to automatically discover duplicate tasks that is compatible with most process mining discovery algorithms. The proposed method utilizes a graph clustering strategy that is resilient to commonly used control flow structures, such as loops, choice or concurrency. Additional extensions to the formalisms of process models are also discussed.

These chapters are based on the following publications:

- J. de San Pedro, J. Carmona, and J. Cortadella, *Log-Based Simplification of Process Models*, in Business Process Management (BPM), Innsbruck, Austria, September 2015.

- J. de San Pedro and J. Cortadella, *Mining Structured Petri Nets for the Visualization of Process Behavior*, in Proceedings of the 2016 ACM Symposium on Applied Computing (SAC), Pisa, Italy, April 2016.

- J. de San Pedro and J. Cortadella, *Discovering duplicate tasks in transition systems for the simplification of process models*, in Business Process Management (BPM), Rio de Janeiro, Brazil, September 2016.

1.1.3 Specification mining for asynchronous circuits

Asynchronous circuits are not driven by a global clock signal, and offer many potential benefits in terms of lower power consumption and higher performing functional units, specially in light of the modern manufacturing processes and the involved challenges. However, they have since long claimed to be more difficult to design. In the traditional design flow of asynchronous control circuits, the desired behavior of the circuit is formally specified in the form of a *Signal Transition Graph*, from which an implementation can be automatically synthesized [46].

In Chapter 8, we introduce the concept of *specification mining*, in which a formal specification is obtained from the implementation of a circuit. We propose a method to perform specification mining valid for many types of asynchronous controllers and considering several delay models. The proposed method combines both graph mining and process mining techniques, such as the discovery algorithm proposed in Chapter 6.

This chapter is based on the following conference article:

- Javier de San Pedro, Thomas Bourgeat and Jordi Cortadella, *Specification mining for asynchronous controllers*, in Proceedings of the 2016 IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Porto Alegre, Brazil, May 2016.

1.2 Structure of this document

This thesis is structured in 9 chapters. This chapter constitutes the introduction to the thesis. Chapter 2 introduces the necessary basic concepts in graphs, circuit design, and process mining required for understanding this thesis.

Chapters 3–4 deal with the explotation of regularity during floorplanning. In Chapter 3, we focus on Chip multiprocessors. Chapter 4 extends the method to all types of circuits by using frequent subgraph mining strategies and providing a more general version of the constraints defined in the previous chapter.

Chapters 5–7 center on the visualization of process models. Chapter 5 introduces a method to simplify process models for visual consumption by directly removing control flow structures that are least important to reproduce the most frequent behavior in the log. In Chapter 6, we propose an alternative method that allows simplification by generating a sequence of visually-friendly process models, each of them centering on specific aspects of the process, rather than constructing a unique model that may be difficult to simplify. Chapter 7 proposes a method to exploit the concept of duplicate tasks for further simplification of process models.

Chapter 8 opens the topic of specification mining for asynchronous circuits, and shows a method to obtain the specifications of asynchronous controllers with specific constraints for various delay models.

Chapter 2

Preliminaries

This chapter introduces the necessary background of concepts, algorithms and knowledge areas that will be used in the rest of this document.

Section 2.1 introduces the area of graph mining, which is behind most of the algorithms and methods presented in this thesis. Section 2.2 gives an overview of the area of process mining, that will be the basis of Chapters 5, 6, 7. Section 2.3 provides a tour of the basic design flow of VLSI circuits to help understand the context of Chapters 3, 4 and 8. Section 2.4 reviews the area of asynchronous circuit design used in Chapter 8. Finally, Section 2.5 summarizes the area of mathematical optimization, including satisfiability and linear programming.

2.1 Graph mining

The domain of *data mining* concerns itself with the extraction of patterns and knowledge from data to facilitate understanding and further use of the data. *Structure mining* is a proper subset of data mining that centers on structured datasets. In particular, *graph mining* centers on providing efficient algorithms to mine structures embedded in graphs [136]. Graphs, being one of the most generic types of structure, are naturally suited to represent most types of structured datasets.

Some examples of popular research areas in graph mining are [81]: frequent subgraph mining; graph classification, clustering, and search; work-flow mining. This section will provide an introduction to two of the most common subareas of graph mining, frequent subgraph mining and graph clustering.

2.1.1 Graph basics

An *undirected graph* $G = \langle V, E \rangle$ is a two-tuple comprising a set V of vertices and a set E of edges. An edge $e = \{v_1, v_2\} \in E$ is an unordered pair of vertices $v_1, v_2 \in V$. We say that any two vertices $v_1, v_2 \in V$ are *adjacent* if there is an edge $e = \{v_1, v_2\} \in E$ connecting them. In this case, we also say that v_1 and v_2 are *incident* to e. The *degree* of a vertex $v \in V$ is the number of edges incident to v. Note that there can be only one edge between any pair of vertices $v_1, v_2 \in V$.

A path in G is a finite sequence of different vertices $v_1, \ldots, v_n \in V$ so that $e_1 = \{v_1, v_2\}, \ldots, e_n = \{v_{n-1}, v_n\} \in E$. A path $v_1, \ldots, v_n \in V$ where $v_1 = v_n$ is also called a *cycle*. A graph is *connected* if there exists a path between each pair of vertices. It is *cyclic* if contains any cycle.

In a *directed* graph $G = \langle V, E \rangle$, V is the set of vertices, while $E = V^2$ is a set of ordered pairs of vertices. Every $e = (v_1, v_2) \in E$ is a *directed edge*. Unlike undirected graphs, every edge $e = (v_1, v_2)$ has a direction, with v_1, v_2 being the *head* and the *tail* of e respectively. In addition, v_2 is *direct successor* of v_1, while v_1 is a *direct predecessor* of v_2.

A path in a directed graph respects the direction of the edges: $v_1, \ldots, v_n \in V$ is a path only if $e_1 = (v_1, v_2), \ldots, e_n = (v_{n-1}, v_n), \in E$. Given a directed graph G and two vertices $v_1, v_2 \in V$, if there exists a path between v_1, v_2, we say that v_2 is a *successor* of v_1, while v_1 is a *predecessor* of v_2. Similarly to undirected graphs, a *directed cycle* is a path $v_1, \ldots, v_n \in V$ in which $v_1 = v_n$.

A directed graph is *strongly connected* if there is a path between every ordered pair of vertices. Otherwise, a graph is *weakly connected* if there is a path between each unordered pair of vertices. A *directed acyclic graph* (DAG) is directed graph in which there are no cycles.

A *subgraph* G' of graph $G = \langle V, E \rangle$ is another graph $G' = \langle V', E' \rangle$ in which $V' \subseteq V$ and $E' \subseteq E$. We say G' is an *induced subgraph* if for every $v \in V'$, all of the incident edges of v in G are in E'.

A graph *vertex labeling* is a function $\mathscr{L} : V \to \Sigma$ that assigns a label $l \in \Sigma$ for each vertex, where Σ is the alphabet of labels. Conversely, an *edge labeling* $\mathscr{L} : E \to \Sigma$ maps a label to every edge. Often the combination of a graph and labeling is referred as *labeled graph*.

Planar graphs

An *embedding* or drawing of a graph $G = \langle V, E \rangle$ on a surface S is an assignment of a unique geometric position to each vertex $v \in V$ and of a curve to every edge $e = \{v_1, v_2\} \in E$ so that the starting and ending points of the curve correspond to the positions of v_1 and v_2. Unless specified, we will always assume S to be the two-dimensional plane, $\mathbb{R}^2$. An embedding is *planar* if no two edges

intersect except possibly at the endpoints. A graph is planar if it has no planar embedding in $\mathbb{R}^2$.

The *crossing number* of an embedding is the number of all intersections of the curves representing the edges (excluding the common endpoints). For a graph G, its *crossing number* is the minimal crossing number from all of its possible embeddings in $\mathbb{R}^2$. Thus, a graph is planar iff its crossing number is 0.

The crossing number of a graph has often been used as a more accurate measure of its complexity than its number of vertices or average degree. For example, the crossing number of a netlist has been used to provide bounds on the area and wire length required for routing a design [102].

Computing the crossing number of an arbitrary graph is a well-known NP-complete problem [68]. Despite that, there are several methods to estimate it. In this thesis we will use a technique derived from the *mincross* procedure as used in the commonly used graph drawing program *dot* [66] from the *Graphviz* suite [67]. As it is only an estimation, *mincross* may sometimes overestimate the number of crossings in large, dense graphs. These pathological cases, most often, are already highly complex even if the crossing number is overestimated.

Graph isomorphism

Two graphs $G_1 = \langle V_1, E_1 \rangle, G_2 = \langle V_2, E_2 \rangle$ are *isomorphic* if there exists a bijective function $f : V_1 \to V_2$ so that if any two vertices $v_1, v_2 \in V_1$ are adjacent in G_1 iif $f(v_1), f(v_2)$ are adjacent in G_2. This forms an equivalence relation on graphs.

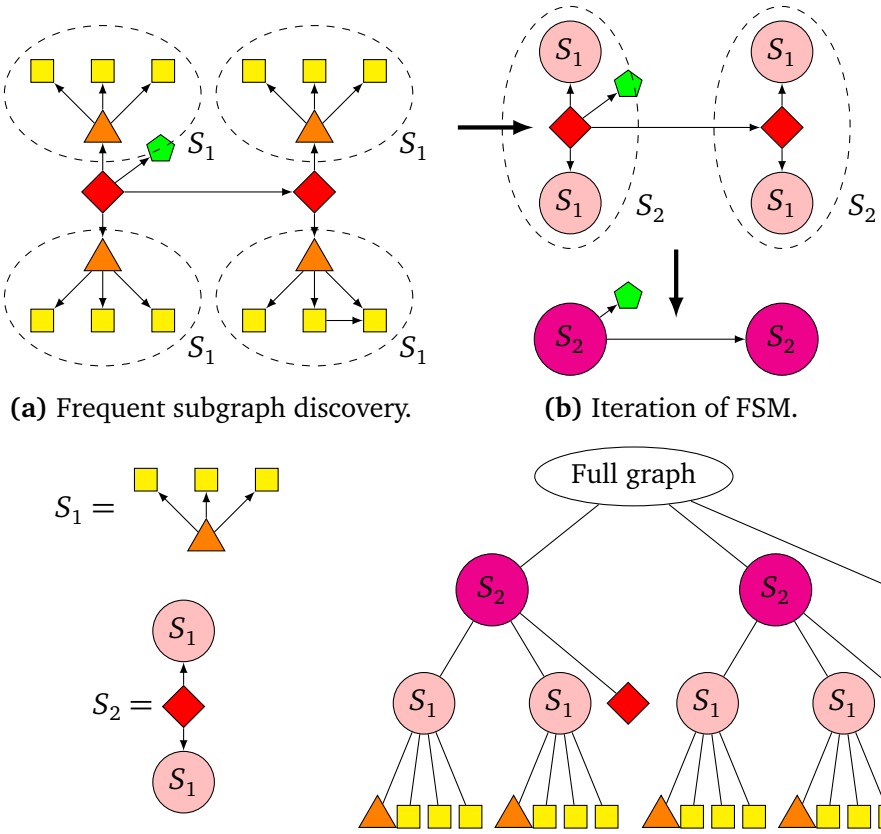
Two graphs G_1, G_2 labeled respectively with $\mathscr{L}_1$ and $\mathscr{L}_2$ are usually only considered isomorphic if the bijection f preserves the labeling of the vertices. That is, if $\forall v \in V_1, \mathscr{L}_1(v) = \mathscr{L}_2(f(v))$.

The complexity of computing whether two general graphs are isomorphic is currently an open question. There are polynomial-time algorithms for specific types of graphs, such as for planar graphs. However, this not the case for all types of graphs, even if in practice it can often be solved efficiently [44].

Given two graphs G_1, G_2, the *subgraph isomorphism* problem is defined as finding G_1', a subgraph of G_1 isomorphic to G_2. Unlike graph isomorphism, the problem of subgraph isomorphism is well-known to be NP-complete [44].

2.1.2 Frequent subgraph mining

Frequent subgraph mining (FSM) [81, 90] is one of the most important areas of graph mining. The objective of FSM is to extract all the subgraphs in a given dataset (a labeled graph, or a set of labeled graphs) which satisfy certain constraints. The most common goal is to find frequently recurring patterns, i.e. runs of isomorphic or almost isomorphic subgraphs with high occurrence



(a) Frequent subgraph discovery. **(b)** Iteration of FSM.

(c) Discovered subgraphs. **(d)** Discovered hierarchal structure.

Figure 2.1: Frequent subgraph and structure discovery.

counts. However, additional constraints may be used. For example, finding subgraphs that satisfy specific structural properties. Some of the most important applications of FSM have been seen in the domains of chemistry, biology, and web mining [81].

An example of the objectives of FSM can be seen in Fig. 2.1a [90]. We will refer to the graph shown in this figure as G. Different labels for every vertex in G are represented by different visual shapes in the figure. In G, subgraph S_1 has been identified as the most frequent, since there are 4 *instances* of S_1 in G. Any other subgraph of G is either less frequent than S_1, or has fewer vertices.

Note how the south-west instance of S_1 in Fig. 2.1a contains an additional edge that does not exist in other instances of S_1. Most FSM algorithms allow for *inexact* isomorphism, in which two subgraphs are not required to be entirely isomorphic to be counted as two instances of the same pattern. Instead, an approximate measure of similarity is used, depending on the nature of the underlying problem.

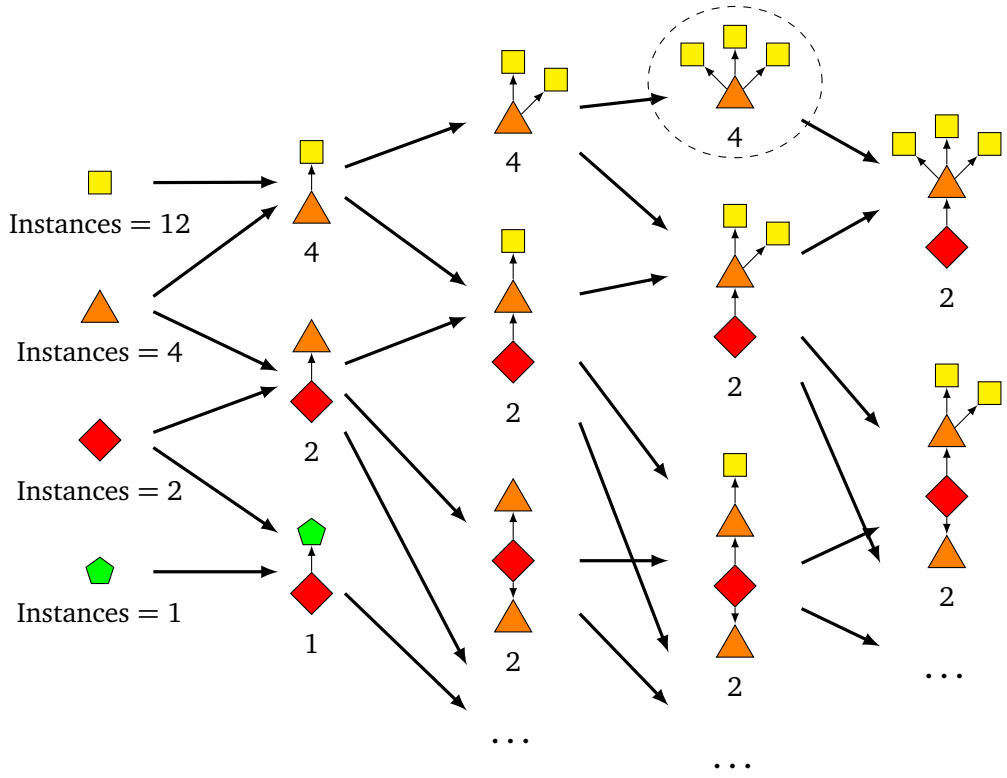


Figure 2.2: Partial search tree of frequent subgraph mining.

General-purpose FSM is a high-complexity problem because of its dependence on subgraph isomorphism, an NP-complete problem. The basic idea behind most FSM algorithms is based on two alternating steps: candidate generation and filtering. During candidate generation, each of the candidate subgraphs from the previous iteration is *grown* by adding new vertices and edges. The new, enlarged subgraphs form the set of candidates for the next step. This next step, filtering, *counts* the number of instances of each candidate subgraph: any candidate that is not frequent enough or violates any other constraint is purged. The process iterates until there are no further candidates.

Figure 2.2 illustrates this by showing a potential search tree of FSM in the graph from Fig. 2.1a (G). The first column corresponds to the first subgraph candidate set, composed of all possible 1-vertex subgraphs of G. The number of instances of each subgraph is also shown. On each next iteration (successive columns), every candidate is extended by adding exactly one vertex. Each candidate has multiple extensions, depending on the number of possible vertex labels. Note how, given a candidate, the number of instances of each successor is

Purpose	Input graph	Desirable subgraphs
Optimizing regularity in chip floorplans (Chapter 4)	Netlist	Maximally frequent subgraphs.
Visually simplify process models (Chapter 5)	Process model (e.g. Petri net)	Subgraph of a specific size that maximizes an objective function.
Mine structurally-simple process models (Chapter 6) Mine specifications from asynchronous circuits (Chapter 8)	Labeled transition system	Subgraphs that satisfy structural properties.

Table 2.1: Summary of graph mining variations proposed in this thesis.

always less or equal that of the candidate. The largest, most frequent candidate found in the search tree is indicated by a dashed line.

Different FSM algorithms are distinguished [81] by the candidate generation method (e.g. whether to enlarge a single vertex at at time or by combining multiple graphs), search strategy (e.g. BFS, DFS, ...), and candidate evaluation (metrics used for selecting the best graph, filtering, etc.). Adapting the algorithm to the nature of the specific application domain may allow significant reductions in the search space.

Subgraph mining is a core concept behind many of the approaches proposed in this thesis. In Chapter 4, frequent subgraph mining is used to discover repetitive patterns in netlists in order to increase the regularity of floorplans. Chapter 5 shows how to simplify existing process models by extracting a single subgraph that maximizes the quality of the model while keeping the complexity under check. In both Chapter 6 and Chapter 8, graph mining is performed on top of a labeled transition system An overview of these variations is described in Table 2.1.

2.1.3 Structure discovery

One of the practical applications of FSM is *structure discovery*, sometimes also called *hierarchical clustering*. The goal of structure discovery is to enhance the interpretation of data in graph form by producing a hierarchical description of the structural regularities in the data [43]. While FSM can be used to discover repeated structures in the graph, structure discovery can be used to organize these repeated structures into a hierarchical description of the data, allowing an higher level of abstraction.

A simple yet common method to perform structure discovery is by performing multiple passes of FSM. This approach is described in Fig. 2.1. Once a frequent subgraph S_1 is discovered in Fig. 2.1a, each of its instances is replaced by a new vertex that acts as a placeholder to the discovered subgraph instance, as seen in Fig. 2.1b. This is commonly referred to as *compressing* the graph, since it reduces the size of the graph.

The better a particular set of frequent subgraphs describe a graph, the more the graph will be compressed by replacing the instances of each subgraph with a placeholder vertex. Repeated iterations will discover additional subgraphs, including hierarchical ones, containing previously compressed subgraphs. This is exemplified by subgraph S_2 in Fig. 2.1b, which comprises two instances of S_1. Whenever a newly-discovered subgraph is defined in terms of existing identified subgraphs, these form a hierarchy. For example, the hierarchy tree formed by S_1 and S_2 is shown in Fig. 2.1d. This structure describes the graph in a much more compact way, and also provides an abstracted view of the regular patterns in the graph.

Structure discovery has been applied to areas such as data compression [117] or knowledge conceptual clustering [84]. The discovered hierarchies can provide varying levels of interpretation, with increased or decreased detail depending on the goals of the data analysis. As in FSM, inexact compression is often used, even allowing for potentially overlapping subgraphs.

In this thesis, structure discovery is the basis of the method described in Chapter 4 to generate regular floorplans. Existing structure in the input netlist is automatically discovered and used to enhance the quality of floorplans.

2.1.4 Graph clustering

Despite the name, *graph clustering* is the problem of trying to find sets of "related" vertices in a graph [124]. It should not be confused with the clustering of graphs themselves.

Clustering in general is one of the main areas of research in data mining. Unfortunately, no single definition of a cluster is universally accepted. Formally, the vertices assigned to a particular cluster should be similar and/or connected in some predefined sense. In some applications, it is desirable for clusters of vertices to be *connected*: the number of edges that remain within a cluster should be high, while there should be few edges that cross cluster boundaries. In this scenario, good clusters usually form dense subgraphs.

Alternatively, it might be desirable for clusters to be composed of *similar* vertices. The higher the similarity index, the more likely two vertices are clustered together. Computing similarities between vertices may not be necessarily

simple. The most straightforward manner to compute a similarity index between two vertices is by using adjacency information, i.e., the overlap of their neighborhoods [124].

The method described in Chapter 7 to simplify process models using duplicate tasks involves clustering vertices on a transition system by their context.

2.2 Process mining

The digital data revolution that is taking place worldwide requires new algorithms that enable acquiring value from the vast amount of data stored by the current technology. *Process-Aware Information Systems* (PAIS) are at the center of this revolution, since they are in charge of monitoring processes taking place in our daily life, like banking, municipalities, shopping, health care, etc. Event data, recorded in a PAIS in the form of *event logs*, denotes the footprints of process executions, and is an important source of information for reasoning on how the PAIS interacts with its environment when running its processes.

The area of *Process mining* uses these logs to discover, analyze and extend process models [2]. Process models deliver valuable insight into the execution of the underlying processes. Models can be used to find errors in real-life systems, such as deadlocks. Bottlenecks and other factors influencing the response time of a system can be identified by using simulation techniques. A process model may also be used as description or specification of a PAIS.

Discovery, one of the major areas of process mining, fosters this goal by constructing abstract process models that describe the high level structure of the process. These models are automatically learned from the execution traces of the proces, without using any other a-priori information.

An additional important area in process mining is *conformance*, in which existing process models are compared with event logs generated by the same process. Conformance checking verifies if the behavior described by the model corresponds to the behavior observed in the event log. Deviations may be detected in either the model or the event log, indicating potential hazards in the execution of the PAIS. The area of *enhancement*, on the other hand, aims at changing or extending an existing process model to better reflect the behavior observed in the event log.

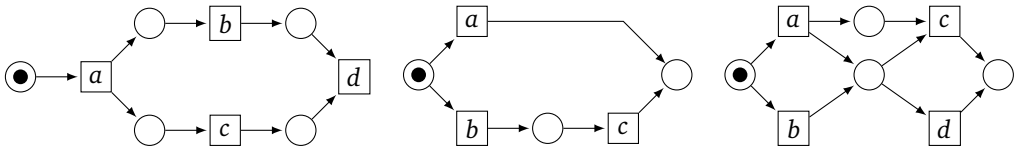
Traces and event logs

Event logs are the starting point to apply process mining techniques, guided towards the discovery, analysis or extension of process models. Informally, an event log is a set of traces, each of them being the footprint of a single execution

Trace
$abcd$
$acba$
$aaaa$

Parikh vector			
a	b	c	d
1	1	1	1
2	1	1	0
4	0	0	0

Table 2.2: Event log and Parikh vectors for each trace.



(a) Marked graph. **(b)** Free choice. **(c)** Non-free choice.

Figure 2.3: Petri net types.

of a process. Traces, thus, are a chronological sequence of events, such as "*Request rejected*". Usually we will refer to the different types of events with a single letter, e.g. $A, B, \ldots$.

Let Σ be an alphabet of *events*. A *trace* is a word $\sigma \in \Sigma^*$ that represents a finite sequence of events. An *event log* $L \in \mathscr{B}(\Sigma^*)$ is a multiset of traces[1].

Given a trace $\sigma \in \Sigma^*$, the *Parikh vector* of σ, $\psi(\sigma) : \Sigma \to \mathbb{N}$ maps every event $e \in \Sigma$ to the number of times it appears in σ. Table 2.2 illustrates this with an example. Parikh vectors are an important concept in transition systems, as will be seen in the rest of this thesis.

Events usually contain additional attributes, such as the timestamp or the actor that initiated the event. However, this work centers on the control flow itself, and thus event attributes are not used. In many scenarios we propose how the work could be improved by the use of such additional information.

2.2.1 Process models

Process models are formalisms to represent the behavior of a process. Among the different formalisms, Petri nets are perhaps the most popular, due to its well-defined semantics. In this thesis, we will primary focus on Petri nets as a process model, although some of the work may be adapted to other formalisms like BPMN [140], EPC [88] or similar.

Petri Nets

A Labeled Petri Net [112] is a tuple $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$, where P is the set of places, Σ is the alphabet of labels (corresponding to events), T is the set of transitions, $\mathscr{L} : T \to \Sigma \cup \{\tau\}$ assigns a label (or the empty label τ) to every transition, $\mathscr{F} : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation, and m_0 is the initial marking. A marking is an assignment of a non-negative integer to each place. If k is assigned to place p by marking m, denoted $m(p) = k$, we say that p is marked with k tokens. Given a node $x \in P \cup T$, the set ${}^\bullet x = \{y | \mathscr{F}(y, x) \geq 1\}$ is the pre-set of x, while $x^\bullet = \{y | \mathscr{F}(x, y) \geq 1\}$ is the post-set of x.

A transition t is *enabled* in a marking m when $\forall p \in^\bullet t, m(p) \geq \mathscr{F}(p, t)$. When t is enabled, it can *fire* by removing $\mathscr{F}(p, t)$ tokens from each place $p \in^\bullet t$ and putting $\mathscr{F}(t, p)$ tokens to each place $p \in t^\bullet$. A marking m' is *reachable* from m if there is a sequence of firings $t_1 t_2 \dots t_n$ that transforms m into m', denoted by $m[t_1 t_2 \dots t_n \rangle m'$. A sequence $t_1 t_2 \dots t_n$ is *feasible* if it is firable from m_0. A trace σ *fits* N if there exists a feasible sequence in N with the same labels.

A Petri net is *live* if for every marking m reachable from m_0, and $\forall t \in T$, there is a marking m' reachable from m which enables t. A Petri net is *k-bounded* if for each $p \in P$ and for every reachable marking m, $m(p) \leq k$. A 1-bounded Petri net may also be referred to as *safe*.

In a Petri net, a *choice* is a place with more than one output transition. Two transitions are said to be *concurrent* if they do not have dependencies between them, i.e. they can fire in any order.

A transition labeled with the empty label τ is called a *silent* transition. A *duplicate* task is a transition with the same label as some other transition in N.

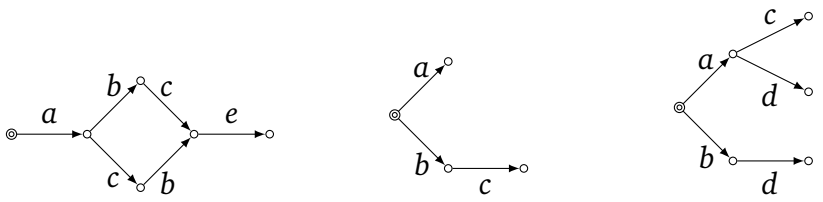
A set of restrictions on the structure of Petri nets define several classes of Petri nets. A Petri net N is a *Marked Graph* if $\forall p \in P : |{}^\bullet p| \leq 1 \wedge |p^\bullet| \leq 1$. It is a *Free-Choice* net if $\forall p_1, p_2 \in P : p_1^\bullet \cap p_2^\bullet \neq \emptyset \Rightarrow |p_1^\bullet| = |p_2^\bullet| = 1$. Note that every marked graph is a free-choice net. Figure 2.3 illustrates these concepts. In Fig. 2.3c, the choice between a, b is free, but the choice between c, d is not.

Workflow nets

We also introduce two additional two classes of Petri nets in which the starting and ending markings are clearly delimited by special *source* and *sink* places.

A *workflow* net [1] is a Petri net $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$ with exactly one source place $i \in P$, with ${}^\bullet i = \emptyset$, and exactly one sink place $o \in P$ with $o^\bullet = \emptyset$. In addition, in a workflow net there is a path from i to every other node $n \in P \cup T$, and a path from each node $n \in P \cup T$ to o.

[1] $\mathscr{B}(A)$ denotes the set of all multisets over A.



(a) **(b)** **(c)**

Figure 2.4: LTS corresponding to the Petri nets in Fig. 2.3

The Petri nets in Fig. 2.3b and 2.3c are workflow nets. However, the Petri net in Fig. 2.3a is not, since there is no sink place.

A workflow net is *sound* [6] if and only if it satisfies the following properties:

- *Option to completion*: from every marking m reachable from m_0, a marking m' with $m'(o) > 0$ can always be reached. Thus, the ending marking is always reachable.

- *Proper completion*: for any reachable marking m where $m(o) > 0$, and $\forall p \in P$ with $p \neq o$, $m(p) = 0$. That is, once the ending marking has been reached, no other transition can fire.

- *No dead transitions*: every transition $t \in T$ is enabled in at least one reachable marking.

These properties ensure that a sound workflow net is both bounded and live for all markings except the ending markings, where $m(o) > 0$. Sound workflow nets are heavily used in process mining because their semantics are similar to real-life processes.

2.2.2 Labeled Transition Systems

A finite labeled transition system is a tuple $A = \langle S, \Sigma, T, s_0 \rangle$ where S is a finite set of states, Σ is the alphabet of labels, $T \in S \times \Sigma \times S$ are the transition relations between states, labeled with Σ, and s_0 is the initial state. A transition system may also be interpreted as a directed graph where S is the set of vertices and T is the set of edges.

We use $s \xrightarrow{e} s'$ as a shorthand for the arc $(s, e, s') \in T$. Similar to Petri nets, a trace $\sigma = e_1 e_2 \dots e_n$ *fits* LTS A if there exists a sequence $s_1, s_2, \dots, s_n \in S$ with $s_0 \xrightarrow{e_1} s_2 \xrightarrow{\dots} s_{n-1} \xrightarrow{e_n} s_n$.

Definition 2.1 (Excitation Set)**.** For a given LTS A and event $e \in \Sigma$, we define the *Excitation Set* of e as the set of states in which e is enabled, i.e.,

$$ES(e) = \{s \in S \mid \exists s' \in S : s \xrightarrow{e} s'\}.$$

The following definitions formalize causality relations between two events:

Definition 2.2 (Concurrency and conflict)**.** Two events a, b are *concurrent* if there are four states, $s_1 \ldots s_4$ in S such that $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_4$ and $s_1 \xrightarrow{b} s_3 \xrightarrow{a} s_4$. In this case we will also say that a, b are concurrent in s_1. Two events a, b are in *conflict* if there is a state $s \in ES(a) \cap ES(b)$ and a, b are not concurrent in s.

Definition 2.3 (Free-choice conflict)**.** Two events a and b are in *free-choice conflict* if they are in conflict and $ES(a) = ES(b)$. In this situation the two events are always enabled or disabled simultaneously, which corresponds to a similar situation in Free-Choice nets.

Definition 2.4 (Trigger events)**.** Given two states s_1, s_2 with $s_1 \xrightarrow{a} s_2 \in T$, we say *a triggers* another event b iff b is enabled in s_2, but not in s_1. In a sense, a triggering b implies a causality relation between the two events. Analogously, we say *a disables b* iff b is enabled in s_1, but not in s_2.

Definition 2.5 (Persistence)**.** An event $e \in \Sigma$ is *persistent* if no event $f \neq e$ disables it.

A bounded Petri net can be transformed into an LTS by creating a state for every reachable marking in the net, and arcs according to the enabled Petri net transitions in each marking. Figure 2.4 shows the LTS associated to the Petri nets from Fig. 2.3. The opposite problem, however, is known as the *synthesis* problem [45, 60], and is not as straightforward for most Petri nets types.

2.2.3 Conformance checking

An important set of techniques in process mining is *conformance checking*, which compare the observed (log) and modeled behavior in order to evaluate the model. There are four quality dimensions for comparing model and log: *replay fitness*, *simplicity*, *precision*, and *generalization* [2]. While the four dimensions are not entirely orthogonal, balancing them is an important aspect to produce high-quality models for real-life processes [30].

Replay fitness

The *replay fitness* of a model indicates how good the model can reproduce the behavior of the process as observed in the event log. A model has perfect replay fitness if all traces in the log can be replayed by the model from beginning to end. This may not be necessary in all scenarios, e.g., if the event log contain noise [7]. Still, fitness is considered the most important metric.

Several metrics for precision exist in the literature [31]. In this thesis we will use the definition provided by [10], which computes an optimal alignment between the log and trace before calculating the fitness score, providing a more fine-grained evaluation in the presence of small deviations.

Simplicity

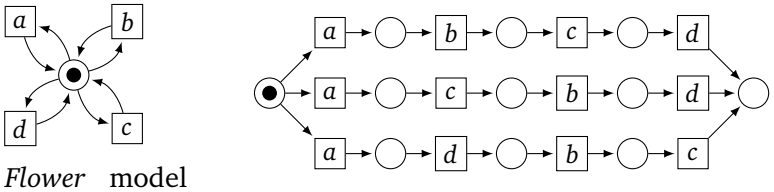
The *simplicity* of a model evaluates how easy it is to analyze and understand it. The simplest model that can explain the behavior seen in the log is the best model, a principle known as Occam's Razor. On the other hand, complicated models, with a high number of elements and dense control flow structure, prevent the extraction of useful insight from process mining. These complicated models are often called *spaghetti models*, such as the one shown in Fig. 1.2.

Process discovery algorithms may derive spaghetti process models in various situations, e.g., when the log represents a complex process with hundreds of different event classes or when the log contains noise. Other problems like *concept drift* (the log contains the executions of different versions of the process model) or *vertical event granularity* (event classes from different hierarchies coexist in the log) may also cause the derivation of a dense process model. Unfortunately, the aforementioned situations happen often in real life [79]. Thus, the discovery of simple process models and the simplification of larger models is a significant challenge when presenting data obtained by process mining.

The most common methods to estimate the complexity of a process model involve the size of the model or the average degree of its vertices [108]. In this work, however, we propose the use of the crossing number of a graph as measure of complexity, a concept closely related to the planarity of a graph. For a formal definition, we refer to Section 2.1.1.

Precision

Fitness and simplicity alone are not sufficient to judge the quality of a discovered process model. For example, it is very easy to construct an extremely simple Petri net (*flower* model, as in Fig. 2.5a) that is able to replay all traces in an event log. However, this model also replays any other event log referring to the



(a) *Flower* model (underfitting).

(b) *Trace* model (overfitting).

Figure 2.5: Underfitting and overfitting Petri net examples.

same set of activities. Thus, the model is not useful for describing the behavior of an specific process.

Precision compares how much behavior is reproduced by the model that is not present in the log. A model is precise if it does not contain behavior that has not been observed in the log. A model that is not precise is *underfitting*, such as the flower model.

In this work, we use the metric proposed in [111], based on the concept of *escaping arcs*. A escaping arc represents a choice that is available in the model, but never taken while replaying the event log.

Generalization

On the other hand, event logs contain only observed behavior and many traces that are possible may not have been captured in the logs. Thus, it may be undesirable to have a model that only allows for the exact behavior seen in the event log.

In contrast to precision, a model should *generalize* and not restrict behavior to just the examples seen in the log. A model that does not generalize is *overfitting*. Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds partial behavior. A good example is a *trace* model (as in Fig. 2.5b). The model may explain a particular sample log with very high precision, but there is a high probability that the model will be unable to explain the next batch of cases.

In this work we will generally use the metric provided in [30], which penalizes models where most parts are visited very infrequently when replaying the logs. If infrequently-visited parts are prevalent in the model, it is unlikely that new, slightly different traces will fit.

Finding a good balance between overfitting and underfitting models is one of the challenges in process mining [5].

2.3 Very-Large-Scale Integration design flow

In this thesis, Chapters 3 and 4 focus on using graph mining during physical design of large-scale chip designs. This section provides a tour of the basic design flow of Very-Large-Scale Integration (VLSI) circuits to help understand the context of these chapters.

Modern chips are extremely huge and complex. In the present day, chip designs with billions of transistors are not uncommon (current commercial designs have well over 7 billion transistors [48]). These designs combine hundreds of cores, on-chip memories, routers, interconnects and many other components in a single chip, possibly pre-designed by third parties. Thus, many of the challenges found during the design of modern chips will revolve around managing this complexity.

Because of such complexity, the design process for any chip is partitioned in interrelated tasks, both in space (e.g. the different modules of the chip) and time (e.g. early core architecture design versus gate-level).

Hierarchy and *abstraction* are two concepts often used during VLSI design. Large systems are often partitioned into many structures that can be recursively partitioned into smaller, independent units. To save time, it is preferable to have many instances of the same module versus many different modules. Thus, the rise of the Chip multiprocessor (CMP), which allows the construction of highly performant machines at a fraction of the cost by replicating hundreds of pre-designed processing tiles.

Furthermore, different teams usually work concurrently on tasks that normally would be done sequentially. Dependencies between stages are lessened by the use of abstraction mechanisms (e.g., so that the designer of the core does not need to be involved into logic gate internals).

Even with these abstraction mechanisms, however, there are still times when the output of one stage is required as input of another stage. For example, the designer of the core microarchitecture needs to know the characteristics of the physical design, including clock speed, in order to properly create a pipelined architecture. Experienced engineers are required in this case in order to create *estimations* so that all the teams have something to start with. These estimations are refined continuously as the design process advances.

Current VLSI design flows usually split tasks into 4 big stages, each composed of many smaller stages. These are not fixed and it is common for the stages to be rearranged depending on the implementation of the design flow. An overview of the 4 stages [139] can be seen in Fig. 2.6, and in the following list:

1. *Architectural design* (also *functional* or *black-box* design), which is the first stage, involves deciding on the general (or *system-level* structure) of the

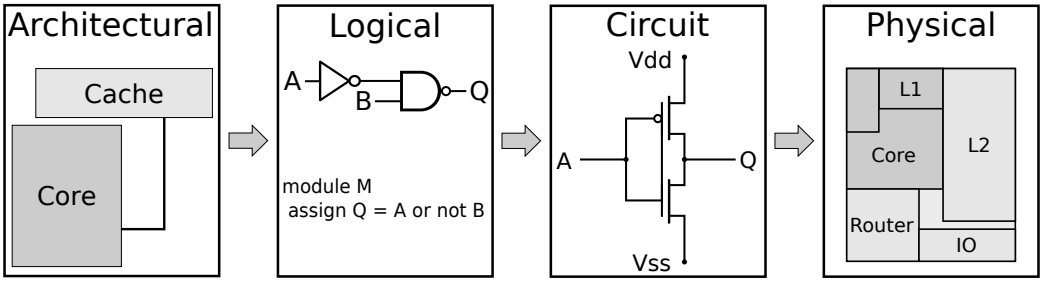


Figure 2.6: Summary of the main VLSI design flow stages.

chip. In the case of CMPs, this stage involves finding the promising configurations (combinations of different core architectures, on-chip memories, etc.) that satisfy the performance, cost and other requirements.

2. *Logic design* describes how the diverse components of a chip work. For a core, the behavior of the ALU, floating point, etc. is described.

3. *Circuit design* considers how to implement the logic design into an electronic circuit (selection of transistors, etc.)

4. *Physical design* layouts all required transistors and wires and thus determines the final geometry of the chip.

Each of these stages will be described in more detail in the following sections.

2.3.1 Architectural design

During architectural design the system-level structure of the chip is defined. A designer usually has a set of required metrics (desired performance, etc.) and a limited budget (in cost, chip area, power usage, etc.). In traditional CPU design, this involved deciding on parameters such as the length of the pipeline, the memory hierarchy, whether to use out-of-order execution, etc.

For example, in the case of CMP design, it is generally assumed that the designer has a library of components (cores, caches, etc.) at its disposal, and that he/she needs to decide which (and how many) of those components are to be placed on the CMP, and how to interconnect each of those components.

Because these components may have never been built yet, or have been manufactured before but using different physical parameters, there might be no real-world estimations of their parameters. Additionally, the high number of possible combinations and permutations of the components in the library often generates a design space with billions of possible configurations. For this reason, during architectural design fast estimators for each of the diverse component

metrics are necessary. Interpolation, analytical modeling or simulations are used, depending on the accuracy required. These methods will be discussed in Section 3.3.

2.3.2 Logic design

At this stage, the required functionality is implemented in terms of boolean functions. The interfaces (inputs and outputs) for the different modules are also usually defined during this stage.

However, logic is not necessarily specified in terms of the traditionally used logic gates. This is because during physical design there is a step, technology mapping, in which, depending on the manufacturing technology used, there might be be more efficient ways to convey boolean functions that by using the standard logic gates.

2.3.3 Circuit design

During circuit design, a electronic circuit is created from the previously created logic. Exactly at which level the circuit is designed depends on the manufacturing technology.

When using HDLs, this process is usually fully automated using a tool called *synthesizer*, which handles an HDL input and converts it to the required representation. The output is usually a list of logic gates and the connections or *nets* between them (a *netlist*). *Technology mapping* is the process by which the high-level logic gets mapped into lower-level elements such as *cells*.

2.3.4 Physical design

At the start of physical design, the design is represented in terms of a *netlist* including all low-level components (cells, gates, transistors, ...) with their shapes and the appropriate interconnections between those components. The result of physical design is a physical *layout*, including both the positions of each of the design components (the *placement*) as well as the paths for each of the interconnections (the *routing*).

From this stage and until the final circuit, limitations of the physical world are now taken into account. Depending on the implementation technology, *design rules* prohibit certain layouts. For example, by setting minimum distance between wires or forcing insertion of repeaters whenever wires exceed a certain maximum length [75].

Physical design is often split in the following sub-stages [86]:

1. *Floorplanning* is often the first step during physical design. A *floorplan* provides an estimation of the shapes and locations of the major units in the circuit. For the case of CMPs, this usually confers the locations of the individual cores, memories, etc. In a hierarchic fashion, a single core might also have a floorplan containing the locations for the arithmetic and logic units, etc.

   Because no physical information about the innards of the components is available at this early stage of physical design, floorplanning often uses area estimations as inputs. As more accurate estimations become available during later physical design stages, the floorplan is updated.

   Additionally, creating a floorplan allows early estimations of the length of the largest nets. This is often called *wire planning*.

2. *Placement* defines the final locations for the individual low-level units (e.g. cells) inside each high-level block. Unlike floorplanning, regular structures such as grids are often considered because most of the cells have similar sizes or sizes in multiples of a common base unit.

3. *Clock tree synthesis*. In sequential circuits, keeping the propagation delay of the clock signal to a minimum is usually mandatory to avoid *clock skew*. Because of this importance and singularity, the clock signal is usually the first net to be routed.

4. *Global routing* estimates and reserves the required routing resources that are required by the interconnections. Usually, global routing considers only those nets that use the most resources. Each net is given a estimated width depending on the design rules and the number of wires composing it, but individual wires are not considered. These estimated paths reduce the search space for the detailed routing stage that will come next.

5. *Detailed routing*, on the other hand, calculates the final routes for each individual wire of the chip, at the same time verifying that the previously generated global routing is feasible.

6. *Timing analysis* checks that all of the design's timing constraints are satisfied. Different logic paths have different timing requirements, and during this process every factor that alters net delay is taken into account: length of wires, repeaters, transistor sizing, etc. If any of the constraints is not met, the design is reiterated, restructuring the layout of the critical paths. Thus, even the floorplan might need to be changed and the physical design process restarted.

In this thesis, Chapter 3 will center on the topic of physical design oriented towards CMPs, while Chapter 4 will expand the proposed methods towards general VLSI design.

2.4 Asynchronous circuits

One the focus of this thesis will be to propose the concept of specification mining for asynchronous controllers (Chapter 8). This chapter introduces the context of this contribution by providing an overview of asynchronous circuit design.

Asynchronous circuits are logic circuits that do not rely on a global synchronization signal, the clock, to dictate when signals are sampled. Instead of clocks, asynchronous circuits favor *handshaking* to synchronize the different components in a circuit. Asynchronous logic offers many advantages [46], among them: increased performance, reduced power consumption, and better composability and modularity.

While early computers were mostly asynchronous, the prevalence of asynchronous design today has significantly decreased in favor of synchronous design and global clocks. In part, this is because of the complexity usually associated with the design of asynchronous circuits. There are few asynchronous design tools in comparison with the huge amount of well-entrenched synchronous tools.

In asynchronous circuits two main parts are often distinguished: the *data path* and the *control*. While the former comprises wide units performing arithmetic or other types of transformations on the data processed by the circuit, the control unit determines and generates the diverse control signals that will configure the data path units. As control manages the synchronization requirements, this thesis will exclusively focus on the analysis of control circuits.

We define a circuit C as a tuple $C = \langle X, G, s_0 \rangle$ where:

- $X = I \cup O \cup Z$ is the set of signals, with I, O and Z being pairwise disjoint sets that represent the input, output and internal signals of the circuit, respectively.
- $G : (O \cup Z) \rightarrow f(X)$ is a set of gates that assigns a Boolean function to each non-input signal of the circuit. We denote by $f_{x_i}(X)$ the Boolean function assigned to signal x_i.
- s_0 is a binary vector representing the value of the signals at the initial state.

The *environment* of a circuit reacts to the outputs from the circuit and sends new inputs to it.

2.4.1 Operating modes and delay models

A *hazard* is a deviation from the expected behavior of a circuit caused by the delay present in real-word gates and wires. Hazards are one of the main challenges during asynchronous circuit design. To help manage the complexity associated with asynchronous circuits, several design styles have been developed. Within each style, different models and assumptions are made about the timing and behavior of physical elements of the circuit: gates, wires, and the environment.

This section overviews some of the common styles. Broadly, we distinguish between *delay models*, i.e. assumptions about the operating delays of gates and wires, and *operating modes*, models about the interaction between a circuit and its environment. For an exhaustive analysis into the design of asynchronous circuits, we refer the reader to existing documentation [17, 46].

Operating modes

Operating modes model the interaction between the circuit and its environment. In *fundamental mode* [77], inputs from the environment are constrained to change only when all the outputs are stable, i.e., the environment allows the circuit to stabilize before generating new inputs.

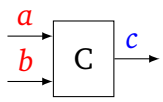
Burst mode [53] is a related operating mode in which a burst of multiple sequential input changes are allowed. However, when all inputs in the burst have changed, the environment must wait for the circuit to stabilize before starting a new burst.

Compare with *input-output mode* [110], in which new inputs may occur at any time, as part of specified responses to changes in the outputs. Thus, input and output changes may be concurrent.

Delay models

The delay model defines the assumptions made, during the design, about delays in gates and wires. Strong assumptions may simplify the design flow, while less strict ones generally lead to designs that are more robust to manufacturing process variations. In this section we describe some of the common models.

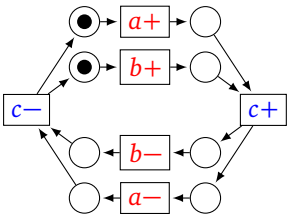
- In the *bounded delay* model, delays of both gates and wires are assumed to be lying within given minimum and maximum bounds. The circuit is guaranteed to work correctly if these bounds are satisfied. While this may lead to smaller circuits, extensive analysis is needed to ensure all bounds are met in all conditions.



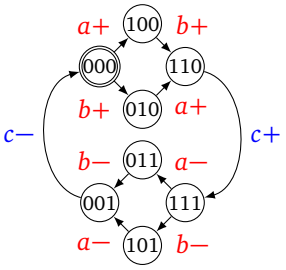
(a) C-element.

a	b	c_n
0	0	0
0	1	c_{n-1}
1	0	c_{n-1}
1	1	1

(b) Truth table.



(c) Signal transition graph.



(d) State graph.

Figure 2.7: C-element and specifications.

- A *speed-independent* (SI) circuit works correctly even in the presence of arbitrary but finite delay on its gates. However, it works with the assumption that wires, on the other hand, are ideal, i.e., with zero delay [110].
- A circuit is *delay-insensitive* (DI) when its correct operation does not depend on neither the delays in the gates nor the wires. While this is an extremely interesting class of circuits because of its robustness, unfortunately it has been proven that very few circuits can truly be made delay insensitive [106, 121].
- *Delay-insensitive interfacing* proposes that only inputs to the circuit are required to be handled in a delay-insensitive fashion. This is a compromise in order to alleviate the impracticality of DI circuits while still allowing long interconnects. The assumption is that even if it is not practical to assume that long wires have zero delay, a designer may still keep control on the delays of short internal wires [121].

2.4.2 Signal transition graphs

The most common design methodologies for asynchronous controllers involve first specifying the behavior of the circuit and the necessary requirements from the environment. From these specifications, automated tools generate hazard-free implementations of the circuits [46].

Asynchronous circuits are intrinsically concurrent, and thus naturally suited to Petri nets (Section 2.2.1), one of the most powerful formalisms for reasoning about concurrent systems. A *Signal Transition Graph* (STG) is formal model based on Petri nets for conveying the behavior of an asynchronous circuit.

Given a circuit $C = \langle X, G, s_0 \rangle$, a Signal Transition Graph is a labeled Petri net $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$ in which $\Sigma = X \times \{+, -\} \cup \tau$. Thus, each transition label corresponds to either the transition (rising or falling) from a signal of C, or a silent event τ that does not change the state of the circuit. By x^+ and x^-, we will distinguish the rising and falling transitions of signal $x \in X$ respectively.

Figure 2.7c shows an example STG of a *C-element*, one of the fundamental components in asynchronous circuits, whose truth table is depicted in Fig. 2.7b. The C-element is a stateful element that sets its output to 0 when both inputs are 0, and to 1 when both inputs are set to 1. In any other input combination, the output does not change.

2.4.3 State graphs

An STG is just a succinct representation of (a part of) the behavior of the circuit, which focuses on the causality relations amongst events. An *state graph* also represents the behavior of the circuit by enumerating all of its possible states and transitions between states as a Labeled Transition System (LTS). While this may result in a much larger representation than an STG, many algorithms require exhaustive explorations of the state space. For the full definition of an LTS, we refer to Section 2.2.2.

Given a circuit $C = \langle X, G, s_0 \rangle$, an state graph of C is an LTS $A = \langle S, \Sigma, T, s_0 \rangle$ in which:

- $S = \{0, 1\}^n$, with $n = |X|$, the set of binary vectors representing all possible states of the signals.
- $\Sigma = X \times \{+, -\}$, i.e. the set of signals of the circuit plus the direction of the transition.
- $T = \{s_1 \xrightarrow{x} s_2\}$ with $s_1, s_2 \in S$ and $x \in \Sigma$, is the set of transitions.
- The initial state s_0 coincides with the initial state of the circuit.

Given a state $s = (x_1, \ldots, x_n)$, we denote by $s(x_i)$ the value of signal x_i in s. Given a state $s = (x_1, \ldots, x_i, \ldots, x_n)$, we denote by $s^{\neg x_i} = (x_1, \ldots, \neg x_i, \ldots, x_n)$ the state in which the values of the signals are identical to the ones of s except for x_i, that has the complementary value. Notice thus that for each $s_1 \xrightarrow{x} s_2 \in T$, $s_2 = s_1^{\neg x}$ and $s_1 = s_2^{\neg x}$.

Figure 2.7d shows an example of the state graph associated to the behavior of the C-element described by the STG in Fig. 2.7c.

2.5 Mathematical optimization

Many of the methods described in this thesis involve optimization problems. This section provides a brief introduction to two commonly used subfields of mathematical optimization, albeit it is not indented to cover the finer details. The approaches described in this section will be frequently used in this thesis when encoding, e.g., desired structural constraints in graphs.

An optimization problem involves finding the *best* solution out of a set of feasible solutions. The search space of feasible solutions is delimited by a set of constraints, such as formulas, inequalities, etc. Generally, the best solution is that which maximizes (or minimizes) the *value*, usually defined as a real function.

2.5.1 Boolean satisfiability

A *formula* P is a combination of boolean variables (denoted by $p, q, \ldots$) built using the 3 logical operators *and*, *or*, *not* (represented respectively by $\wedge, \vee and \neg$). An *interpretation* I of P is an assignment of $\{0, 1\}$ to each variable in P. I *satisfies* P iff the evaluation of P under I is 1. P is *satisfiable* if it is satisfied by at least one interpretation. Generally, formulas are written as conjunctions of *clauses*, which are disjunctions of (possibly negated) variables.

Satisfiability (SAT) is the problem of determining, given a formula P, whether there is an interpretation I that satisfies it. SAT is a well-known NP-complete problem, with all known algorithms having worst-case exponential cost on the size of P [24].

The *maximum satisfiability problem* (MaxSAT) is the optimization version of SAT. Given an formula P, MaxSAT involves finding an interpretation that maximizes the number of clauses that evaluate to 1. Inversely, the *minimum satisfiability problem* (MinSAT) finds satisfying interpretations that minimize the number of clauses that evaluate to 1. As typical extensions, weights can be added to individual clauses, allowing for arbitrarily complicated optimization goals [24]. MaxSAT and MinSAT are heavily used as a natural way to model many optimization problems.

2.5.2 Linear programming

A linear inequality $a \cdot x \leq b$ is defined by a vector $a \in \mathbb{R}^n$ and a constant $b \in \mathbb{R}$. A *linear programming problem* (LP) is a set of linear inequalities plus a linear function that needs to be maximized, called the *objective function*. It is usually represented as:

$$\begin{aligned} &\text{maximize } c^T \cdot x \\ &\text{subject to } A \cdot x \leq b \end{aligned}$$

where A is a matrix with a row for every linear inequality, b contains the constant terms of the inequalities, and c is a vector with the coefficients of the objective function. A solution of the LP, thus, is a vector $x \in \mathbb{R}^n$ that satisfies all linear inequalities. From the potentially infinite set of solutions, a solution x is optimal if it also maximizes the objective function $c^T \cdot x$ over the set of all solutions. An LP is *feasible* if it has at least one solution.

There are algorithms to solve LPs in polynomial time [91]. However, the most common algorithm used is *simplex*, which is exponential in the worst case, although performs efficiently in practice [93].

An *integer linear program* (ILP) is an LP in which some of the variables are constrained to have only integer values. Unlike LP, ILP is NP-complete. There are many available methods to solve ILP. ILP solvers may also be used to solve MaxSAT problems [52].

Chapter 3

Physical planning for the architectural exploration of Chip multiprocessors

At the early stages of the design of a CMPs, physical parameters are often ignored and postponed for later design stages. In this chapter, the importance of physical-aware system-level exploration is investigated.

Additionally, this chapter presents an strategy for deriving chip floorplans that include physical constraints specific for tiled hierarchical CMPs. Over-the-cell routing is also used as a major area savings strategy. Wire planning of the on-chip interconnect is also studied, as its topology and organization affect the physical layout of the system.

This chapter will be structured as follows. Section 3.1 introduces and motivates the topic, evaluating the impact of the physical aspects on the selection of architectural parameters. Section 3.2 reviews the existing literature. Section 3.3 describes the proposed combination of architectural exploration and physical planning. Section 3.4 and Section 3.5 give details on how to perform efficient physical planning, centering on floorplanning and wire planning respectively. Finally, in Section 3.6 the proposed flow is evaluated, with future work and conclusions discussed in Section 3.7.

3.1 Motivation

This section will justify the need to adapt the traditional physical design flow with physical planning constraints during CMP design. In addition, it will show how using this new physical design flow during early design stages improves the quality of designs and minimizes the number of design reiterations.

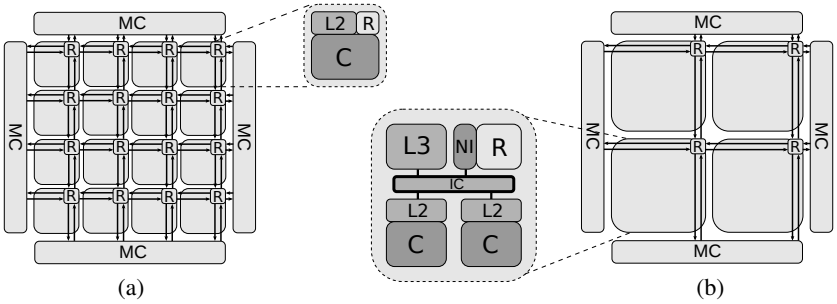


Figure 3.1: Two different configurations for a CMP: (a) tiled flat, (b) tiled hierarchical.

3.1.1 Chip multiprocessors

During the past decades, many-core chip multiprocessors [16] have become the major trend in designing scalable computing architectures. Multiple processing units with distributed memory combined with power saving schemes are the platforms used today for exploiting application parallelism while keeping power consumption under control.

A CMP integrates more than one computing *core* in a chip. Each of the cores is individually similar to the one inside a single-core processor, containing and arithmetic and logic unit, registers, private cache, a datapath, and a control unit. In addition to private caches, however, a CMP also contains caches shared by two or more cores, and possibly more than one input/output ports to external memories. The *interconnect* provides communication between cores, shared caches and I/O. The latency and throughput of the on-chip interconnect is crucial to the overall performance of a CMP. Thus, it is a very important factor to consider during design. Networks-on-Chip (NoCs) [50] have been firmly established as a the paradigm of choice for scalable interconnects.

Tiled CMP architectures facilitate the design process of CMPs by offering a rapid way to assemble platforms with tens or hundreds of cores. A tiled CMP is constructed by replicating pre-designed tiles [16, 18, 76]. An example is shown in Fig. 3.1a, where each tile contains a single core (C), cache (L2) and a router (R) that connects it with neighboring tiles.

Nevertheless, challenges appear when constructing many-core CMPs using tile replication, as the two-dimensional mesh structure means increased distance and power consumption for every new core. To overcome this problem, *hierarchical* CMP organizations have been proposed to better exploit spatial locality [16, 51].

Figure 3.1b depicts the block diagram of a tiled hierarchical CMP with 8 cores and distributed L3 cache. The chip is organized as a 2×2 regular grid of

tiles (*clusters*), each one including two computing cores (C) with private cache (L2), a distributed shared cache (L3), a router of the global mesh (R) and a local interconnect (IC). The two-level hierarchical interconnect constitutes the backbone of this architecture. The purpose of the global mesh is to provide inter-cluster communication, as well as access to the memory controllers (MC). Intra-cluster communication is supported by low-latency rings that significantly improve the bandwidth of the system given the locality of memory references inherent to the applications.

The problem of *system-level design* for a many-core CMP consists of selecting high-level architectural parameters (e.g., number of cores, size of cache, topology of the interconnect, etc.) so as to maximize system performance for the selected workload and satisfy the design constraints (e.g., area and power). System-level design is performed early in the design cycle. The main complexity of this task is determined by the vast space of potential architectural configurations and the inaccuracy of the models to represent the components of the system and the workload.

To alleviate the problem complexity, most strategies for architectural exploration disregard physical parameters and postpone them to later design stages. However, in this chapter we show that physical planning has a non-negligible impact on performance and area of a CMP. In the rest of this chapter we will propose methods for *floorplanning* and *wire planning* of tiled hierarchical CMPs and show the impact of physical parameters in the configuration of the architecture.

3.1.2 Physical design flow for CMPs

The problems of physical planning for CMPs are related to traditional problems in VLSI physical design [125]. CMP floorplanning is similar to classical VLSI floorplanning, while wire planning is more common with global routing. However, there are several aspects inherent to tiled hierarchical CMPs which motivate us to extend existing approaches.

As shown in Fig. 3.1a, the tiled organization of CMPs reduces the floorplanning problem from chip to cluster level. However, the cluster floorplan has to satisfy the property of symmetry in the location of the North/South and East/West ports at the boundaries of the tile. This enables the construction of a full chip by replicating and abutting of tiles.

Floorplanning of the local interconnect introduces another complexity into the design. For example, when considering rings, it is required that the links between the ring routers (r) have *balanced* lengths to guarantee similar hop delays. If the link delays are imbalanced the communication through the ring may have a negative impact on performance.

A special type of constraints, such as *adjacency* or *maximum net delay* constraints are required to prevent certain components be placed far from each other. A typical example may be a core and its L2 cache. Placing a cache far from the core may increase its access delay and result into a significant performance penalty. While adjacency of the two components may appear as a too strict constraint, a weaker requirement of the inter-component distance to be less than one hop will be enough to assure no loss of performance.

An important observation is the recent tendency to design CMPs with wide links. Communication links of the on-chip interconnect may incorporate thousands of wires, aiming at transferring a complete cache line in one cycle. Given the ITRS prediction for minimal wire spacing [80], links of a global mesh can have a width of about 10^2 μm, occupying a significant amount of chip area.

One of the possible ways to alleviate the area overhead is to benefit from *over-the-component* routing. Some of the CMP components, such as memories, do not use all the metal layers available in the technology and, therefore, these available resources can be used to implement global nets across the chip.

In this scenario, the most complex components using all metals layers may act as blockages for over-the-component routing. Hence, one of the purposes of wire planning is to verify chip *routability*. Another purpose is the estimation of wire length, which is one of the main parameters when evaluating design quality [131].

3.1.3 Impact of physical planning in exploration

During architectural exploration, parameters for the system-level design of a CMP, such as the number of cores, size of cache, etc. are selected so as to maximize the performance of the chip and satisfy the area and power constraints. Physical planning usually comes after architectural exploration in the design flow, and thus, physical parameters are often disregarded during architectural exploration. However, we will show that disregarding this information at this level has a non-negligible impact in the performance and area of the chip.

Sample configuration

Let us assume that an architectural exploration tool has generated a configuration such as the one shown in Fig. 3.2. This configuration has a total of 224 identical cores, split in tiles of 4 cores each. In total, there are 56 tiles arranged in a 7×8 mesh.

Figure 3.2 does not include any physical information. We assume that these cores are predesigned and have an estimated area of 1.2 mm^2, including private

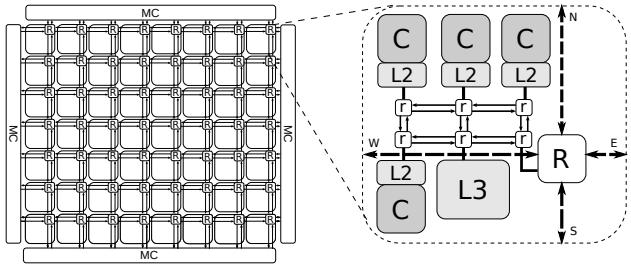


Figure 3.2: Structural representation of a hierarchical tiled CMP with two-level interconnect: a global mesh and a ring inside each tile.

L1 cache, with an aspect ratio of 0.8, where the aspect ratio r is defined as:

$$r = \frac{h}{w}$$

The layout of the cores can be flipped and rotated, but not resized.

At the same time, each core has an associated private L2 cache of 1 mm^2, and each tile has 1 mm^2 of shared L3 cache (thus having a total of 56 mm^2 of shared cache in the entire chip). Unlike the cores, we assume that the caches are *soft blocks*, where the width and height are not fixed as long as the area is kept constant. It is generally expected to at least have minimal and maximum aspect ratios that limit the available shapes.

Apart from cores and caches, each tile contains a router for the global mesh interconnect (R, 0.99 mm^2), and every component participating in a ring contains a corresponding ring router (r, 0.17 mm^2). Similarly to cores, we assume that every router is a hardblock with an aspect ratio of 1.

From the point of view of over-the-component routing, we also assume that cores and routers are complex components that will use all available routing resources (metal layers), while memories will leave at least two layers available for routing.

Using conventional floorplanning

When we consider the floorplan of the entire system, we face a problem with about 900 components, including cores, L2 and L3 caches and routers. However, in this work we deal with tiled hierarchical CMPs, which have several proven benefits by enabling a divide-and-conquer design strategy. Floorplanning, placement, routing, and timing closure are processes that can be applied to a single tile while guaranteeing correctness for the global system. For this reason, we will center on the floorplanning of a single tile.

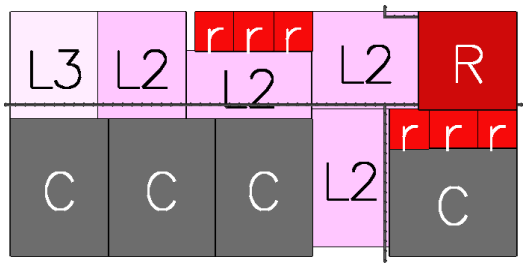


Figure 3.3: A minimal area floorplan for the configuration in Fig. 3.2

Figure 3.3 depicts a minimum-area floorplan that could be obtained by a conventional floorplanner such as CompaSS [37]. In this example, the total area of the tile is 12.52 mm^2. However, from the point of view of a hierarchical CMP, this floorplan has some undesirable problems:

1. Some cores are not adjacent to their private L2 caches, potentially increasing the communication latency between them. Similarly, there are long distances between some caches and the corresponding ring routers.

2. Ring routers for the local interconnect are not evenly separated. In a ring, the wire length of the longest hop dictates the maximum speed for the entire ring. If this distance is too long, some timing constraints might be violated. Therefore, it is desirable to minimize the length of each link hop separately instead of minimizing the total link length.

3. Assuming that cores (C) and the router (R) use all metal layers, the two rightmost ring routers (r) have no available routing area in their boundaries. Thus, the design cannot be routed without whitespace insertion.

Using CMP-aware floorplanning

An alternative floorplan is shown in Fig. 3.4. This floorplan has been generated using all the constraints and enhancements discussed in this work. Since area minimization is no longer the only objective, this floorplan has a 16% area increase (14.57 mm^2). However, all of the cores are now adjacent to their private L2 caches. Additionally, a route can be found between all the ring routers so that the the link length for each hop is always between 0.1 and 1.1 mm, and the distance between a component and its attached ring router is strictly less than 0.5 mm.

As an example, Fig. 3.5 shows a floorplan for the entire system, including all clusters, based on the cluster floorplan from Fig. 3.4.

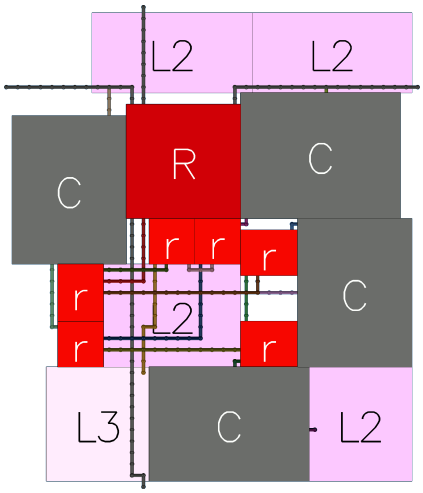


Figure 3.4: CMP-aware floorplan for the configuration in Fig. 3.2

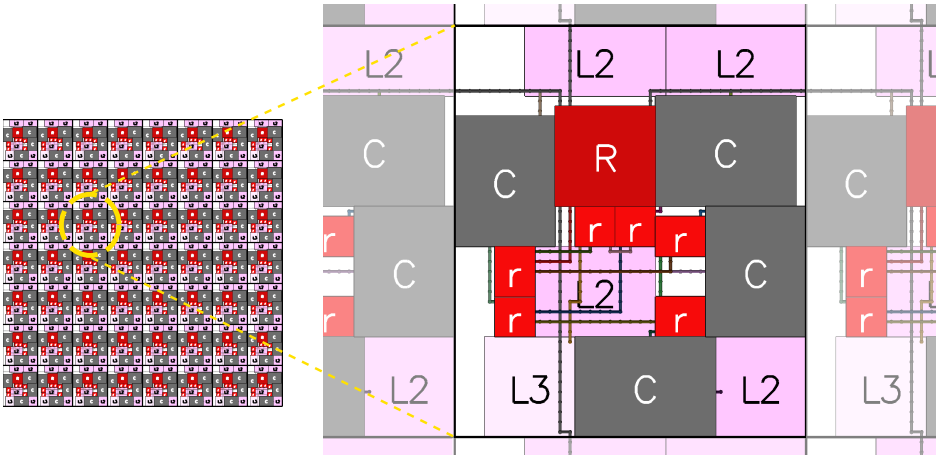


Figure 3.5: CMP-aware floorplan (full chip).

A 16% increase in area may induce an unacceptable overhead in manufacturing cost. This fact may encourage a designer to select an alternative architectural configuration, with a slightly lower performance, although with better floorplan properties.

3.2 Related work

Floorplanning as a part of the VLSI design flow has been extensively studied for decades. The traditional definition involves minimizing a linear combination of area and estimated wire length [85], leaving actual wire planning to posterior stages in the design process.

Hierarchical approaches to floorplanning have already been shown to reduce the algorithm runtime. Quite often hierarchical floorplanning is applied to the design of Systems-on-Chip (*SoCs*), for which every component can be considered as a fixed-size block. These blocks can be generated using fixed-outline floorplanners such as [12], while the system-level floorplanning can be solved using the traditional minimal area techniques such as [37]. In this work, we will instead exploit the regularity of tiled hierarchical CMPs.

When floorplanning a CMP, it might also be desirable to optimize factors other than area and wire length. Previous approaches exist that evaluate floorplans based on other qualities such as temperature minimization [109,123] or power consumption [129], using analytical models. For floorplanning at the system-level, [145] proposes a method that creates tile arrangements which minimize the overall wire length for several 3D topologies.

Floorplanning with constraints is also commonly considered in modern floorplanning [131]. For example, restrictions on the valid placement of blocks: adjacency constraints, limits in the distance between pairs of blocks, and objects in fixed positions [146]. Specifically considering CMP constraints at this stage, as in this chapter, is less common. In [142] the authors show how over-simplified models for those constraints (e.g., disregarding pin placement) produces sub-optimal floorplans, but only for classic bus-based interconnects.

On the integration of floorplanning with earlier design stages, the work in [20] incorporates a linear programming-based floorplanner into a synthesis framework for application-specific System-on-Chips. The floorplanner is used to obtain better area estimates.

The influence of physical information on system performance at the micro-architectural level was studied in [42]. The authors proposed physical planning to estimate area and link delay, which were then used to refine the accuracy of throughput estimations obtained by simulation.

3.3 Architectural exploration

This section overviews the flow for architectural exploration of CMPs and introduces the context for physical planning. Consider the problem of maximizing CMP performance (throughput) subject to a resource budget, i.e. constraints on area and power. The given formulation is an example of the architectural exploration problem with the objective of efficiently distributing the chip resources among the components of a multi-core system, e.g. cores, memories and interconnect.

The design space for exploration is specified through a set of models and design constraints. The *models* describe the behavior of individual components. There can be different models for cores characterizing different micro-architectural features that trade-off area, power and performance (in-order/out-of-order execution, multi-threading, etc). The memory models define the size, area and latency of different memory modules. The models for the interconnect define their physical and performance properties (latency, contention, etc).

The expected workload for the CMP requires another type of models that characterize the observable behavior produced by the generated memory patterns (memory locality, burstiness, etc). *Constraints* on power consumption and area are typically defined to confine the design space.

Exploration is a complex optimization problem due to the vast discrete space of architectural variables that determine the configuration of a CMP (e.g. number of cores, cache sizes, interconnect topology, link width). To handle this complexity, in this work we resort to a three-stage divide-and-conquer approach to solve the exploration problem. Figure 3.6 illustrates our methodology, with the main stages being the *architectural exploration*, *physical planning* and *validation*.

Architectural exploration

During the first stage, analytical models are used to rapidly prune the design space and generate a set of promising configurations in the area/power/performance space. The analytical model from [114] is used to evaluate CMP configurations and discriminate those with poor performance. Static and dynamic power are also evaluated using analytical approximations based on the area and activity of the CMP components [115]. The area is approximated as the sum of the areas of all components on chip.

Analytical models are used as a cost estimator for an iterative metaheuristic-based search to efficiently navigate through the design space. This space is described with a set of architectural variables and a set of *transformations* is defined to explore the neighborhood of any particular configuration. Some

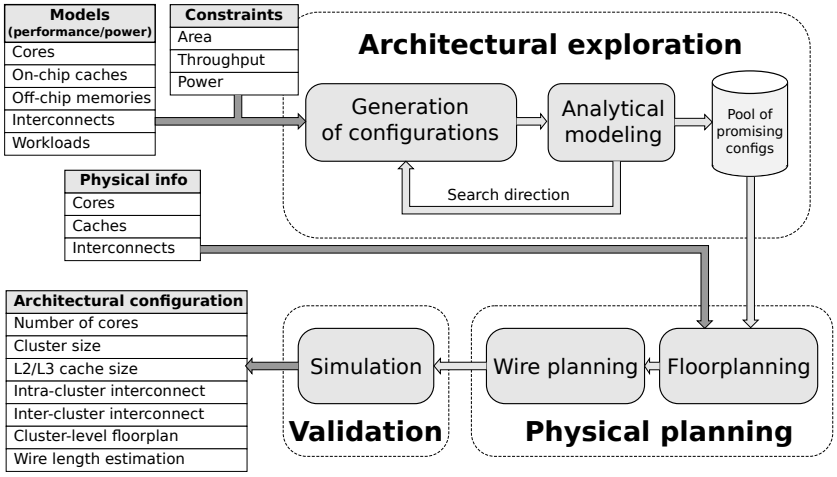


Figure 3.6: Modified architectural exploration flow that includes physical planning.

examples of transformations include modifying the dimensions of the top-level mesh, the number of cores per cluster or the topology of the local interconnect, among others. *Simulated Annealing* [92] and *Extremal Optimization* [26] are used to explore the design space by probabilistically applying transformations and tracking the best discovered solution.

Physical planning

The objective of this stage is to evaluate wire length and give a more accurate area estimation. The floorplanning and wire planning algorithms at this stage consider physical constraints for individual CMP components, such as the aspect ratio and the number of metal layers. This accuracy comes at the expense of a higher algorithmic cost, which is however tolerated by performing the planning for a moderate number of configurations, selected during the first stage.

Validation

Finally, the validation phase of the flow is aimed at verifying performance and power, which may differ from the initial analytical estimates. In the current setup we use a cycle-accurate simulation for CMP interconnect, supplied with probabilistic automata models for cores and memories [55].

The following sections will focus on algorithms for the physical planning of hierarchical CMPs. Their objective is to accurately estimate the chip area and wire length, subject to the physical constraints. The methods proposed in this work are applied at the second phase of the described exploration flow.

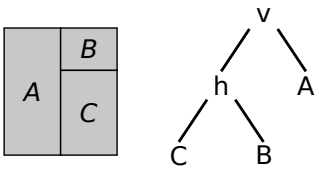


Figure 3.7: Example of a slicing floorplan and associated slicing tree.

3.4 Floorplanning methodology

This section presents the first step of physical planning: floorplanning.

3.4.1 Floorplan representations

Floorplanning is the task of defining tentative locations for the blocks of system under certain geometric constraints. The blocks represent pre-designed CMP components such as cores, memories and routers. The blocks can either have a fixed size or accept a set of different *aspect ratios*. The traditional floorplanning problem only considers the minimization of the total area occupied by the components. More advanced floorplanning strategies can also consider the minimization of other metrics such as the estimated wire length.

Because of the complexity of the problem, it is essential to select efficient data structures to represent floorplans. In this work, we use Simulated Annealing for the exploration of slicing floorplans similarly as proposed in [141], where the cost function is defined as a linear combination of area and wire length approximated with half-perimeter wire length. In addition, the cost function is extended with other components that aim at generating floorplans with some properties and constraints for tiled hierarchical CMPs.

Slicing trees

Slicing trees [141] is one of the most popular floorplan representations. It can represent only a family of floorplans called *slicing floorplans*. This subset of all possible floorplans contains only floorplans that can be represented entirely by a series of horizontal or vertical cuts. A slicing tree is just a tree representation of such series of cuts (see Fig. 3.7).

It has been proven that slicing trees, when combined with a compaction post-process, can represent all possible maximally compact layouts of any given library of components [99]. Hence the use of the slicing floorplans does not limit the search space. In this work, compaction is not applied and the generation of area-optimal floorplans is not guaranteed. However, the difference is expected to be acceptable, specially in the presence of soft blocks [147].

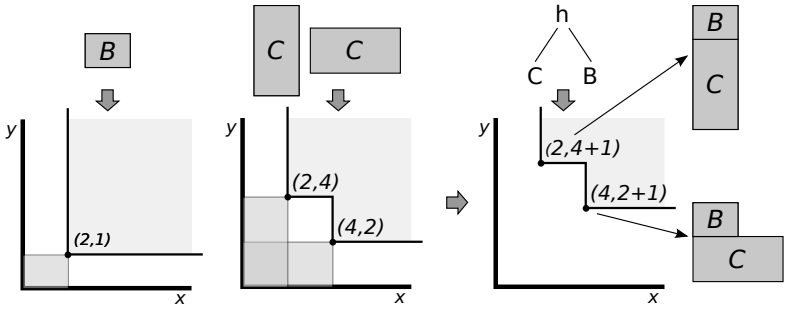


Figure 3.8: Example of vertical composition of the bounding curves for two components B and C.

Bounding curves

To represent the possible shapes of the individual components, *bounding curves* are used. A point (x, y) belongs to the bounding curve of a component if x and y are a valid width and height for that component (Fig. 3.8). There are efficient vertical and horizontal composition operations on bounding curves [116] to calculate all the valid aspect ratios of such compositions.

3.4.2 Search strategy

Algorithm 1 Floorplanning algorithm (Simulated Annealing)

```
FP ← "Initial slicing floorplan"
T ← "Initial temperature"
while improvements in the last k iterations do
    for p iterations do
        select FP_new randomly from neighbors of FP
        gain ← COST(FP) − COST(FP_new)
        if RANDOMACCEPT(T, gain) then FP ← FP_new
    T ← T · α
return FP
```

Even reducing the search to slicing floorplans, the space of solutions is high enough that the use of metaheuristics is unavoidable [144]. The floorplanning process described in this work uses an extension of the Wong-Liu algorithm, except for changes to the cost function that will be described in this section and Section 3.4.3.

The Wong-Liu algorithm [141] is a customization of Simulated Annealing [92] for the search of slicing trees. It defines a neighborhood function

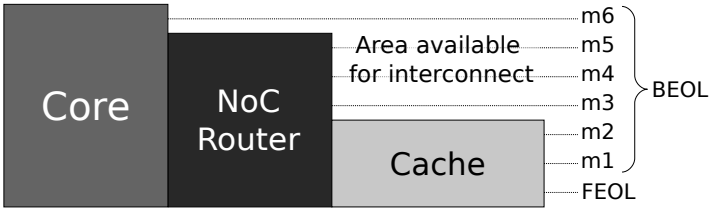


Figure 3.9: Multi-layered CMOS technology: *FEOL* includes the front-end layers (polysilicon and diffusion), *m1-m6* represents the available metal layers.

consisting of three movements that operate on top of Polish expressions, which are a string representation of slicing trees.

An overview of the search procedure is presented in Algorithm 1. One important ingredient of any Simulated Algorithm is the cost function. To allow trade-offs between area and other factors to be explored, we introduce weights to the different factors of the cost function:

$$\text{COST}(\text{FP}) = \alpha Area(\text{FP}) + \beta WL(\text{FP}) + \gamma WL_{Eq}(\text{FP}) + P(\text{FP})$$

In this expression, FP is the floorplan being evaluated, *Area* is defined as the effective area of the floorplan, *WL* is the sum of the wire length estimation for each net, and WL_{Eq} is the sum of the squares of the estimated wire lengths for nets in the ring interconnect, if any. The goal of WL_{Eq} is to penalize floorplans where equidistantly-spaced nets have excessively diverging lengths. The last term, P(FP), aggregates all possible penalties. We will explain each of these factors in more detail in section Section 3.4.3.

The α, β and γ parameters are weights that a designer can use to guide the search towards floorplans with smaller area or towards floorplans with smaller wire lengths. An example of this trade-off will be seen in Section 3.6.

3.4.3 CMP-aware floorplanning

In Section 3.1 we mentioned some of the requirements for the physical planning of tiled hierarchical CMPs. In this section we address them in more detail.

Over-the-cell routing

Current CMOS-VLSI design is multilayered. Individual devices such as transistors are patterned on the bottom layer, which in modern fabrication is composed of *polycrystalline silicon* (*polysilicon*) [139]. This layer is often called the *front*

end of line (FEOL). Successive layers are applied that can be used to make connections between the different transistors and external connections, collectively called the *back end of line* (BEOL). An example can be seen in Fig. 3.9.

The BEOL layers are used for the required wiring inside the various CMP components. However, different component types have different requirements of routing resources. On-chip memory often uses less layers than more complex circuits such as cores. If the input data models this, these free layers can be used for the wires between the different CMP components.

Because of the prevalence of cache memories in CMP tiles, we can assume that every configuration can be routed using the available metal layers on top of the components without requiring any extra whitespace. During floorplanning, and as part of the wire length estimation that will be described later in this section, unroutable configurations are discarded.

Abutability

Because only a single tile of a chip is floorplanned, some nets that connect different clusters will have floating terminals that must be placed on one of the boundaries of the tile. However, the placement of this terminal must lie adjacent to the placement of a corresponding terminal on the next cluster. Thus, a special symmetry constraint is created between pairs of nets. All the global interconnectnets have this property.

Wire length constraints

Due to performance reasons, certain critical nets must have a wire length constraint. In case these constraints are violated the floorplan is rejected. This maximum length will depend on the desired interconnect operating frequency, wire sizing and other parameters [80].

Equidistantly-spaced nets

For most interconnects, the communication delay is determined by the maximum length of a set of links. For example, in a ring, the cycle period must be long enough to allow packets to propagate across the longest of the ring hops. In these cases, it is desirable not to strictly minimize the total wire length, but to balance the individual lengths of the respective links. For this reason, nets that must satisfy this requirements are evaluated differently in the cost function (Section 3.4.2), minimizing the sum of the squares of the lengths instead:

$$WL_{Eq}(\mathrm{FP}) = \sum_{\forall \mathrm{net} \in \mathrm{Ring}} WL(\mathrm{net})^2$$

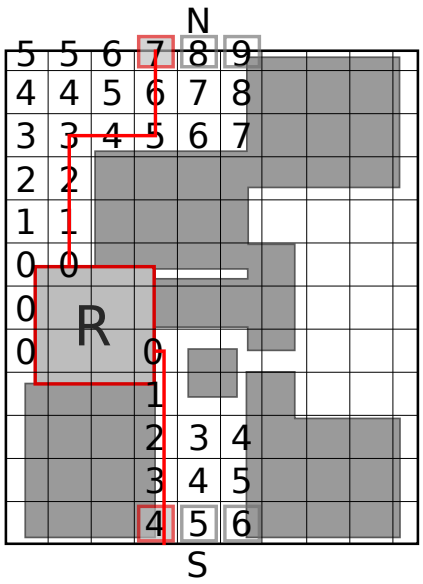


Figure 3.10: Maze routing a pair of nets with abutability constraint (floorplan from Fig. 3.4, blockages marked in black).

Wire length estimation

A good wire length estimator is important for the evaluation of the cost function. Wire length estimations are used in the $WL(\text{FP})$ and $WL_{Eq}(\text{FP})$ terms of the cost function. Additionally, it is used to check satisfiability of some of the constraints, such as abutability and wire length limits.

In over-the-cell routing, the only space considered for routing is the free space over the components that have the top metal layers available. Since cores and routers typically implement a complex internal wiring and thus utilize the highest number of layers, memories are the only components in the entire design that leave some metal layers unused. In fact, the relative area of memories in a tile is defined by the configuration, but it usually ranges between 50%-60% for the best configurations as seen in our tests.

Thus, the lowest metal layers will typically have no space for routing, while the upper layers will have up to 60% of space available, thereby making over-the-cell routing possible. An example can be seen in Fig. 3.10, which represents a middle metal layer from the floorplan in Fig. 3.4, with the area occupied by components marked in a dark color.

The work upon this floorplanning algorithm has been based on, [141], proposes the use of the half-perimeter wire length as an estimator. In this work, we propose the use of Lee's algorithm [119], often known as Maze routing. The reduce the complexity of the algorithm, three relaxations are applied:

1. Routing full links, not individual wires

2. Routing each net independently, so that no collisions are considered

3. Routing is performed on one metal layer only

Thus, routes might be generated that may be found unfeasible during wire planning. However, for the case of nets with two terminals, we can guarantee that a route found using this method is a valid lower bound. Thus, this information can be used to verify wire length and routability constraints. Because of simplification (1), the size of the routing grid is determined by the minimum link width.

The use of Lee's algorithm also enables checking for violations of the abutability requirement. When planning pairs of nets with such requirement, the algorithm will only accept a path if a matching path has been found on the opposite side for the paired net. The algorithm also will not stop at the first path, but rather collect all paths and select the one where the route is shortest to both opposing extremes of the tile. In Fig. 3.10, this algorithm is applied to estimate the length of the two vertical mesh links (from the Router to the north side and from the Router to the south). The shortest route for the north net is discarded because at the opposing side of the tile (same column, last row) there has been no path found for the south net.

A more accurate estimation of routability is performed during wire planning (Section 3.5) to discard those floorplans that are unroutable when considering all signals simultaneously.

3.5 Wire planning

In order to fully realize the floorplan estimated in the previous section, we need to establish a wire planning that connects all the required nets between the components and that allows the tiling of the cells. This wire planning must use over-the-cell routing and minimize its wire lengths, while balancing the nets.

This problem corresponds to a routing problem and we solve it in two steps. In the first step, we formulate the routing problem as a Boolean satisfiability problem for which we obtain a feasible solution with a SAT solver. Then, in the second step, we iteratively reduce the wire length of several nets by converting the satisfiability problem to an integer linear programming problem that we solve with an ILP solver. In the following, we describe their essential elements.

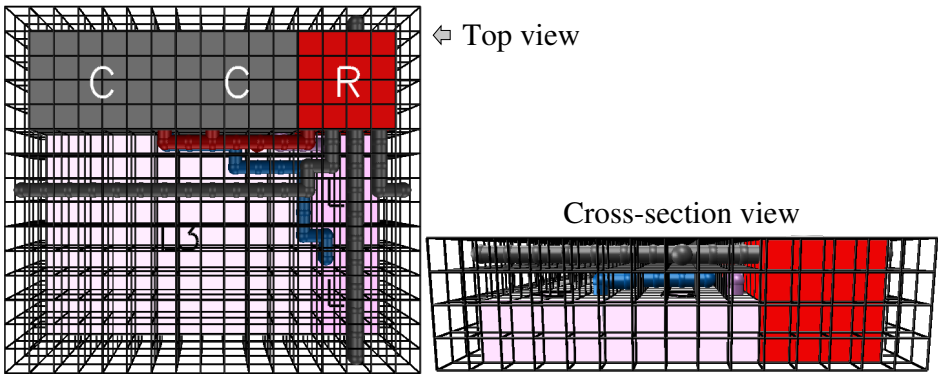


Figure 3.11: Grid structure used for wire planning.

3.5.1 Problem formulation

We formulate the routing problem as a Boolean satisfiability problem in the lines of [78], which we extend with some insights that are needed in the context of CMPs.

At this early design stage, wire planning is only performed on the wide communication links of the system, neglecting local control wires. These links can have more than 10^3 wires, e.g. full cache lines with data, address and control bits. The routes are calculated globally for the complete links and not for the individual wires that compose each link.

The routing region is represented by a uniformly-sized coarse grid (Fig. 3.11). The grid unit is determined by the minimum width of a link, $w = n \cdot p$, where n is the number of wires of the narrowest link and p is the *wire pitch*, i.e., the smallest distance between wires. Routing is performed on a 3D grid with blockages according to the metal layers occupied by the CMP components, as illustrated in Fig. 3.9.

The main variables of the SAT problem correspond to the presence (or absence) of a wire segment between two adjacent nodes of this 3D grid. Another set of variables encodes the assignment of wire segments to specific nets. The SAT problem includes several types of constraints:

- Consistency constraints enforce the expected behavior of the variables we have introduced, e.g., if an edge is assigned to a net, then the edge must be occupied by a wire.

- Routability constraints define a legal routing between the components. Basically, these constraints establish that a set of wire segments guarantee the connectivity of all pins of a net. The formulation is similar to the one presented in [78] but extended to handle floating terminals. Our

solution is based on the idea that routing must be performed among regions of points that define the endpoints of the nets. These regions are characterized by a set of (not necessarily adjacent nor disjoint) points that may describe the location of a component or the set of all possible locations for a pin. The correctness of our routability constraints is based on Euler's graph theory.

- Abutability constraints ensure the symmetry between the wires that are used to interconnect tiled cells. These constraints assert that if a wire in the North boundary provides a signal for a net that interconnects adjacent cells, another wire for the same cell must be placed in the same position in the South boundary. Similar relations must also occur in the other direction and for East/West boundaries.

- Optionally, constraints for design rules can be requested in order to fulfill fabric requirements or to reduce running time. One of the typical design rules is to assign one direction to each metal layer.

Solving the previous satisfiability problem provides a first feasible solution for the wire planning problem (or shows the absence of such a solution!).

3.5.2 Reduction of wire length

Once we have a feasible solution for the wire planning problem, we improve it by reducing its wire length while maintaining its feasibility. Our strategy is iterative, where each iteration consists in ripping out a small set of nets from the feasible solution and reroute them, subject to the previously specified constraints and minimizing the total wire length.

To do so, we convert our Boolean satisfiability problem into an integer linear problem: Boolean variables are transformed in 0/1 variables, Boolean constraints are easily converted to linear inequalities and, the linear function that counts the amount of wire is used as the objective function of the ILP.

Since the above process is applied for a small set of nets at each iteration, the resulting problem is tractable and can be solved with efficient solvers in a moderate amount of time. Note that solving the original problem with all the nets and seeking for the absolute minimum is too slow for the sizes of the problems we are faced to.

The currently implemented iterative process proceeds by just ripping out and rerouting one net at a time, with the exception of the set of nets that interconnect tiled cells, which are ripped out and rerouted in one step. This process is repeated while reductions in the wire length are obtained, favoring the reduction of long nets before the reduction of shorter nets.

Parameter	Value		
Maximum chip area	350 mm^2		
Maximum chip power	350 W		
Interconnect frequency	1.6 GHz		
Global interconnect types	Mesh		
Global mesh dimensions	2×2 to 16×16		
Local interconnect types	Bus, Ring		
Local interconnect sizes	Limited by chip area only		
Memory density	1 mm^2/MB		
Cache latency (per size)	$5.0 \cdot CacheSize^{0.5}$ cycles		
Off-chip memory latency	100 cycles		
Interconnect link width	10 μm (10^3 wires×10 nm)		
Available metal layers	m1, m2, m3, m4		
Used by cores	All		
Used by routers	All		
Used by cache memories	m1, m2		
Core types	*C1*	*C2*	*C3*
Core performance (IPC)	1.75	2	2.5
Core area	1 mm^2	1.25 mm^2	2 mm^2
L1 size	64, 96, 128 KB per core		
L2 size	64 KB to 1 MB per core		
L3 size	Up to 100 MB per chip		

Table 3.1: Parameters for system-level exploration.

3.6 Results

In this section we demonstrate the impact of using physical planning during system-level exploration, and also show the need of CMP-specific constraints during physical planning for a proper evaluation of architectural configurations.

3.6.1 Exploration setup

All of the experiments from this section use configurations that were obtained using automated system-level exploration [113]. The parameters of this exploration are described in Table 3.1. We limit the search to tiled hierarchical CMPs using a mesh as the global interconnect, with the second level interconnect being a bus or a ring (bi-directional or uni-directional). The number of tiles, the number of cores and the distribution of cores among the tiles are exploration variables. We assume that three different models of cores are available (C1, C2 and C3), with different performance and area characteristics obtained by scaling publicly available data of the Intel Core 2 Duo E6400 processor [48]. We also assume that, while cores and interconnect routers occupy all metal layers,

cache memories only use two of them. Therefore, routing can be performed over the cache memories. The operating frequency of the interconnect has been used to define the constraints on the maximum wire length for the links.

The wire planning models were solved using PicoSAT [23], and Gurobi [72] was used to optimize the wire length as per Section 3.5.

To characterize the memory accesses, a model extracted from the SPEC2006 *soplex* benchmark is used. The exploration generates 200 configurations in around 20 minutes. Each configuration is described by its architectural parameters. For example, the best configuration from this exploration has 25 clusters connected with a 5×5 mesh. Each cluster has a bus as local interconnect, two C2 cores and two C3 cores, along with 1 MB of L2 cache per core. The CMP has a total of 50 MB L3 cache distributed across the 25 clusters. It has an estimated throughput of 107.77 IPC.

3.6.2 Impact of physical planning

In order to prove how the use of physical planning can significantly alter the results of system-level exploration, we applied our physical planning tool to the 200 configurations found by the exploration. This floorplanning process, if run sequentially, takes 5 hours (an average of 90 seconds per configuration). However, on a machine with multiple cores each of the 200 configurations can be run separately.

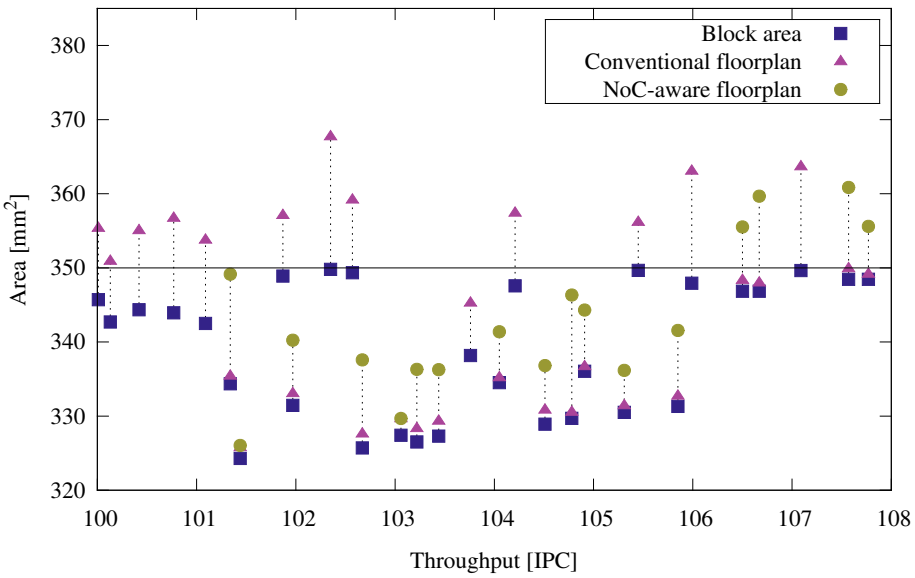


Figure 3.12: Area as measured by different floorplanning strategies.

The results are shown in Fig. 3.12. For each configuration, *block area* indicates the sum of the areas from all components. The exploration tool, before physical planning, uses this value as estimator for the expected chip area in order to satisfy the maximum area constraint. In this example, no configuration has a block area larger than 350 mm^2. *Conventional floorplan* shows a minimal area floorplan obtained without using any of the constrains described in this work (abutability, link length optimization, etc.). On the other hand, *NoC-aware floorplan* depicts the floorplan with minimal area that satisfies these constraints. A dashed line connects the block area data point with the minimal NoC-aware floorplan area for the same configuration.

Despite the fact that all configurations have a block area lower than the limit, a large number exceeds the area limit once physical planning is taken into account. As an example, the best configuration found by the exploration (rightmost in Fig. 3.12) has a block area of 348.45 mm^2, which is below the area constraint. A conventional, minimal area floorplan exists with an area of 349.17 mm^2, also below the constraint. However, using the tool presented in this work, we find that the smallest floorplan satisfying all floorplanning constraints has an area of 355.59 mm^2. This violates the area constraint and, therefore, is not actually a valid configuration.

The first viable configuration with area below the limit has a significantly lower performance at 105.85 IPC. Out of the 200 configurations selected during the exploration, 39% of configurations had no floorplan satisfying all the constraints. Even for the configurations for which such a floorplan was found, only 23% satisfy the 350 mm^2 area limit. Configurations using rings as local interconnect, despite their excellent performance characteristics, have much stricter physical constraints and thus often violate design constraints. Without physical planning, those configurations would have been tagged as “promising” and would have been analyzed with more accurate simulation tools.

3.6.3 Physical planning search space

A single CMP configuration can have a large number of alternative floorplans. Nevertheless, it is desirable to select one or few candidate floorplans. At the same time, we are considering two metrics by which feasible floorplans can be evaluated: area and wire length. Thus, there is a trade-off.

In Section 3.1 we showed two candidate floorplans where one had much shorter total wire length at the cost of a 15% increase in the chip area. Since this trade-off might be inconvenient for some designs, the weights in the cost function (described in Section 3.4) can be modified to guide the search towards floorplans with better area or towards shorter wire length.

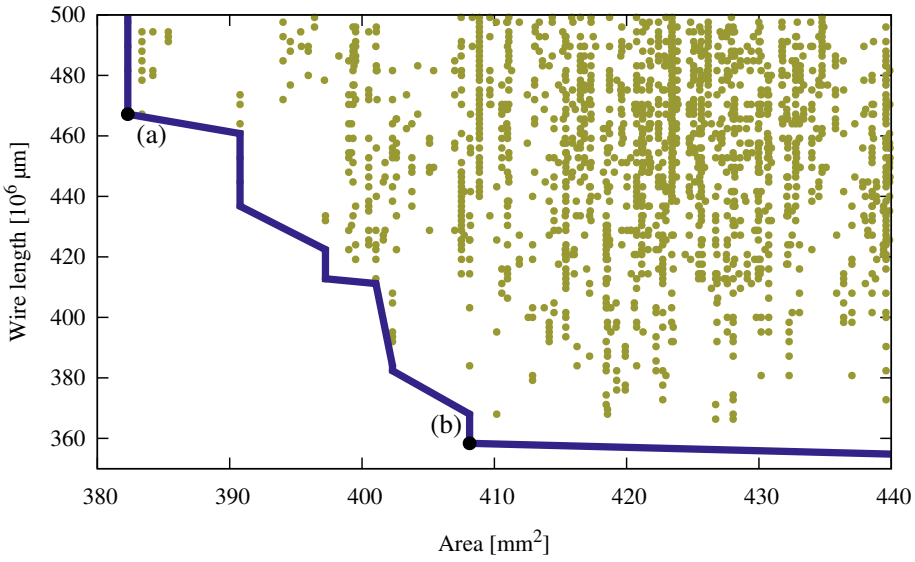


Figure 3.13: Example of physical planning search space for a single CMP configuration.

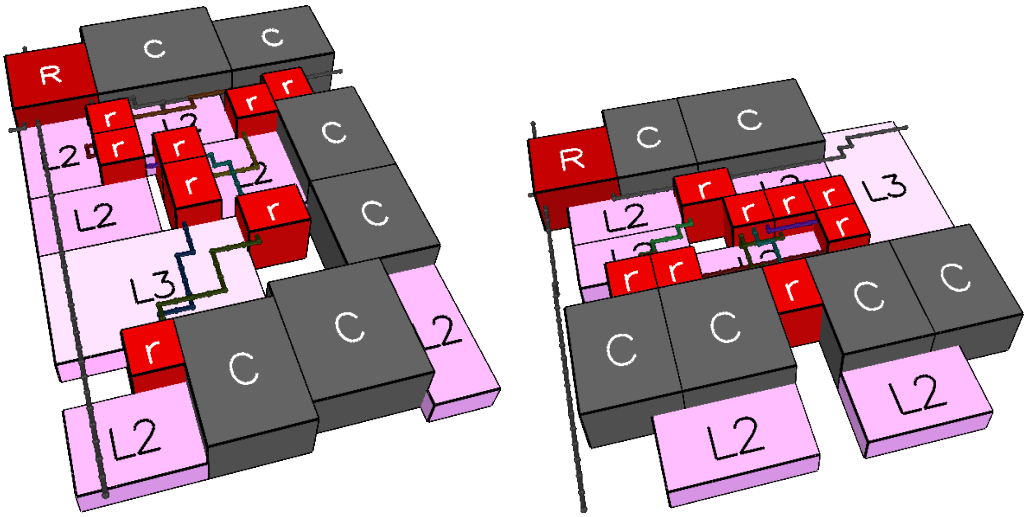


Figure 3.14: Two design points from the exploration space in Fig. 3.13.

Figure 3.13 is an example of the available floorplans for a given CMP configuration. In the chart, each point represents a valid floorplan and its position depends on the area and wire length for that floorplan. The 10 Pareto-dominating solutions are represented as a solid line (Pareto frontier). By changing the weights in the cost function, a designer can decide which of these solutions are most desirable. We expect this small set to be further reduced by area or other types of physical constraints.

To illustrate, we selected two representative floorplans from the Pareto frontier that we show in Fig. 3.14. These are, respectively, the floorplan with the minimal area (but satisfying all constraints) and the overall best floorplan assuming we give the same weights to both area and wire length minimization.

3.6.4 Publications

As part of this research topic we have published the following articles:

- J. de San Pedro, N. Nikitin, J. Cortadella, and J. Petit, *Physical Planning for the Architectural Exploration of Large-Scale Chip Multiprocessors*, in Proceedings of the 2013 IEEE/ACM Seventh International Symposium on Networks-on-Chip, Tempe, Arizona, USA, 2013, pp. 1–2.
- J. Cortadella, J. de San Pedro, N. Nikitin, and J. Petit, *Physical-aware system-level design for tiled hierarchical chip multiprocessors*, in Proceedings of the 2013 ACM International Symposium on Physical Design, New York, NY, USA, 2013, pp. 3–10.

The first article centers on the integration of physical planning into architectural exploration, while the second article centers on the specifics of floorplanning for CMPs.

3.7 Conclusions

The impact of physical layout is often neglected when exploring the architectural parameters of a CMP. This chapter has presented a framework in which physical planning has been integrated with architectural exploration to generate physically-viable high-performance CMPs. The presence of physical constraints has been shown to have an important impact in deciding the parameters for the design of CMPs.

This is a first step towards future design frameworks in which accurate power-performance models and advanced technologies with hierarchical on-chip interconnects can be incorporated. Future work in this direction has to

address the issues of physical planning for alternative interconnect topologies, such as crossbars. Additionally, the accuracy of performance prediction can be improved by using physical information while estimating the system throughput. Another important task is to look for alternative approaches to physical planning in order to speed-up the phase of system-level design for CMPs.

Chapter 4

Regularity-constrained floorplanning

The complexity of the VLSI physical design flow grows dramatically as the level of integration increases. An effective way to manage this increasing complexity is through the use of regular designs which contain more reusable parts. In this chapter we introduce *HiReg*, a new floorplanning algorithm that generates regular floorplans.

In Chapter 3, the benefits of regular designs were already demonstrated with tiled Chip multiprocessors (CMPs). In a tiled CMP, there is a single design for a tile that is designed once but replicated many times, thereby reducing the design effort. HiReg goes further and automatically extracts repeating patterns in a design by using graph mining techniques, without any information from the designer. As will be seen, this also allows HiReg to extract multiple hierarchical levels of repeating patterns, instead of being limited to a single tile pattern.

Regularity is exploited by reusing the same floorplan for multiple instances of a pattern, as long as neither area, wire length or existing hierarchy constraints are violated or compromised. Experiments will show the scalability of the method for many-core CMPs and competitive results in area and wire length with traditional floorplanners.

This chapter starts by giving a brief motivation of the problem in Section 4.1. The current state of the art is reviewed in Section 4.2. Section 4.3 uses an example to demonstrate the challenges solved by the presented approach. Section 4.4 describes the inner workings of HiReg. In Section 4.5, HiReg is compared with other floorplanning tools to evaluate the benefits of regularity. Conclusions are discussed in Section 4.6.

4.1 Motivation

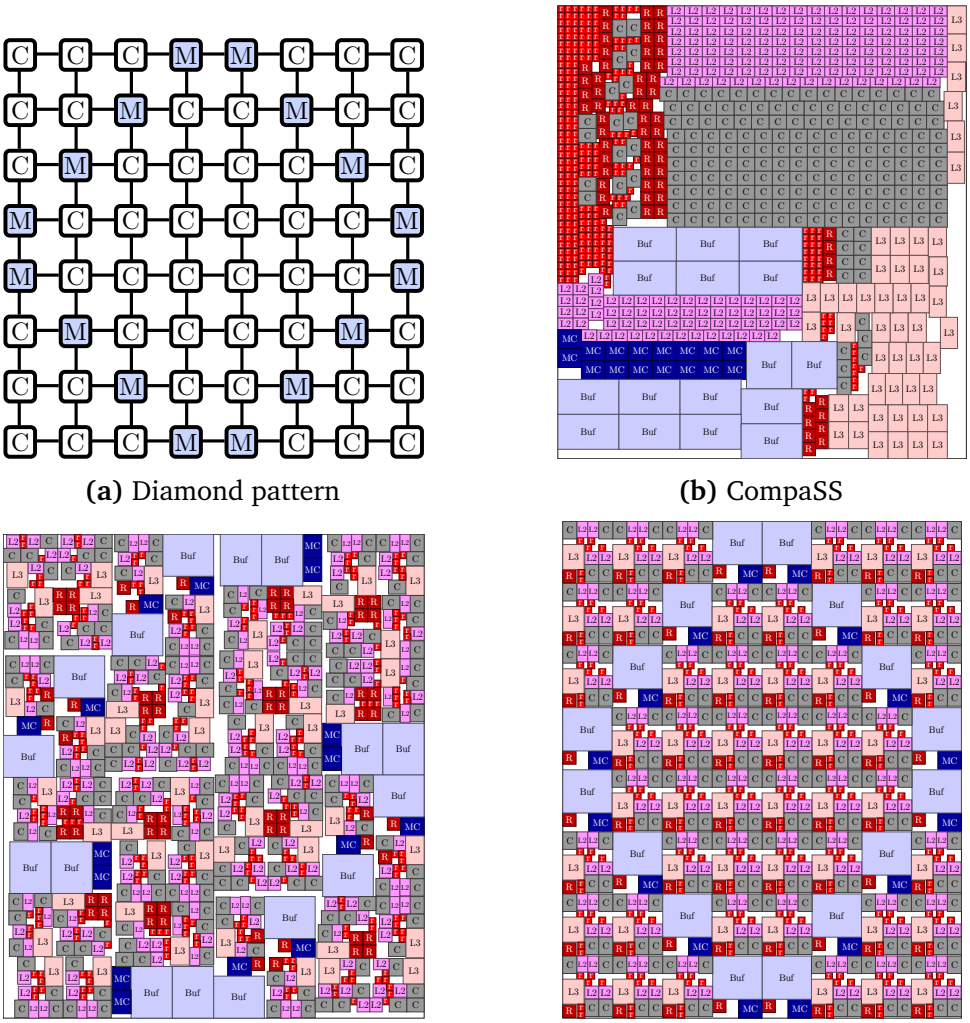
The computational complexity of the floorplanning problem highly depends on the number of components of the system. For large systems, a flat view makes the floorplanning problem intractable. For this reason, hierarchical methods [135, 143] has been proposed and successfully used to reduce this complexity. Hierarchical methods divide the floorplanning problem into multiple subproblems that may be either fully or partially independent from each other, thereby enhancing scalability.

An important metric often disregarded during floorplanning is *regularity*, known to lead to efficient and economical designs [128]. Large-scale systems have significant amounts of regular patterns than can be exploited (on-chip memories, many-core CMPs, etc.). The design cost of such systems can be brought down by reducing the number of distinct subcircuits to be designed, and then replicating the pre-designed subcircuits as many times as possible. To allow this reduction, a regular floorplan uses the exact same layout for all replications of a subcircuit. To reduce complexity of timing closure, it is also desirable for all of the adjacent components to be placed in similar relative positions, so that the interconnect geometries are regular and timing analysis is similar.

In many-core CMPs, tiled layouts [16] are often used to exploit regularity, as in the previous chapter. The design is split into homogeneous tiles that are only floorplanned once and then replicated. However, with industry moving towards heterogeneous CMPs [97], it is no longer possible to assume that most CMPs will have only a single type of tile. Integrated graphics, accelerators and I/O blocks are some of the special types of co-processors that introduce heterogeneity.

On the other hand, enforcing regularity in a design may compromise other floorplan metrics such as area or wiring. Existing designs are often hierarchical in nature. CMPs with hierarchical topologies are a good example. Preserving the pre-defined hierarchy may result in a better wiring quality (e.g. by reducing the number of wires that cross between different subcircuits). Very often, hierarchy is manually enforced by designers to split the design and assign components to different design teams. Thus, breaking the existing hierarchy could be counterproductive to the goal of simplifying design. Another example is the concept of *choppability* [122]. In a choppable floorplan, large functional blocks can be chopped away, reducing the total die size and varying performance/power metrics in order to construct multiple versions of a product from the same basic design.

In this chapter we introduce a new floorplanning algorithm, *HiReg*, that considers area, wiring, regularity and hierarchy as floorplanning objectives. Very often these objectives are conflicting, e.g., reducing final area compromises regularity and vice versa. To deal with this issue, HiReg uses a new method



(a) Diamond pattern

(b) CompaSS

(c) DeFer

(d) HiReg

Figure 4.1: Example CMP floorplans generated using different floorplanning strategies.

that can trade-off hierarchy and regularity constraints. Both hierarchy and regularity are automatically discovered from the block netlist.

4.2 Related work

There has been little work in the area of regular floorplans. Regularity is more common in the area of physical design for analog circuits, where it is often a strict requirement due to the peculiarities of analog design [15]. However,

	Hierarchy	**Regularity**
[38]	No	Arrays only
REGULAY [145]	Yes	Tiles only
DeFer [143]	Yes	No
CompaSS [37]	By similarity	No
[135]	Yes	No
ArchFP [64]	Manual	Manual
HiReg	Yes	Yes

Table 4.1: Comparison of related work.

most of the techniques in analog design involve symmetry properties that are not relevant for maximizing design reusability.

[38] acknowledges the importance of regular designs in CMPs and describes a simulated annealing-based floorplanner that organizes groups of similar blocks in regular arrays. However, the technique does not fully exploit regularity since adjacent components may not be placed in aligned locations that enable regular interconnection geometries. The blocks that are to be placed in regular groups must also be manually selected by the designer.

In System-on-Chip design, *REGULAY* [145] also mentions the importance of preserving regularity and hierarchy. *REGULAY* discovers the optimal mapping of heterogeneous tiles into a regular grid arrangement, and does not consider the floorplanning of the individual tiles themselves.

On the other hand, the advantages of using hierarchy during floorplanning are not new [49]. Nonetheless, most existing work uses hierarchy only to improve the scalability of the floorplanning problem, allowing efficient generation of floorplans with large numbers of components, and differ in the methodologies used to discover hierarchy.

DeFer [143] uses graph bipartitioning to generate a binary tree of balanced netlist partitions, a method similar to the one proposed in this work for hierarchy discovery. This hierarchy tree is then used to generate a slicing tree, reducing the number of floorplans that need to be explored during the search. *CompaSS* [37] automatically clusters blocks with similar or identical shapes, and then creates grid floorplans for them. However, *CompaSS* ignores connectivity information. [135] applies a recursive slice-and-partition method derived from cell placement strategies.

All three examples use slicing floorplans and bounding curves that will also be used in this work to efficiently represent floorplans. Slicing floorplans are not able to represent the entire set of optimal floorplans. However, this difference is minimal given a large number of soft blocks [147], as it often occurs in CMPs.

Component	Area	Aspect ratio
Core (*C*)	1.38 mm^2	0.8 or 1.25
L2 cache	1 mm^2	0.5 ÷ 2
L3 cache	3 mm^2	0.5 ÷ 2
Ring router (*r*)	0.27 mm^2	1
Mesh router (*R*)	0.99 mm^2	1
Memory controller (*MC*)	2.5 mm^2	0.8 or 1.25
Buffer (*Buf*)	12 mm^2	0.5 ÷ 2

Table 4.2: Physical information for Fig. 4.1.

	Whitespace	**HPWL** (m)
CompaSS	6.3%	6801
DeFer	8.2%	630
HiReg	12.6%	516

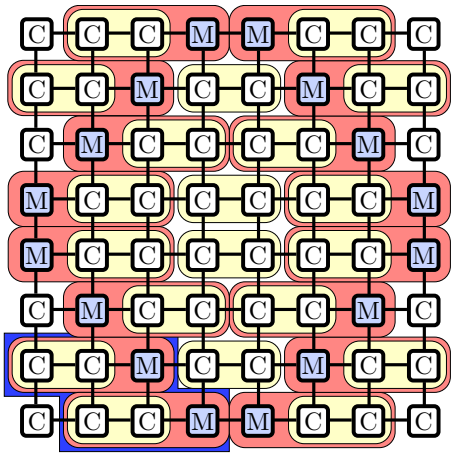
Table 4.3: Floorplanning results for Fig. 4.1.

ArchFP [64] describes a different strategy that can produce floorplans that are both regular and hierarchical. However, *ArchFP* assumes that a designer will construct, previously to the floorplanning process, a manual hierarchy of the CMP components and will choose a floorplanning approach for each group of components. Our work extracts hierarchy and regularity in an automated way, without any previous knowledge of the topology of the input netlist. Table 4.1 summarizes the differences between these strategies.

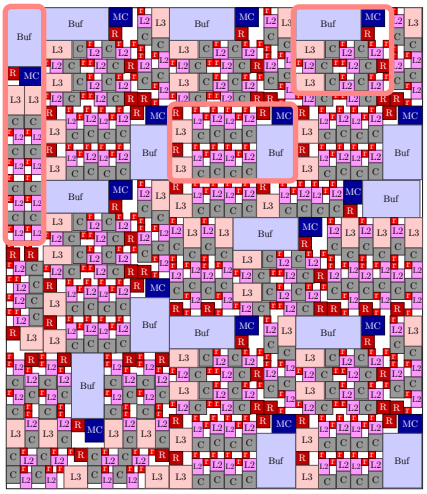
4.3 Exploring regularity and hierarchy

This section will use an example to illustrate the trade-offs between regularity and hierarchy. Figure 4.1 shows the result of floorplanning the same netlist using HiReg and two other hierarchical floorplanners. Table 4.3 contains whitespace and wire length results. This netlist represents a hierarchical tiled CMP, with 192 cores. It contains 64 tiles, with 48 *processing* tiles containing 4 cores each, and 16 *memory controller* tiles. This CMP uses a hierarchical Network-on-Chip topology. An 8×8 mesh interconnects all tiles. Inside each tile, a ring provides connectivity.

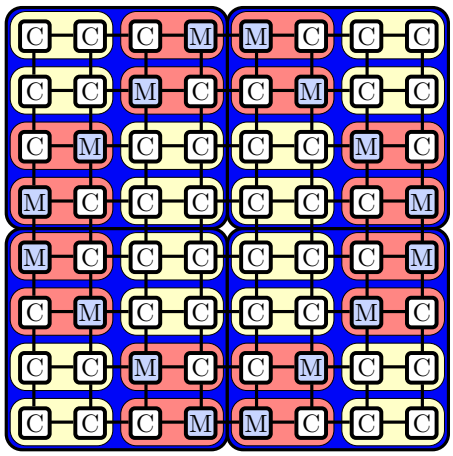
The mapping of processing and memory controller tiles has been selected to match the *diamond* pattern (Fig. 4.1a, [9]) which maximizes off-chip memory performance. Thus, this configuration is representative of a potential many-core CMP design. For the sake of easy visualization, both types of tiles have similar



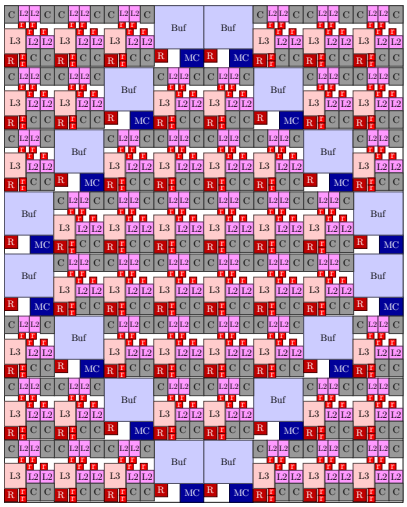
(a) Regularity without hierarchy



(b) Floorplan generated from (a)



(c) Regularity preserving hierarchy



(d) Floorplan generated from (c)

Figure 4.2: Example floorplans with and without hierarchy constraints.

area requirements in this example. In general, each tile may have a different area constraint.

In addition to cores (C), processing tiles contain L2 caches private to each core, a tile-shared L3 cache block, ring routers for intra-tile communication (r) and mesh routers for inter-tile communication (R). The memory controller tiles each contain a buffer (Buf), mesh router, and a memory controller itself (MC). Physical information is described in Table 4.2. Cores come in several hard aspect ratios, but we assume memories to be flexible within a limited range. We also assume every net represents a link with 1024 wires.

In Fig. 4.1b, CompaSS groups blocks by similarity and creates arrays in order to improve packing quality, thus resulting in floorplans that have some regularity. However, connectivity information is not considered, resulting in floorplans with good area metrics but poor wire length.

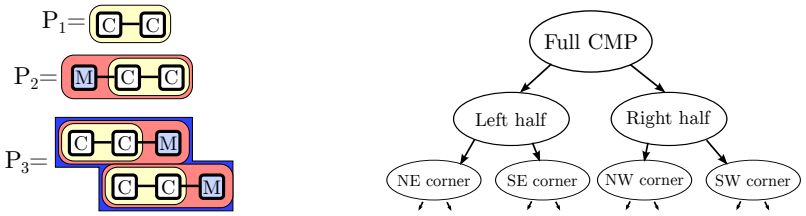
DeFer minimizes area and wire length, and uses hierarchy to floorplan efficiently. In Fig. 4.1c, we disabled compaction in order to easily visualize the effects of hierarchy, but it was enabled for obtaining the results in Table 4.3. Because of hierarchy, the floorplan is divided in 4 quadrants, with each quadrant also divided in 4 quadrants, and so on. However, small differences in the floorplans used for every quadrant prevent reusing the same design for all sub-quadrants. The construction of hierarchy from connectivity information results in a 2% area increase compared to CompaSS, but generates a significantly reduced wire length.

HiReg, on the other hand, constructs a floorplan that exploits the regularity and hierarchy inherent to a tiled CMP design. It is able to extract additional regularity by grouping cores inside tiles in blocks of 2. This tiled structure is discovered despite HiReg not having any previous knowledge of the interconnect topology. The use of hierarchy and regularity causes an additional 4% area increase over DeFer results, but generates a 20% reduction in wire length. Because of regularity, the entire CMP can now be constructing by replicating the two types of tiles. At the same time, two cores in every processing tile can be constructed by replication, resulting in significant design time savings.

4.3.1 Discovering regularity

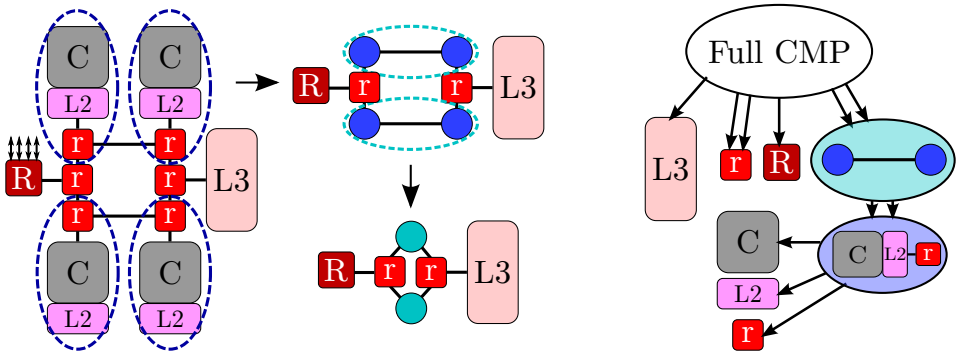
In order to create regular floorplans, HiReg automatically finds repeating patterns in the input netlist. We define a pattern as a subgraph from the netlist. We consider a pattern P to be repeated if there is at least one additional subgraph in the netlist isomorphic to P. We call all the repetitions of P the *instances* of P.

An example of the way HiReg extracts regularity is shown in Fig. 4.4a. The initial netlist contains 4 instances of the same pattern (composed of C, $L2$ and r each). After identifying this pattern, HiReg *compresses* the graph, replacing



(a) Patterns used in Fig. 4.2a **(b)** Discovered hierarchy tree

Figure 4.3: List of patterns and hierarchy tree.



(a) Regular extraction process **(b)** Generated DAG

Figure 4.4: Regular extraction process and example generated DAG.

every instance with a new vertex, representing the compressed instance. The process iterates until no additional patterns can be found.

Because of this iterative process, the result of regularity discovery is actually a directed acyclic graph (Fig. 4.4b). In this DAG, there is a leaf node (with no exit edges) for each component type in the netlist. Every other node represents a pattern. An edge between two patterns P_a, P_b indicates that pattern P_a contains an instance of P_b. The root pattern (with no entry edges) represents the entire original netlist.

HiReg uses this DAG to apply a divide-and-conquer strategy. Instead of floorplanning the entire netlist, the problem is split into floorplanning every pattern. To ensure regularity, HiReg enforces using the same or similar floorplans for all instances of a pattern, albeit this restriction may be relaxed if better area or wire length results are required.

4.3.2 Trading off regularity and hierarchy

An important contribution of this chapter is the importance of preserving existing hierarchy when discovering regularity.

Our initial approach completely disregarded hierarchy and centered on regularity as defined in Section 4.3.1. The regularity extraction process is primarily based on local decisions and lacks a global vision of the entire netlist. By centering on regularity only, the results may contradict existing design hierarchy, which can be counterproductive to the goal of reducing design time.

A visual example is shown in Fig. 4.2. This example shows a CMP design identical to the one in Fig. 4.1, containing processing (C) and memory controller (M) tiles. In a and b floorplanning is performed using discovered regularity only, without preserving hierarchy. c and d show the results of floorplanning using both regularity and hierarchy discovery.

The set of repeating patterns that have been extracted to construct a are shown in Fig. 4.3a. Because the regularity discovery process lacks global vision of the netlist, it discovers a set of patterns that break the natural tile hierarchy of the CMP. Despite patterns P_1 and P_2 being frequent patterns, using these to compress the netlist limits further extraction of regularity. The only remaining patterns left after compressing the netlist with P_1 and P_2 are combinations that do not respect the mesh topology, such as P_3. P_3 groups a non-rectangular set of tiles. In these situations, good area and wire length metrics cannot be obtained if the same layout must be strictly replicated for all instances of P_3. Thus, the regularity of the design is compromised, as shown in Fig. 4.2b.

In HiReg, existing netlist hierarchy is automatically discovered using recursive graph bisection (similar to [143], described in Section 4.4.1). The recursive graph bisection procedure divides the design into a recursive series of area-balanced partitions. To account for topologies with a non-power-of-two number of partitions, HiReg allows for up to 1-to-3 imbalance in the area of the automated bisections. Alternatively, the hierarchy information may be provided by the designer.

This hierarchy information is represented as a tree, such as the one in Fig. 4.3b, which guides the regularity discovery process. Specifically, no pattern instances can cross boundaries delimited by hierarchy. In Fig. 4.2c, no pattern instance was allowed to extend to more than one CMP quadrant, based on the hierarchy tree discovered by bisection (Fig. 4.3b) which separates the four CMP quadrants. This restriction reduces the number of P_1 instances, but eventually allows a larger number of more regular patterns to be found. In this way, maintaining hierarchy adds a global vision to the regularity extraction process.

4.4 Regular floorplanning algorithm

The algorithm can be divided in 5 stages, as seen in Fig. 4.5. The first 3 stages of the algorithm perform *hierarchy discovery* and *regularity discovery* based on

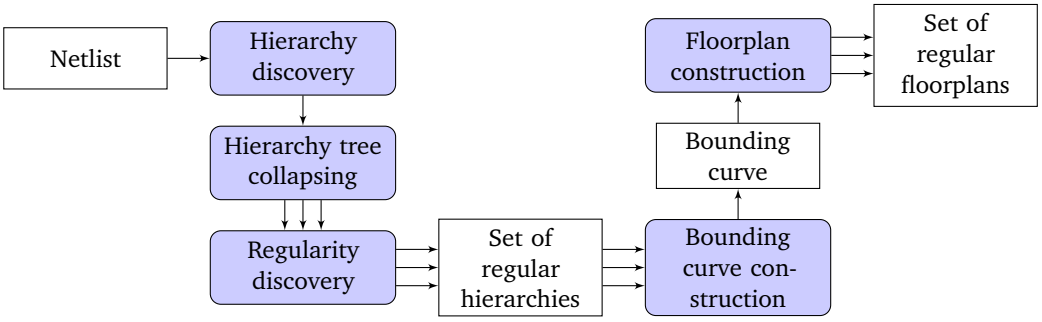


Figure 4.5: High-level flow of the algorithm.

the input netlist. During *hierarchy tree collapsing*, trade-offs between hierarchy and regularity are explored. Instead of generating a single regular hierarchy, between stages 3 and 4 we store a set of candidate regular hierarchies, delaying the selection on which hierarchy is most optimal until after all hierarchies have been evaluated.

The latter two stages perform actual floorplanning for all the candidate hierarchies. Stage 4 (*bounding curve construction*) enumerates all possible floorplans for each of the hierarchies, and stores the outlines efficiently as a single bounding curve. After this stage, the outlines for each possible floorplan are known and the designer can select a smaller subset based on physical metrics such as aspect ratio. Stage 5 (*floorplan construction*) constructs the selected floorplans.

Algorithm 2 contains a formal definition of this multiple stage flow, showing all 5 stages. The stages will be explained in detail during this section.

4.4.1 Hierarchy discovery

The first stage discovers the existing hierarchy in the input design. This is performed in order to ensure that an existing high-level topology in the design is preserved. Respecting existing design hierarchies is not only desirable from a reusability point of view, but also generates results with improved wire length when compared to results that create layouts which ignore the existing topology.

The method proposed in this section is based on hypergraph partitioning, an extension of the methodology proposed in [143].

We assume that the input netlist, represented as the hypergraph G, has a natural number of partitions. For example, a CMP with 8×8 tiles would have 64 natural partitions. A 63-way or 65-way partition would result into more interconnections between the different subcircuits than the natural 64-way partition. The goal of the algorithm is to discover these natural partitions.

Algorithm 2 General overview of the algorithm

function REGULARFLOORPLANNING(G)

1. Hierarchy discovery:

CandidateHierarchies $\leftarrow \emptyset$
HierarchyTree $\leftarrow$ HIERARCHYDISCOVERY(G)
for *threshold* in $\{0 \dots n\}$ **do**

2. Hierarchy tree collapsing:

CollapsedHierarchyTree $\leftarrow$
COLLAPSEHIERARCHY(*HierarchyTree*, *threshold*)

3. Regularity discovery:

RegularHierarchyDAG $\leftarrow$
REGULARITYDISCOVERY(*CollapsedHierarchyTree*)
append *RegularHierarchyDAG* to *CandidateHierarchies*
▷ CandidateHierarchies *contains candidate regular hierarchy DAGs*

4. Bounding curve construction:

$\Gamma \leftarrow$ empty bounding curve
for all *RegularHierarchyDAG* $\in$ *CandidateHierarchies* **do**
$\Gamma \leftarrow \Gamma \cup$
CONSTRUCTBOUNDINGCURVE(*RegularHierarchyDAG*)
▷ Γ *contains the bounding curve of all possible floorplans for all of the candidate hierarchies*
SelectedPoints $\leftarrow$ select desired outlines from Γ

5. Floorplan construction:

return CONSTRUCTFLOORPLANS(*SelectedPoints*)

The input netlist G is partitioned into two smaller circuits, minimizing the total number of interconnections between the two partitions, as long as both partitions have an area imbalance $\leq \frac{2}{3}$. Such imbalance margin allows handling circuits whose natural number of partitions is not a power of 2, by dividing the circuit into a partition with $\frac{2}{3}$ of the area and one with $\frac{1}{3}$. Between multiple bipartitions with the same number of interconnections, the most balanced partition is preferred.

The process is recursively applied to the two generated partitions, until all partitions contain a low enough number of blocks so that further bisectioning is not required (*MinSize*). This process is described in Algorithm 3.

During the process a binary tree is created where every leaf node is a subcircuit (with a number of components $<$ *MinSize*), and every other node is

a bipartition, the root node being a bipartition of the input netlist G. We call this tree the *hierarchy tree* (Fig. 4.3b).

Algorithm 3 Hierarchy discovery algorithm

```
function HierarchyDiscovery(G)
    ▷ G is the input netlist
    if |G| <MinSize then
        ▷ Trivial case if there are too few elements left
        return G
    G1, G2 ← bipartition of G minimizing number of
                edges between G1 and G2 with
                AreaImbalance(G1, G2) ≤ 2/3
    T1 ← HierarchyDiscovery(G1)
    T2 ← HierarchyDiscovery(G2)
    return CreateTree(T1, T2)

function AreaImbalance(G1, G2)
    return Max(Area(G1),Area(G2)) / (Area(G1)+Area(G2))
```

4.4.2 Hierarchy tree collapsing

The trees generated during hierarchy discovery are used to constrain the regularity discovery procedure and ensure that existing circuit hierarchy is preserved. However, strictly preserving all hierarchy would prevent the floorplanner from finding regularity, as discussed in Section 4.3. Thus, this stage generates hierarchies that have been *relaxed*, giving more flexibility to the posterior regularity discovery procedure.

It is often the case that only the high-level hierarchy is significant. For example, it is important to ensure that tile boundaries are honored in a CMP, but the contents of the tiles themselves often have a less well defined hierarchy, and a reduced connectivity impact if such hierarchy is not preserved. For this reason, it is preferable to relax hierarchy at the leaves of the hierarchy tree.

Algorithm 4 shows the details of the algorithm. *threshold* is an input parameter, specified as an absolute area value. For tree nodes where the total area is less than this threshold, the tree node is *collapsed*: all of its descendants are enumerated, combined into a single subcircuit, and the tree node is replaced by the new leaf subcircuit node. Thus, the resulting collapsed tree will have fewer nodes than the input hierarchy tree, with larger leaf nodes containing more components. See Fig. 4.6 for a visual example.

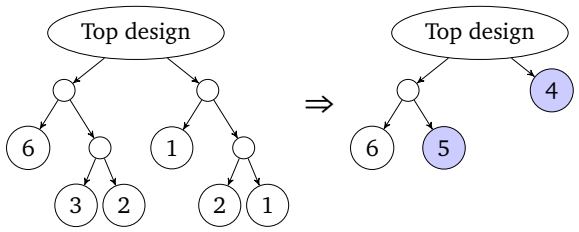


Figure 4.6: Tree collapsing with a threshold of 4 mm^2. Labels in leaf nodes indicate block area (in mm^2).

Multiple possible hierarchies are generated by this process by automatically testing for several values of the *threshold* parameter. This way, the trade-off between hierarchy and regularity is explored. The selection of the best threshold is thus deferred until floorplans are generated and area, wire length and additional metrics are available.

Algorithm 4 Tree collapsing

```
function CollapseTree(HierarchyTree, threshold)
    ▷ any nodes representing parts of the netlist with less area than threshold
      are collapsed
    if BlockArea(Tree) > threshold then
        ▷ Keep this node intact; continue walking the tree
        T1 ← CollapseTree(HierarchyTree.leftChild,
                          threshold)
        T2 ← CollapseTree(HierarchyTree.rightChild,
                          threshold)
        return CreateTree(T1, T2)
    else
        T ← combine all descendants of H into single node
        return T
```

4.4.3 Regularity discovery

An essential stage in the floorplanning algorithm is finding repeated patterns of blocks in the netlist. The methodology proposed in this work is based on the ideas of frequent subgraph discovery (FSM), a popular research area within the domain of data mining [90]. As seen in Section 2.1.2, the goal of FSM is to identify repeated subgraphs in a graph.

We consider two distinct subgraphs G_1, G_2 of a graph G to be a repetition if they are isomorphic. In such case, we call both G_1 and G_2 *instances* of the same

repeating *pattern P*. In our formulation, each type of block in the netlist (core, router, memory module, etc.) has a different *label*. Two vertices of a graph are considered to be isomorphic only if they have the same label.

The algorithm is shown in Algorithm 5. REGULARITYDISCOVERY starts from a *FlattenedTree* as input. At every iteration of the inner loop the most frequent pattern is found, and then the netlist graph is compressed with all the instances of such graph. This iterative process generates the regularity DAG as seen in Fig. 4.4.

The outer loop ensures that the existing hierarchy indicated by *FlattenedTree* is preserved. Instead of finding the most frequent pattern of the entire netlist, we initially limit our search to the subcircuits in nodes *FlattenedTree* whose depth is equal to the maximum depth of the entire tree.

Only if no repeating patterns are found in those subcircuits, the search proceeds by enlarging the search are to include all subcircuits in nodes with fewer depth, decreasing the minimum allowed depth (*CurMinDepth*) by one. This way, the search algorithm ensures that patterns that are fully contained inside the partitions marked by hierarchy boundaries are preferred before patterns that do not.

Procedure FINDMOSTFREQUENTPATTERN implements frequent subgraph discovery based on [90]. It is based on a constructive beam search model [120]. At every iteration, we keep a list L of candidate patterns. This list is initialized with all trivial patterns of size 1 (that is, every subgraph with a unique label). At every iteration, every pattern P in L is tested to check which of its instances can be extended by including an adjacent vertex. Each possible extension is stored in L_{new}. The extended patterns in L_{new} are sorted by their VALUE and only a subset of them is selected according to their best value. The number of surviving patterns is determined by b (beam width). The algorithm finishes when no further extensions for any pattern in L can be found.

The VALUE function is used to discriminate between valid patterns when more than one pattern is found for a given graph G. In HiReg the following function is used:

$$\text{VALUE}(P) = \text{“number of instances of } P \text{ in } G\text{”} \times |G| + |P|$$

This value ensures all patterns are ordered firstly by their frequency. When comparing two patterns with the same number of repetitions, the pattern with the largest vertex count ($|P|$) is preferred.

4.4.4 Bounding curve construction

In this stage, a bounding curve is constructed for each one of the regular hierarchy trees discovered by the previous stage. The bounding curve is constructed

Algorithm 5 Frequent subgraph discovery

```
function FindMostFrequentPattern(G)
    G ← input graph
    L ← {∀ label l ∈ G : a subgraph with a single node
          v ∈ G with label(v)= l}
    b ← beam width
    P_best ← Front(L)
    while ¬ Empty(L) do
        L_new ← ∅
        for all P ∈ L do
            for all vertex u ∉ P adjacent to v ∈ P do
                L_new ← L_new ∪ Extend(P, u)
        Sort(L_new by descending Value())
        L ← first b elements of L_new
        if Value(Front(L)) > Value(P_best) then
            P_best ← Front(L)
    return P_best

function RegularityDiscovery(CollapsedTree)
    PatternList ← ∅
    CurMinDepth ← maximum depth of CollapsedTree
    while CurMinDepth ≥ 0 do
        G ← contents of all nodes from CollapsedTree
             with depth >= CurMinDepth
        repeat
            P ← FindMostFrequentPattern(G)
            append P to PatternList
            G ← Compress(G, P)
        until no repeating patterns in G
        CurMinDepth ← CurMinDepth −1
    return PatternList
```

by a post-order walk in the hierarchy tree. This is similar to the techniques used in other hierarchical floorplanners [37, 143].

Every regular hierarchy tree is a directed acyclic graph where every non-leaf node is a subcircuit representing a regular pattern, and and edge between P_1 and P_2 indicates that P_1's definition contains an instance of P_2. Only once the bounding curves for every children of a pattern P have been constructed the algorithm can proceed to construct the bounding curve of P itself.

To construct the bounding curve of a pattern P, two different search strategies are used depending on the number of blocks n. For patterns where n is less than a threshold N, an exhaustive branch-and-bound search algorithm is used. This algorithm explores every possible slicing floorplan.

If $n \geq N$, a more efficient heuristic search based on simulated annealing and slicing trees is used. This heuristic search generates a much reduced number of results than the branch-and-bound approach, but is capable of handling patterns with a much larger number of components. The threshold N depends on the specifications of the host computer, such as the amount of available memory.

The bounding trees for each hierarchy tree are combined into a single bounding curve that represents the outlines of all floorplans found. Because many hierarchy trees will contain similar sets of patterns, HiReg uses memoization in order to decrease the runtime of this stage.

4.4.5 Floorplan construction

After selecting a subset of points from the final bounding curve (Γ), the final stage of the algorithm constructs floorplans starting from the outlines specified by these points.

A single point in a bounding curve can represent multiple floorplans with the same outline, including floorplans that only differ in mirroring and simple block swapping (but also floorplans with entirely different layouts that happen to share the same outline). Because the outline is fixed, the selection of these floorplans cannot affect whitespace, but it may have a significant impact on the wire length and regularity metrics.

The algorithm in Algorithm 6 implements a greedy search that finds a combination of floorplans for a selected point P of the bounding curve that minimizes a given cost function comparing wire length and regularity. By manipulating the cost function, the designer is able to guide the search to either enforce more regular floorplans, or on the other hand prefer floorplans with increased connectivity quality.

An important technique used during this process is *terminal propagation* [59]. When selecting a floorplan for a pattern T, we know the layout and positions of all instances of other subpatterns $t_1, t_2, \ldots, t_n$ contained in T. Thus, for all

nets that have a terminal in a child subpattern t_i, but also have other terminals in any other subpatterns of T, the algorithm can propagate the approximate terminal positions of those terminals outside t_i. This allows calculation of the wire length when selecting floorplans for t_i, even for nets external to t_i.

Algorithm 6 Floorplan construction

function CONSTRUCTFLOORPLANS(P, T)
▷ *P is the selected point of* Γ
▷ *T is the hierarchy tree that was used*
$F \leftarrow$ floorplans from Γ with size P
for all t in CHILDREN(T) **do**
propagate terminal positions from F to t
$p \leftarrow$ shape of t in F
$F_t \leftarrow$ CONSTRUCTFLOORPLANS(p, t)
select combination of $F_1, \ldots, F_n$ that minimizes COST($F, F_1, \ldots, F_n$)
expand F with $F_1, \ldots, F_n$
return F

function COST($F, F_1, F_2, \ldots, F_n$)
return $\frac{\text{WIRELENGTH}(F_1)+\text{WIRELENGTH}(F_2)+\ldots}{\text{number of floorplans in } F_1, F_2, \ldots \text{ with the same layout}}$

To increase floorplan quality or in order to generate more than one result, HiReg combines this algorithm with the technique of beam search [120]. Instead of keeping a single current solution, the b best solutions are kept.

4.5 Results

We implemented HiReg in C++ and tested it on a set of design examples. All the experiments in this section were run on a Intel Xeon 2.8Ghz CPU with 32GB of RAM. While the implementation can make use of multiple cores, it was limited to a single thread for fair comparisons. METIS [87] was used to generate the graph bisections required for hierarchy discovery.

Since there is little previous work on regularity-constrained floorplanning for multi-processors, there are not many available benchmarks designed to compare the quality of regular floorplans. Commonly used floorplanning public domain benchmarks are mostly from old designs that do not contain much regularity. Thus, during this section, we will use artificial benchmarks based on many-core CMP designs.

It is hard to give a numerical metric for regularity in a floorplan. For HiReg generated floorplans, we can provide an estimation of regularity based on the

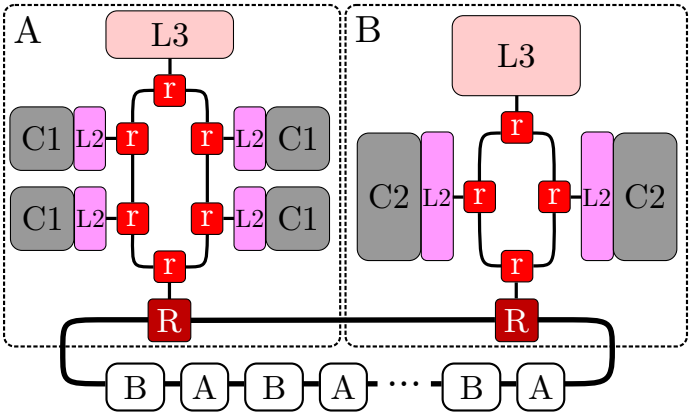


Figure 4.7: Netlist used for the scalability and quality experiments.

regularity DAG used to generate them. Every node in the DAG with multiple input edges represents a sublayout that has been replicated. Thus, a regularity metric can be built by comparing the area of all nodes in this DAG with an expanded version where none of the layouts are replicated:

$$Regularity = 1 - \frac{area\ of\ DAG}{area\ of\ equivalent\ expanded\ tree}$$

Because other tools do not target the creation of regular floorplans, we cannot provide similar regularity metrics for non-HiReg floorplans.

4.5.1 Heterogeneous tiled CMP example

We start this section by mentioning the results presented during Section 4.3, the heterogeneous tiled CMP. We also show the differences in wire length from a floorplanning that preserves hierarchy versus one that does not.

The netlist used represents a CMP containing 64 tiles, with 48 tiles being processing tiles, containing 4 cores each, and 16 memory controller tiles. The tiles are arranged according to the *diamond* pattern from [9]. The benchmark has a total of 816 blocks. The physical information for these blocks is shown in Table 4.4.

For this example CMP configuration, HiReg generates a floorplan (Fig. 4.2d) with 12.6% whitespace and a HPWL of 516 m. Up to 81% of the floorplan area is regular. This floorplan was used by combining hierarchy and regularity. HiReg automatically prefers floorplans where hierarchy is preserved to around the tile level, as those provide the best regularity with minimal loss in other metrics. For comparison, the floorplan in Fig. 4.2b, which was created without hierarchy constraints, has a slightly worse whitespace (13.5%), worse regularity (66.6%

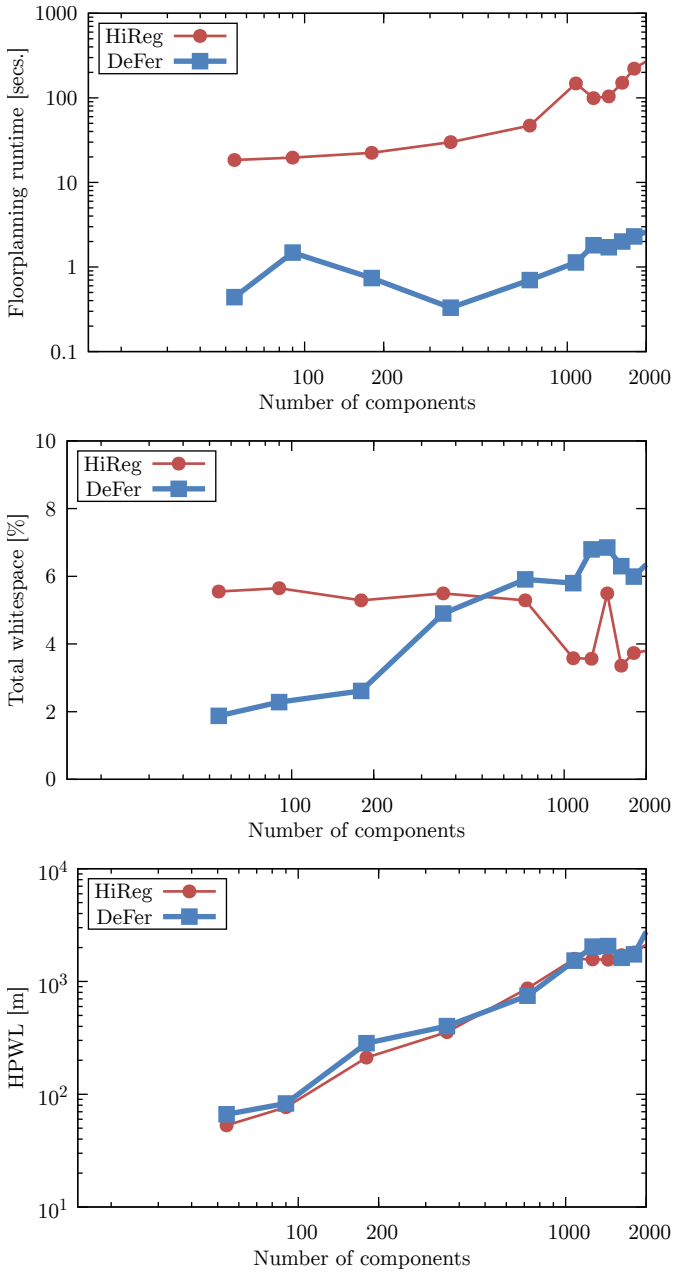


Figure 4.8: Comparison of runtime, area and wire length.

Component	Area	Aspect ratio
Global ring routers (R)	0.27 mm^2	1
Clusters of type A		
4 × Core ($C1$)	1.38 mm^2	0.8 or 1.25
4 × L2 cache	1 mm^2	0.5 ÷ 2
1 × L3 cache	3 mm^2	0.5 ÷ 2
6 × Local ring router (r)	0.27 mm^2	1
Clusters of type B		
2 × Core ($C2$)	3.75 mm^2	0.8 or 1.25
2 × L2 cache	2 mm^2	0.5 ÷ 2
1 × L3 cache	6 mm^2	0.5 ÷ 2
4 × Local ring router (r)	0.27 mm^2	1

Table 4.4: Physical information for Fig. 4.7.

of area) and a much worse HPWL (1076 m). These hierarchy-less floorplans were explored but discarded by HiReg because of the lower metrics.

When compared to other tools (Table 4.3), HiReg provides results that are competitive in wire length but slightly less in area. We configured DeFer to optimize for area and wire length given a maximum aspect ratio constraint of $\frac{4}{5}$. CompaSS was configured to optimize for area within the same aspect ratio constraint. HiReg used 15.4 seconds to generate that example floorplan. 13% of the time was spent during hierarchy and regularity discovery, 27% of the time was spent floorplanning all the discovered patterns, and 60% generating the final floorplans. Both DeFer and CompaSS took less than one second to generate the floorplans.

4.5.2 Scalability and floorplan quality

This experiment measures the loss of optimality in area and wire length caused by the use of regularity, as well as measure the execution time required for the implemented algorithm. We compare HiReg, configured to optimize for maximum regularity, to DeFer, configured to optimize for both area and HPWL.

All the test cases were generated based on the configuration shown in Fig. 4.7. This configuration is a *ring of rings*, where a global ring connects a set of *clusters* of two alternating types. The physical characteristics are detailed in Table 4.4. For this example, every net contains 1024 wires, and the wire pitch is 0.1µm. This configuration is used in this testcase because the total number of clusters in the ring can be easily parameterized, providing multiple testpoints with different numbers of components.

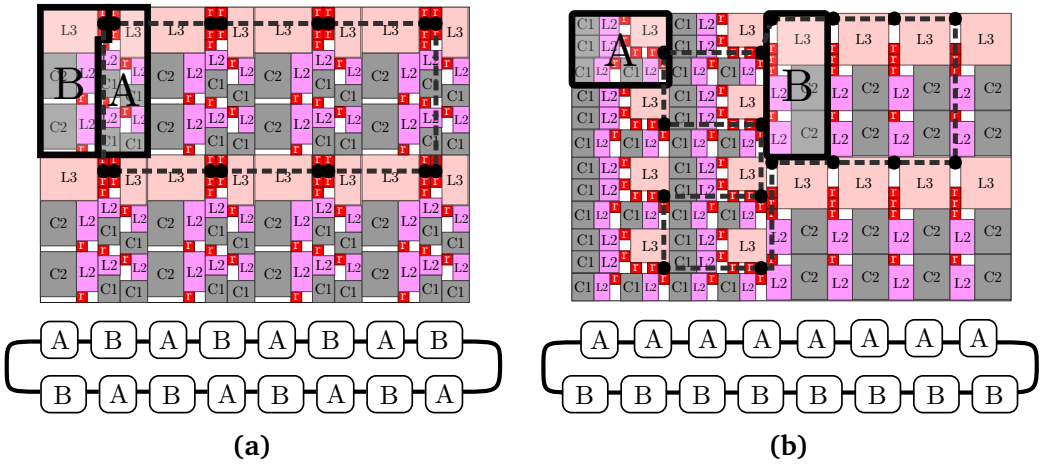


Figure 4.9: Regular floorplans for two different ring configurations.

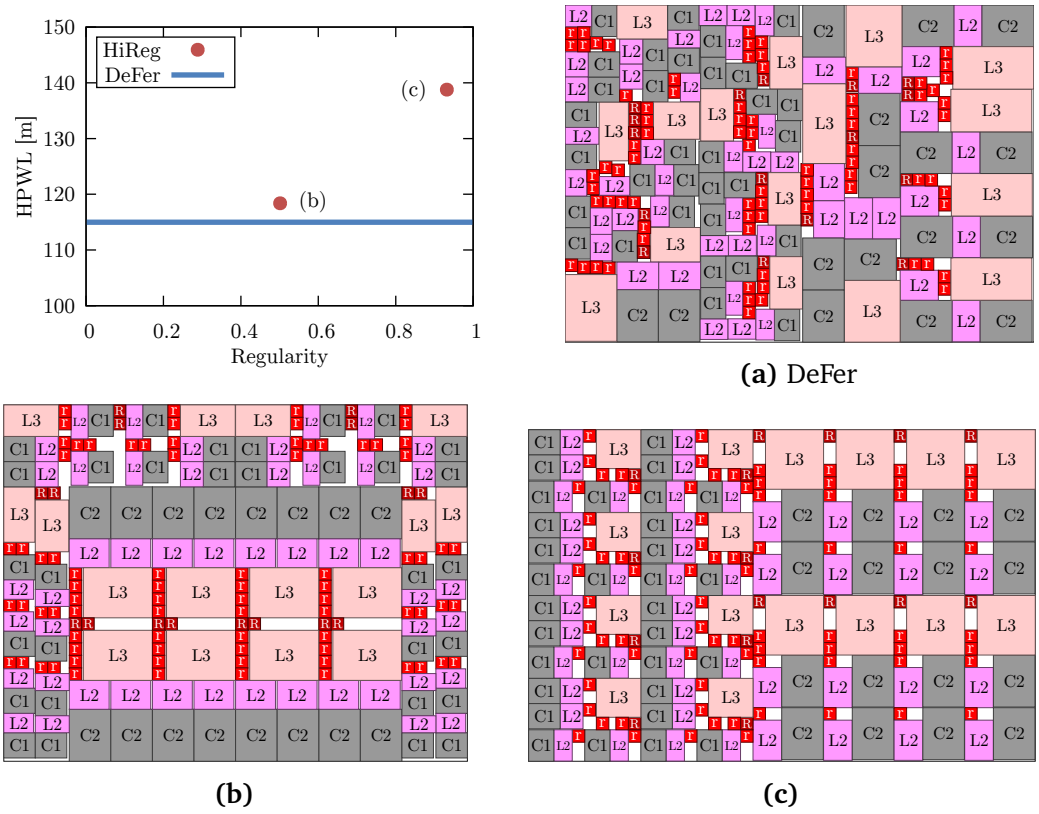
Figure 4.8 shows the results of the comparison. For a highly regular netlist such as the one used in this experiment, the runtime growth of both hierarchical floorplanners (proposed and DeFer) is close to linear. However, a purely hierarchical floorplanner such as DeFer is still much faster. On the other hand, the results show that both the whitespace and HPWL of floorplans generated by HiReg are comparable to the results provided by DeFer.

4.5.3 Trading off hierarchy and regularity

Figure 4.9 shows two different potential configurations for the ring configuration described in Fig. 4.7. These two configurations differ in the organization of the global ring. In configuration a, clusters A and B appear in alternating order in the global ring. Configuration b contains the same clusters but configured so that clusters of the same type are grouped together. The net for the global ring is shown in both figures as a dashed line.

For Fig. 4.9b, HiReg generates a regular floorplan where the different cluster types are strictly separate. The floorplan provides good whitespace metrics because the similar clusters are being packed into arrays. However, in case a, such packing would not be possible without impacting wire length. Therefore, the best floorplan found by HiReg relaxes hierarchy a bit, and combines the two cluster types into a single repeating pattern, providing better packing.

HiReg can also be configured to further relax regularity when improved wire length or other metrics are preferable. In figure 4.10, we show three different floorplans for the configuration in Fig. 4.9b. Figure 4.10b and c are generated by varying the cost function mentioned from Section 4.4.5, while a



(a) DeFer

(b)

(c)

Figure 4.10: Trading off regularity and wire length in configuration described in Fig. 4.9b

is generated using DeFer. The plot shows how by trading off regularity, HPWL can be improved, approaching the HPWL of the floorplan generated by DeFer for the same configuration. As DeFer does not generate regular floorplans, the plot uses only its HPWL as baseline for comparisons.

4.5.4 Publications

As part of this research topic we have published the following conference article:

- J. de San Pedro, J. Cortadella, and A. Roca, *A hierarchical approach for generating regular floorplans*, in Proceedings of the 33th IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, USA, 2014.

4.6 Conclusions

This chapter has introduced HiReg, a new floorplanning tool that generates regular floorplans while preserving the inherent regularity of the design. The method is specially suited for CMPs with many cores and can handle systems with heterogeneous tiles. The method delivers layouts with high regularity and acceptable area, and also reduces wire length when compared to other hierarchical approaches.

Chapter 5

Log-based simplification of process models

The visualization of models is essential for user-friendly human-machine interactions during Process mining. A simple graphical representation contributes to give intuitive information about the behavior of a system. Quality-preserving model simplifications can be of paramount importance to alleviate the complexity of finding useful and attractive visualizations.

This chapter presents a collection of log-based techniques to simplify process models. The techniques trade off visual-friendly properties with quality metrics related to logs, such as fitness and precision, to avoid degrading the resulting model. The algorithms, either cast as optimization problems or heuristically guided, find simplified versions of the initial process model, and can be applied in the final stage of the process mining life-cycle, between the discovery of a process model and the deployment to the final user.

A tool called *PNsimpl* has been developed and tested on large logs, producing simplified process models that are one order of magnitude smaller while keeping fitness and precision under reasonable margins.

This chapter is organized as follows. Section 5.1 exemplifies the goals of this work with a simple example, and Section 5.2 compares these goals with the state of the art. In Section 5.3, a log-based technique to estimate the importance of arcs and places in a Petri net is described. Section 5.4 proposes several algorithms to simplify a Petri net using this information. The techniques are evaluated in Section 5.5. Finally, conclusions are discussed in Section 5.6.

5.1 Motivation

The understandability of a process model can be seriously hampered by a poor visualization. Many factors may contribute to this, being complexity a crucial one: models that are unnecessarily complex (incorporating redundant components, or components with limited importance) are often not useful for understanding the process behind. On the other hand, process models are expected to satisfy all quality metrics when representing an event log: fitness, precision, simplicity and generalization [2]. In this chapter we present techniques to simplify a process model while retaining the aforementioned quality metrics under reasonable margins.

Given a spaghetti-like process model, one may simply remove arcs and nodes until a nice graphical object is obtained. However, this naive technique has two main drawbacks. First, the capability of the simplified model to replay the process executions may be considerably degraded, thus deriving a highly unfitting model. Second, the model components, arcs and places in a Petri net, are not equally important when replaying process executions, and therefore one may be interested in keeping those components that provide more insight into the real boundaries on what is allowed by the process (i.e., its precision). Given a Petri net and an event log, *PNsimpl* first ranks the importance of places and arcs using a simple simulation of the log by the Petri net, and then simplifies the model by retaining those arcs and places that are important in restricting the behavior allowed by the model. Thereby, *PNsimpl* is more accurate at mantaining the precision of the model. Several alternatives are presented, which extract certain Petri net subclasses (State Machines, Free-Choice) or structural subclasses (Series-Parallel graphs).

Some of the proposed alternatives can, as a user decision, also reduce fitness in order to further increase the simplicity of the model. This option can be used when dealing with highly complex processes, where a slightly unfit model may be preferable to a non-understandable model. Additionally, this feature may be used to remove the complexity caused by noise [7] in the original process logs. In Chapter 6, we propose an alternative method to handle highly unstructured process by splitting them into several easier to understand slices.

5.1.1 Example

We will illustrate one of the techniques presented in this chapter with the help of an example. We have used the general-purpose tool *dot* [66] to render these examples. Figure 5.1a reports a process model that has been discovered by the ILP miner from a real-life log, a well-known method for process discovery [138]. This miner guarantees perfect fitness (i.e., the model is able to reproduce all

the traces in the log), but its precision value is low (31.5%) which indicates that the model may generate many traces not observed in the log.

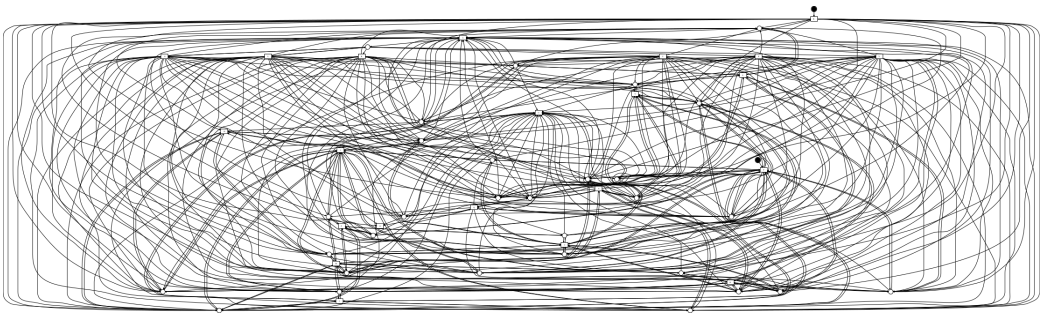
Clearly, this model does not give any insight about the executions of the process behind. Hence, although it is a model having perfect fitness, some of the other quality metrics (precision, simplicity) are not satisfactory. Applying the simplification techniques of *PNsimpl*, a process model can be transformed with the objective of improving its understandability. The process models at the bottom of Fig. 5.1 are the result of applying two of the proposed techniques. In Figure 5.1b the model is simplified while preserving as much as possible the quality metrics of the original model. The model has 6 times less places and arcs, making it much easier to understand. The resulting fitness is still perfect, but the precision has been reduced to 22.5%.

In Figure 5.1c we reduce the model to a *series-parallel graph*, further improving its simplicity and understandability. Fitness has been reduced to 64.1%, but on the other hand its precision has improved considerably (now 48.7%).

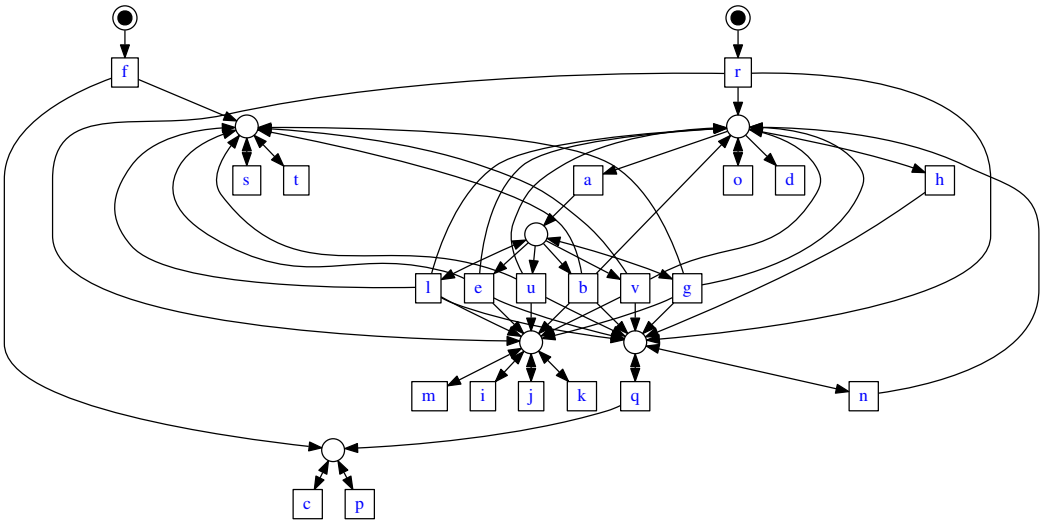
5.2 Related work

Currently, most of the existing academic tools for visualization of process models are based on the *dot* algorithm [66, 67], which is a four-stage approach for laying out directed graphs. *dot* tries to minimizes both edges crossing and edge length. Using *dot* on very dense graphs results in spaghetti-like visualizations. All the models displayed in this thesis have been laid out with *dot*. When the underlying graph has certain structure (as general business process models have), then ad-hoc algorithms that take advantage of this structure can be considered, like the one presented in [70]. In summary, the aforementioned work does not consider log-based simplifications like the ones presented in this work, and therefore, they can be used in combination with the techniques of this chapter to optimize the visualization.

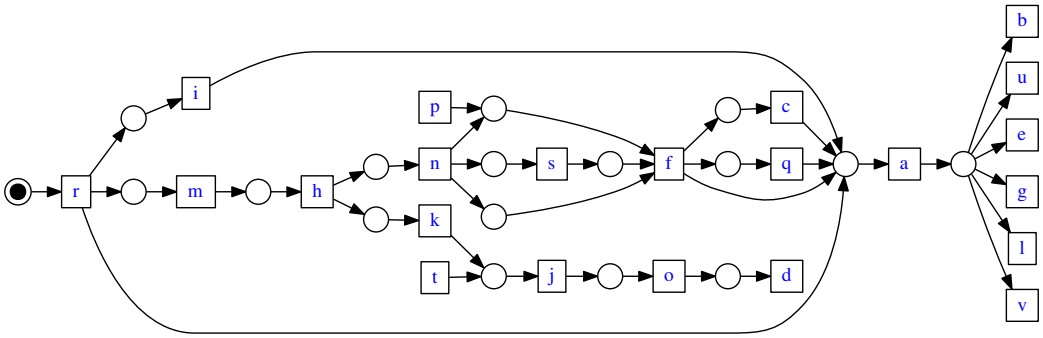
The closest work to the methods of this chapter is [63], where a technique was presented for the simplification of process models that controls the degree of precision and generalization. It applies several stages. First, a log-based unfolding of the model is computed, deriving a precise unfolded model. Second, this unfolding is then filtered, retainning only the frequent parts. Finally, a folding technique is applied which controls the generalization of the final model. Further simplifications can be applied, which help on alleviating the complexity of the derived model. There are significant differences between the two approaches: while in our case, the techniques rely on light methods and can be oriented towards different objectives, the approach in [63] requires the computation of unfoldings, which can be exponential on the size of the initial



(a) Initial process model.



(b) Simplified fitting process model.



(c) Simplified series-parallel process model.

Figure 5.1: Log-based simplification of an spaghetti-like process model.

model [107]. Also, the filtering on the unfolding is done on simple frequency selection on the unfolding elements, while in this work the importance of model elements is assessed with the frequency but also triggering information, which is related to the precision dimension. On the other hand, the techniques of this chapter may need to verify model connectedness at each iteration. In conclusion, both techniques can be combined to further improve the overall simplification of a model.

The simplification of a process model should be done with respect to quality metrics, and in this chapter we have focused on fitness and precision. An alternative to this approach would be to include these quality metrics in the discovery, a feature that has only been considered in the past by the family of genetic algorithms for process discovery [4, 28, 134]. All these techniques include costly evaluations of the metrics in the search for an optimal process model, in order to discard intermediate solutions that are not promising. This makes these approaches extremely inefficient in terms of computing time.

Furthermore, there exist discovery techniques that focus on the most frequent paths [71, 137]. These approaches are meant to be resilient to noise, but on the other hand give no guarantees on the quality of the derived model. Additionally, these methods are oriented towards modeling formalisms with less expressiveness, such as heuristic nets, or formalisms with less strict semantics, such as fuzzy models. A recent technique that is guided towards the discovery of block-structured models (*process trees*) and that addresses these issues may be a promising direction [101]. However, this technique is guided towards a particular class of Petri nets (workflow and sound), describing a very restricted type of behaviors. An important drawback of this technique is that silent activities need to be introduced in the resulting process model to represent the model with this restricted structure. For instance, for the log used to generate the example of Figure 5.3, the inductive miner derives a model with twice the size of the models generated by our techniques. Finally, the techniques of this chapter can be combined with abstraction mechanisms to further improve the visualization of the underlying process model.

5.3 Metrics for relevant arcs

Given a Petri net and an event log, in this section we introduce a technique to obtain a scoring of the arcs (and, indirectly, places) of the net with respect to their importance in describing the behavior underlying in the log.

The idea of the proposed technique is simple: when a Petri net replays a particular trace in the log, some arcs may have more importance than others for that particular trace. Hence, *triggering* and *utilization* scores will be defined

to provide an estimation of the importance of the arcs in replaying the log. Arcs $\mathscr{F}(p,t) \neq 0$ with high trigger score correspond to frequent situations in the model where more behavior should not be allowed (i.e., the arc, and therefore p, is frequently disabling certain transitions to occur). By keeping these arcs/places in the model, one aims at deriving a model where precision is not degraded. Conversely, an arc $\mathscr{F}(t,p) \neq 0$ with high utilization score denotes a situation where transition t is frequently fired (thus frequently adding tokens to p), and therefore should not be removed to avoid degrading fitness.

Definition 5.1 (Trigger Arc)**.** Let $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$ be a Petri net, σ a fitting trace for N, and $e \in \sigma$ an event that is represented by firing $m[t'\rangle m'$ in N. For any pair $p \in P$, $t \in T$, an arc $\mathscr{F}(p,t) \neq 0$ is trigger in $m[t'\rangle m'$ iff t is not enabled in m but enabled in m' and $m(p) < \mathscr{F}(p,t)$ but $m'(p) \geq \mathscr{F}(p,t)$.

Intuitively, an arc $\mathscr{F}(p,t) \neq 0$ is trigger at every transition $t' \in \sigma$ in which t becomes enabled and p is in the set of places which, in that transition t', received the last tokens required for enabling t. Thus, a frequently-trigger arc indicates p is important in restricting the behavior allowed by the model, and that p or $\mathscr{F}(p,t)$ cannot be removed without sacrificing precision. Note that for a single transition t there may be more than one trigger arc, even in the same transition $t' \in \sigma$. To use this information, we define a trigger score which characterizes the frequency of an arc in playing the trigger role:

Definition 5.2 (Trigger Score of an Arc)**.** Let $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$ be a Petri net, and L a log fitting N. The trigger score of an arc $\mathscr{F}(p,t) \neq 0$, denoted by $\mathscr{T}(p,t)$, is the number of transitions from L in which $\mathscr{F}(p,t)$ is trigger.

For $\mathscr{F}(t,p) \neq 0$ arcs, we use a simpler frequency metric:

Definition 5.3 (Utilization Score of an Arc)**.** Let $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$, and L a log fitting N. The utilization score of an arc $\mathscr{F}(t,p) \neq 0$, denoted by $\mathscr{U}(t,p)$, is the number of times transition t is fired in L.

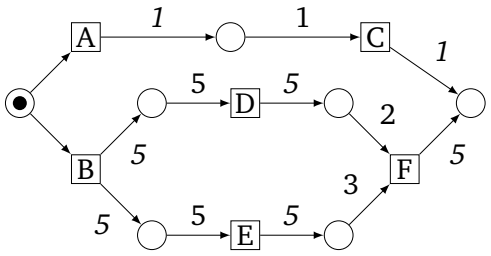
Given a log and a Petri net, obtaining the trigger scores can be done by *replaying* all traces in the log, shown by Algorithm 7. For every transition in the log, the scores are updated by comparing the markings from the predecessor places of all newly enabled transitions.

Figure 5.2 shows the results of computing, on an example trace and model, both trigger and utilization scores. Utilization scores are shown in *italics*.

Finally, notice that the definitions of this section assume fitting traces. Given an unfitting trace (i.e., a trace that cannot be replayed by the model), an *alignment* between the trace and the model will provide a feasible sequence that is closest to the trace [10]. This allows widening the applicability of the scoring techniques of this section to any pair (log, model).

AC
BDEF
BDEF
BDEF
BEDF
BEDF

(a) Example trace



(b) Petri net with trigger/utilization scores

Figure 5.2: Trigger and utilization score computation for an example trace and model.

Algorithm 7 TRIGGERSCORES

Input: An event log L and a Petri net $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$
Output: A score $\mathscr{T}(p,t)$ for every arc $\mathscr{F}(p,t) \neq 0$
for $\sigma \in L$ **do**
  Let $m_0[t_1\rangle m_1[t_2\rangle \ldots [t_n\rangle m_n = \sigma$
  for $i \leftarrow 1 \ldots n$ **do**
    for $t \in T$ **do**
      if t is enabled in $m_i \wedge t$ was not enabled in m_{i-1} **then**
        for $p \in {}^\bullet t$ **do**
          ▷ *t is enabled in $m_i \implies m_i(p) \geq \mathscr{F}(p,t)$*
          if $m_{i-1}(p) < \mathscr{F}(p,t)$ **then** $\mathscr{T}(p,t) \leftarrow \mathscr{T}(p,t) + 1$
return $\mathscr{T}$

5.4 Simplification methods

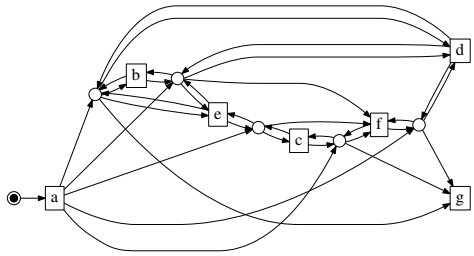
The triggering and utilization scores computed in previous section provide an estimation of the importance of the arcs in replaying the log. Arcs $\mathscr{F}(p,t) \neq 0$ with high trigger score correspond to frequent situations in the model where more behavior should not be allowed (i.e., the arc, and therefore p, is frequently disabling certain transitions to occur). By keeping these arcs/places in the model, one aims at deriving a model where precision is not degraded. Conversely, an arc $\mathscr{F}(t,p) \neq 0$ with high utilization score denotes a situation where transition t is frequently fired (thus frequently adding tokens to p), and therefore should not be removed to avoid degrading fitness.

In this section, 3 different approaches to the simplification problem are shown. Figure 5.3 illustrates these approaches by applying each one to the input model shown in Fig. 5.3a. The first approach reduces the input to a Petri net that is visually close to a series-parallel graph, removing the least important arcs and places according to their scores (Fig. 5.3b). However, it has the greatest computational cost. We introduce a second approach that reduces the simplification problem to an Integer Linear Programming (ILP) optimization problem that is more efficient and optionally guarantees the preservation of fitness (Fig. 5.3c and d). These two techniques use scoring information computed from the log, as described in the previous section. The third approach, however, does not consider this information. Instead, the Petri net is projected into different structural classes: free choice (Fig. 5.3e) and state machine (5.3f). The following subsections will describe each one of these approaches in detail.

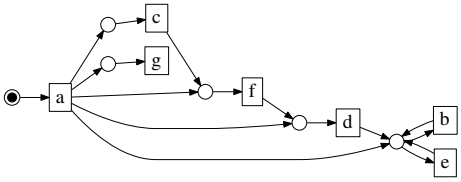
5.4.1 Simplification to a Series-Parallel Net

A series-parallel net is one obtained by the recursive series or parallel composition of smaller nets. Series-parallel Petri nets are amongst the most comprehensible types of models. In a series-parallel net, forks and choices (and thus concurrency) are immediately visible. In fact, existing documentation often uses series-parallel nets as examples to illustrate concepts related to Petri nets.

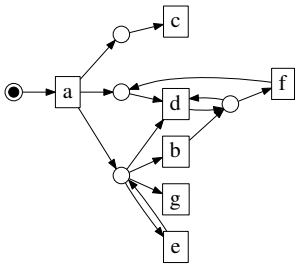
For this reason, one of the main contributions in this work is a heuristic that reduces a complex Petri net into an almost series-parallel net. The algorithm iteratively removes the least important edges until the graph is either strictly series-parallel, or no additional reduction can be applied without losing the connectedness of the net. The importance of every arc is determined by their trigger score $\mathscr{T}(p,t)$, for place-transition arcs, and their utilization score $\mathscr{U}(t,p)$ for transition-place arcs. The approach is grounded in the notion of a set of reduction rules, explained below.



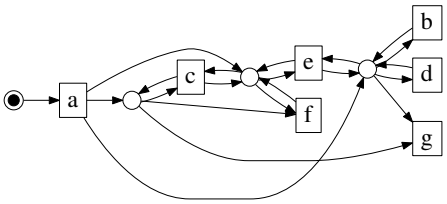
(a) Original model using the algorithm in [138].



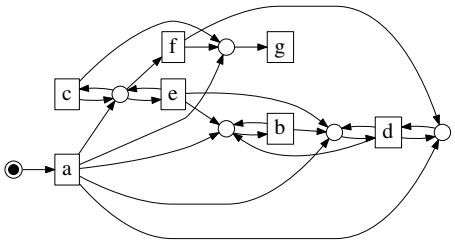
(b) Simplified to series-parallel.



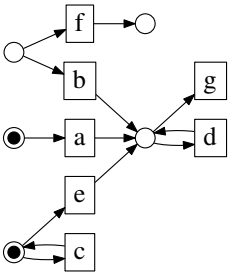
(c) Simplified using ILP model.



(d) Simplified using ILP model, preserving fitness.

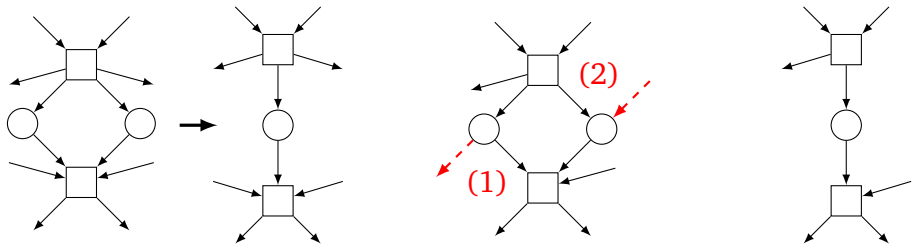


(e) Simplified to free choice.



(f) Simplified to state machine.

Figure 5.3: Overview of the different simplification techniques.



(a) Reduction rule. **(b)** Source Petri net (with rule violations). **(c)** Transformed net.

Figure 5.4: Applying a transformation and transformation cost.

Reduction rules

In [112] a set of reduction rules used for the analysis of large Petri net systems is introduced. Each of the transformations preserves liveness, safeness and boundedness of a Petri net. Thus, verification of these properties can be done in the simplified net instead of the original one. The transformations proposed are: *fusion of series places/transitions*, *fusion of parallel places/transitions* and *elimination of self-loop places/transitions*. A rule can be applied only when its preconditions are satisfied. An example of the *fusion of parallel places* rule can be seen in Fig. 5.4a.

Because of the construction of a series-parallel Petri net, it is possible to reduce such a net to a single place or transition by recursive application of these transformations. Therefore, every violation of the preconditions of a rule indicates a subnet which is not series-parallel.

To reduce a Petri net to a series-parallel skeleton, this work uses these reduction rules in an indirect way. We do not use the transformed Petri net that results from the application of the rules. Instead, the proposed method removes those arcs and places *which prevent the rules from being applied*. For every one of the reduction rules, a *transformation cost* is defined: the sum of the trigger and utilization scores of all the arcs that would need to be removed in order to apply such transformation. The transformation cost therefore models the importance of the arcs that would need to be removed.

Figure 5.4 shows an example rule, the computation of its transformation cost, and the resulting graph after applying the transformation rule. This rule can only be applied in this input Petri net if two arcs (dashed lines in Fig. 5.4b) are removed. Thus, the transformation cost is equal to the trigger score of arc (1) and utilization score of arc (2).

Algorithm

Algorithm 8 describes the main iteration of the method. Function APPLICABLETRANSFORMATIONS identifies all possible applications of the reduction rules, and computes the transformation cost for each of the possible applications.

Algorithm 8 Series-Parallel algorithm

Input: A Petri net $N_0 = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$, a trigger score $\mathscr{T}(p,t)$ for every (p,t) arc, and a utilization score $\mathscr{U}(t,p)$ for every (t,p) arc.
Output: A simplifed Petri net
$N \leftarrow N_0$
$M \leftarrow$ APPLICABLETRANSFORMATIONS(N)
while $|M| > 0$ **do**
  $m \leftarrow$ transformation with least cost from M
  $N' \leftarrow$ APPLYTRANSFORMATION(N, m)
  if N' is disconnected **then**
    $M \leftarrow M \setminus \{m\}$
    continue
  $N_0 \leftarrow N_0 \setminus$ PRECONDITIONVIOLATINGARCS(N, m)
  $N \leftarrow N'$
  $M \leftarrow$ APPLICABLETRANSFORMATIONS(N)
return N_0

At every iteration the transformation m with the least cost is selected, that is, the one that requires removing the least amount of important arcs in order to be applied. Function APPLYTRANSFORMATION applies such transformation m. If applying the transformation breaks the net into more than one connected component, the next best transformation is selected instead. Otherwise, function PRECONDITIONVIOLATINGARCS enumerates all the arcs that had to be removed in order to satisfy the preconditions of transformation m. Those arcs are removed them from the original Petri net N_0. The next iteration repeats the process on the transformed net N', finding new APPLICABLETRANSFORMATIONS only around the nodes that were changed on the previous iteration.

Once no additional reduction rules can be applied (e.g. because the net is now a single place or transition), the algorithm stops. The currently graph N is discarded, and the result of the algorithm is the simplified Petri net N_0. A final postprocessing step removes unneeded places (e.g. without incident arcs).

The nets generated by this heuristic are not necessarily fully series-parallel, since arcs necessary to preserve connectivity are never removed. This is the only method from this work that presents such a global guarantee, with the

other methods providing weaker connectivity constraints. It is also possible to configure the method to generate strictly series-parallel models.

5.4.2 Simplification Using ILP Models

This section shows a different approach to simplify a Petri net for visualization. The selection of which arcs to remove is seen as an optimization problem, and modeled as an Integer Linear Program (ILP). The use of ILP allows for highly efficient solving strategies. On the other hand, some constraints cannot be modeled using ILP. For example, the models attempt to preserve connectivity of the net at a localized level (i.e. ensuring transitions maintain at least one predecessor and successor place), but cannot guarantee global net connectivity.

The aim of the ILP model is to reduce the number of arcs as much as possible. The inputs are a Petri net $N = \langle P, \Sigma, T, \mathscr{L}, \mathscr{F}, m_0 \rangle$, trigger scores $\mathscr{T}(p,t)$ and utilization scores $\mathscr{U}(t,p)$. We define a binary variable $S(p)$ for every $p \in P$, and a binary variable $A(p,t)$ or $A(t,p)$ for every arc in N. In a solution of this model, variable $S(p)$ is 0 when place p is to be removed from the input graph (similarly for arc variables $A(p,t)$ and $A(t,p)$). Below we describe the ILP model in detail.

$$\min \sum_{\mathscr{F}(p,t)>0} A(p,t) + \sum_{\mathscr{F}(t,p)>0} A(t,p) \tag{5.1}$$

$$\text{s.t. } \forall p \in P : S(p) \iff \sum_{t \in p^\bullet} A(p,t) > 0 \wedge \sum_{t \in {}^\bullet p} A(t,p) > 0 \tag{5.2}$$

$$\sum_{\mathscr{F}(p,t)>0} \mathscr{T}(p,t) A(p,t) >= \Gamma \tag{5.3}$$

$$\sum_{\mathscr{F}(t,p)>0} \mathscr{U}(t,p) A(t,p) >= \Phi \tag{5.4}$$

$$\forall t \in T : \sum_{p \in t^\bullet} A(t,p) > 0 \wedge \sum_{p \in {}^\bullet t} A(p,t) > 0 \tag{5.5}$$

$$\forall p \in P : M(p) > 0 \implies S(p) \tag{5.6}$$

$$\forall t \in T, p \in P : \mathscr{F}(t,p) > 0 \wedge S(p) \implies A(t,p) \tag{5.7}$$

The objective function, Eq. 5.1, minimizes the number of preserved arcs. Constraint 5.2 encodes the relationship between A and S variables. A place is retained in the output net iff at least one predecessor/successor arc is retained.

The model ensures that the most important arcs, according to the trigger scores $\mathscr{T}$, are preserved. For this, constraint 5.3 imposes a minimum number of preserved arcs. Γ can be configured as a percentage of the combined trigger

score from all place transition arcs. A similar threshold constant Φ is imposed using the utilization score $\mathscr{U}$ for transicion place arcs (Eq. 5.4).

A fully connected graph cannot be guaranteed by the ILP model. Instead, Eq. 5.5 models a weaker constraint: every transition will preserve at least one predecessor and successor arc. In addition, every place marked in m_0 is always preserved, to avoid deriving a structurally deadlocked model (Eq. 5.6).

Preserving fitness (optional)

The ILP model as described so far does not guarantee preservation of fitness from the original Petri net. A simple modification can ensure that the existing fitness is preserved, at the cost of being able to remove only a reduced number of arcs from the model. Following a well-known result in Petri net theory, removing only $\mathscr{F}(t,p)$ arcs never reduces the fitness of a model for any given log. Constraint 5.7 implements this restriction.

5.4.3 Projection into structural classes

In this section we present ILP models to reduce Petri nets to two types of structural classes: free choice and state machines [112]. These methods do not require a log as they do not use trigger or utilization scores. Therefore, these proposals can be used to simplify Petri nets for visualization even when logs are not available, albeit their results may be of lower quality since scoring information is not used.

Note that [138] can also be configured to generate state machines or marked graphs, but this approach requires having a log. In addition, the models extracted may still be complex because of the requirement to preserve fitness.

Free Choice

In this method, we simplify Petri nets by converting them into free choice nets. This method preserves the fitness of the model, but reduces precision. While this reduction does not necessarily result in models simple enough for visualization, complexity is reduced while mantaining most structural properties. Thus, reducing a dense net into free choice both opens the door to efficient analysis and to further decomposition (state machine or marked graph *covers*) techniques [57].

We encode this definition as a set of constraints and create a ILP problem which maximizes the number of arcs. For every $p \in P, t \in T$, we define a binary variable $A(p,t)$ which indicates whether arc $\mathscr{F}(p,t)$ is preserved.

$$\max \quad \sum_{\mathscr{F}(p,t)>0} A(p,t) \tag{5.8}$$

$$\forall p \in P : |p^\bullet| > 1 \wedge |{}^\bullet(p^\bullet)| > 1 \implies$$

$$\text{s.t.} \quad \sum_{t \in p^\bullet} A(p,t) = 1 \quad \vee \quad \forall t \in p^\bullet, p' \in {}^\bullet t : p \neq p' \implies \neg A(p',t) \tag{5.9}$$

Equation 5.9 guarantees a free choice net. If $|p^\bullet| > 1$ (it is a choice) and ${}^\bullet|p^\bullet| > 1$ (it is not free), then p contains a non-free choice, and one of the conditions must be removed. Either only one of the successor arcs of p is preserved, eliminating the choice, or it is turned free by removing every predecessor arc of $p^\bullet$ except for the ones originating from p itself. Because $\mathscr{F}(t,p)$ arcs are never being removed, this simplification preserves fitness.

State Machine

In a state machine Petri net, every transition has exactly one predecessor arc and one successor arc. To encode this requirement into an ILP model, we again define a binary variable $A(p,t)$ or $A(t,p)$ for every arc in N.

$$\max \sum_{\mathscr{F}(p,t)>0} A(p,t) + \sum_{\mathscr{F}(t,p)>0} A(t,p) \tag{5.10}$$

$$\text{s.t. } \forall t \in T : \sum_{\mathscr{F}(p,t)>0} A(p,t) = 1 \tag{5.11}$$

$$\forall t \in T : \sum_{\mathscr{F}(t,p)>0} A(t,p) = 1 \tag{5.12}$$

Constraints 5.11 and 5.12 encode the definition of a state machine. However, note that this method may reduce the fitness of the model. A similar ILP model can be created to extract a marked graph.

5.5 Results

The methods proposed in this chapter have been implemented in C++. The ILP-based methods have been implemented using a commercial ILP solver, Gurobi [72]. To obtain the input models, the *ILP miner* [138] available in ProM 6.4 was used over a set of 10 complex logs, both real-life [29, 56] and synthetic [137]. The publicly available `dot` utility [66] has been used to generate the visualizations of all the models of this section. The measurements

	Nodes	Arcs	Crossings	Fitness	Precision
(*a*) Original net	13	35	7	100%	43.1%
(*b*) Series-parallel	13	17	0	100%	37.9%
(*c*) ILP model	12	16	0	68%	75.4%
(*d*) ILP (fitting)	11	21	0	100%	40.7%
(*e*) Free choice	13	24	1	100%	31.3%
(*f*) State machine	13	13	0	49.2%	81.3%

Table 5.1: Simplicity, precision and fitness comparison for models in Fig. 5.3.

of fitness and precision have been done using alignment-based conformance checking techniques [10]. Both the logs and our implementation are publicly available at `http://www.cs.upc.edu/~jspedro/pnsimpl/`.

5.5.1 Comparison of the simplification techniques

In Section 5.4 (Fig. 5.3), an artificial model was used to illustrate the different simplification techniques presented in this work. Table 5.1 shows the details for each one of the simplified models.

Several metrics are used to evaluate the results from the simplification techniques. To evaluate the understandability and simplicity of a model, we use the size of the graph, in number of *nodes* and *arcs*, as well as the number of *crossings*. This is the number of arcs that intersect when the graph is embedded on a plane. Thus, a planar graph has no crossings. A graph with many crossing arcs is clearly a *spaghetti* that is poorly suited for visualization. To approximate the number of crossings, the *mincross* algorithm from `dot` [66] is used.

To measure how much the simplified Petri nets model the behavior of the original process we use *fitness* and *precision*, as defined in [10]. In this example, the series-parallel reduction offers perfect fitness, and only 5% loss of precision while removing half of the arcs and all crossings. However, the other methods also remain interesting. For example, the state machine simplification offers the best reduction in simplicity and increases the precision of the model to 80%, at the cost of reducing the fitness by 50%.

This section also includes a comparison with some of the previous work in the area: the *Inductive Miner* (IM) [101] and a *unfolding*-based method [63]. The IM is a miner guided towards discovering block-structured models and which we see as a promising technique (see Section 5.2) since it can be tunned to guarantee perfect fitness. On the other hand, the unfolding procedure is more closely related to the methods proposed in this work. This technique uses an *unfolding* process to simplify an existing Petri net. We have evaluated both methods using the same nets as with our proposed methods.

Figure 5.5 shows the Petri nets produced by the different techniques on a real-life log [56] that is more *spaghetti*-like. The high number of crossings in the original model make it unsuitable for visualization. In this example, the series-parallel method no longer offers perfect fitness but still shows a good trade-off between complexity and fitness/precision. The other methods may be used if for example strict fitness preservation is required, at the cost of more complex models.

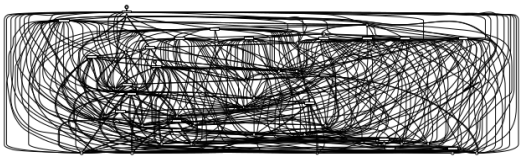
In Fig. 5.6, we compare numerically the techniques of this chapter for the 10 logs. For most of the logs, the series-parallel reduction and the ILP-based techniques are able to reduce the number of crossing edges by several orders of magnitude (note the logarithmic scale), creating small visualizable graphs from models that would otherwise be impossible to layout. On the other hand, the simplification to free choice results in very large and complex models. As mentioned, the benefits of deriving free choice models come from the ability to apply additional reduction strategies. Simplifying to state machines generally produces poorly fitting models, but they tend to have very few crossings and high precision.

The models generated by the Inductive Miner, one of the existing methods included in the comparison, generally contain fewer crossings, caused by the addition of a significant number of *silent* transitions [1] which increase the size of the model. For example, in the *incidenttelco* example the number of transitions of the model derived by the IM is 37, whilst the original model (and, correspondingly, those generated by the simplification techniques) has 22. The addition of silent activities can be beneficial for visualization, specially if the underlying process model is meant to be block-structured.

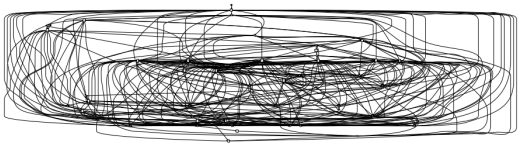
On the other hand, the unfolding procedure is more closely related to the methods proposed in this work. This technique uses an *unfolding* process to simplify an existing Petri net, and has been evaluated using the same nets as with our proposed methods. In general, it produces better results in terms of fitness and precision with respect to the ILP models, at the expense of longer computation time. When compared with the series-parallel method, the results in fitness and precision are comparable, but the unfolding method requires more computation time and the results are worst in terms of visualization.

In Fig. 5.7 we compare the runtimes of the different methods. The ILP solver resolved all the ILP simplification models in less than 1 minute, even for the largest of the input Petri nets from the test set (25K nodes and arcs). The series-parallel simplification, which is not ILP based, has a lower performance. However, there are many parts where the algorithm could be optimized. Still, the total execution runtime for the largest graph (25 minutes) was less than the

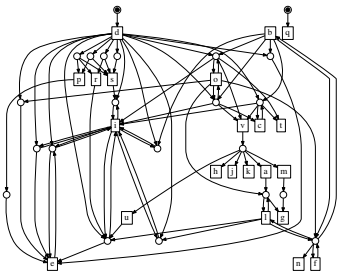
[1] A silent activity in the model is not related to any event in the log.



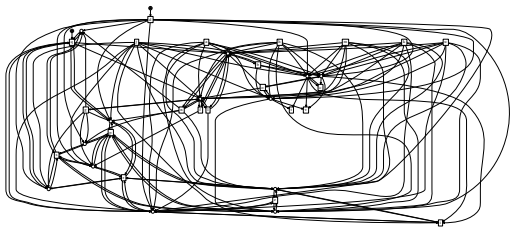
(a) Original Petri net.



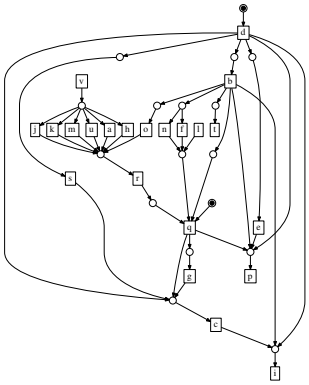
(b) Simplified to Free Choice.



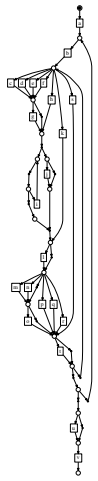
(c) Using ILP model, 60% threshold.



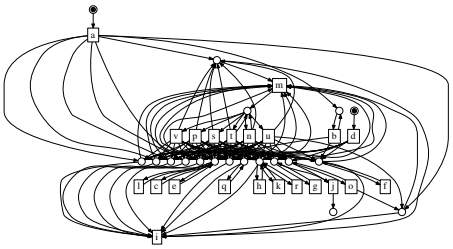
(d) Using ILP model (fitting), 60% thres.



(e) Simplified to Series-parallel.



(f) Using *Inductive Miner* [101].



(g) Using *unfoldings*-based method [63].

	Nodes	Arcs	Crossings	Fitness	Precision
(*a*)	54	448	9805	100%	31.5%
(*b*)	54	320	5069	100%	19.4%
(*c*)	44	93	76	76.7%	42.2%
(*d*)	37	163	728	100%	15.6%
(*e*)	39	54	2	74.5%	37.8%
(*f*)	56	76	0	100%	47.24%
(*g*)	41	158	1448	99.5%	25.8%

(h) Fitness and precision results.

Figure 5.5: Running all methods on real-life log (*incidenttelco*).

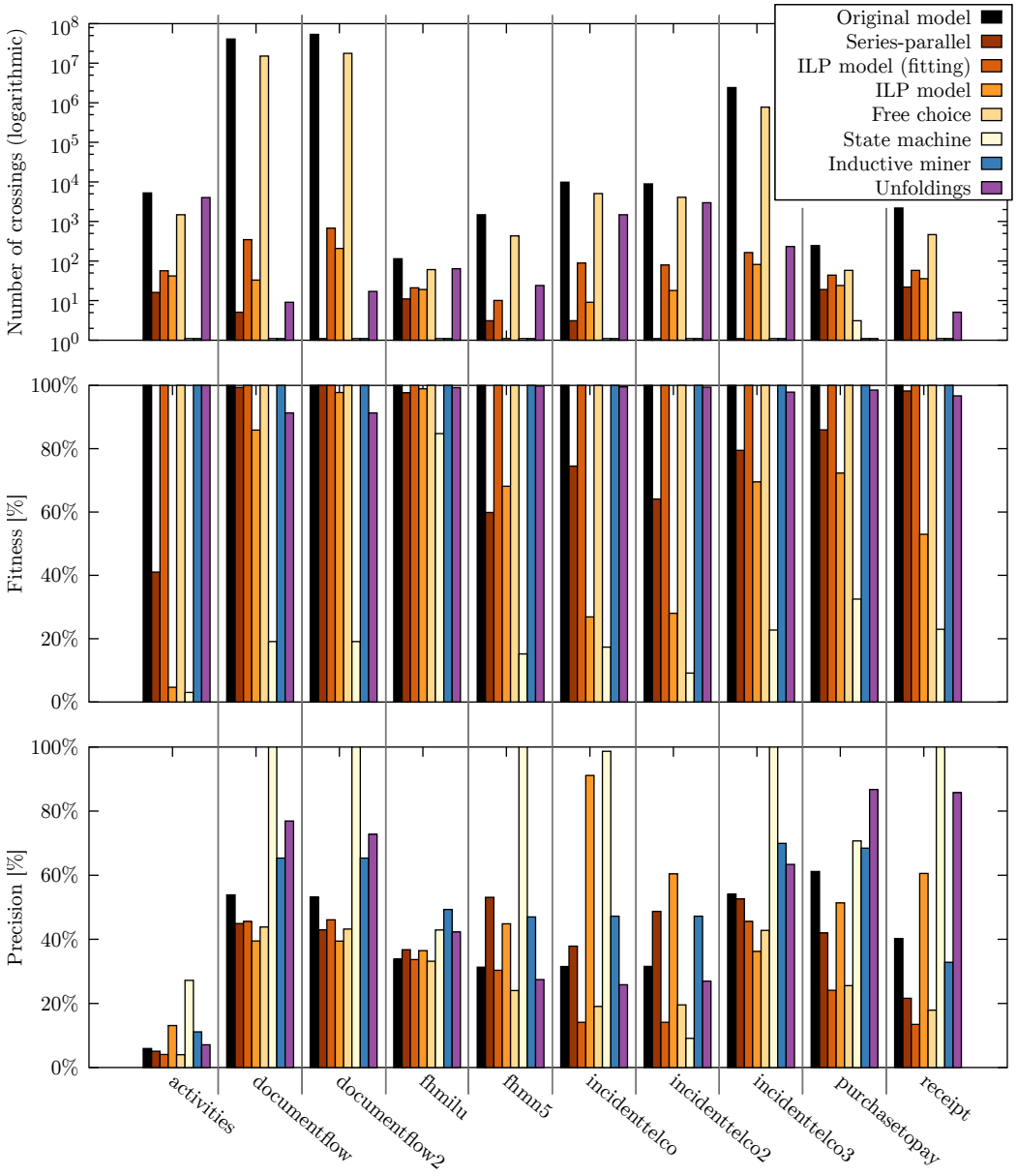


Figure 5.6: Simplicity (logarithmic scale on the number of crossings), fitness and precision comparison between the different techniques using 10 different logs.

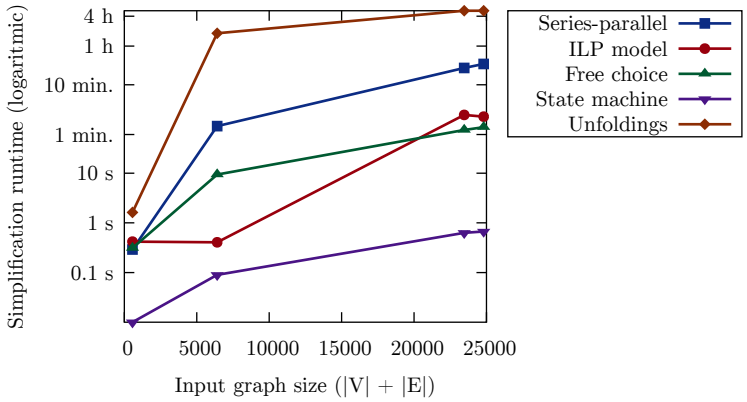


Figure 5.7: Execution runtimes for the different simplification techniques.

1 hour required for the miner in [138] to generate the input Petri net from the log, and significantly less than the 5 hours required by the unfolding technique presented in [63] (also shown in the plot).

The experiments presented in this section show the proposed simplification ILP models to be highly efficient and able to generate models that are orders of magnitude simpler than the original models. If additional simplification is required, the series-parallel method can be used with an increased runtime.

5.5.2 Effect of the ILP model parameters

The ILP simplification model presented in Section 5.4.2 contains a threshold parameter (Γ and Φ) which can be used to tune the complexity and size of the simplified models. In previous experiments and figures, a threshold was set manually so that models with approximately $2|T|$ arcs were generated (where T is the set of transitions from the input Petri net).

To illustrate how varying these thresholds affects the model complexity and quality, the ILP simplification model was executed for each of the input logs, with varying threshold parameters. Fig. 5.8 shows the number of crossings and the fitness for each combination. Generally the fitness decreases with the threshold parameter, but there are some models where the trend reverses. This is because nothing in the model ensures that a log with a given threshold Γ will strictly capture all the behavior of a log simplified using Γ' with $\Gamma > \Gamma'$.

5.5.3 Publications

The methods and the tool described in this chapter have been described in the following conference article:

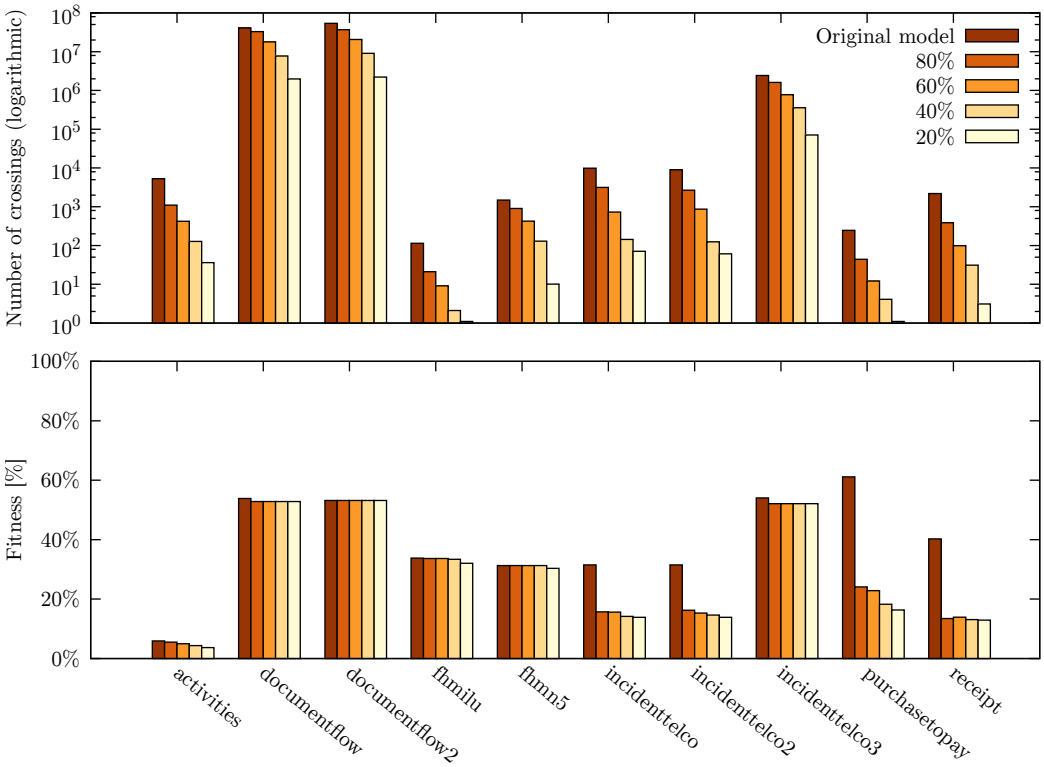


Figure 5.8: Simplicity and fitness comparison using different thresholds for the ILP model.

- J. de San Pedro, J. Carmona, and J. Cortadella, *Log-Based Simplification of Process Models*, in Business Process Management (BPM), Innsbruck, Austria, September 2015.

In addition, the tool is available in `http://www.cs.upc.edu/~jspedro/pnsimpl/`.

5.6 Conclusions

A collection of techniques for the simplification of process models using log-based information has been presented in this chapter. The techniques proposed tend to improve significantly the visualization of a process model while retaining its main qualities in relation with an event log. This contribution may be used on the model derived by any discovery technique, as an intermediate step between discovery and visualization. Also, the analysis of simplified models may be considerably alleviated (e.g., if deriving a free-choice net). The experiments done on dense models have also shown a significant simplification capability in terms of visualization metrics like density or edge-crossings.

Chapter 6

Structured mining of Petri nets

This chapter presents a novel approach for generating process models with structural properties that induce visually friendly layouts. Instead of simplifying an existing complicated model that captures all behaviors, as in Chapter 5, the approach proposed in this chapter delivers a set of models, each one covering a subset of the traces in the log.

The models are mined by extracting slices of labeled transition systems with specific properties from the complete state space produced by the process logs. As the results will show, in most cases a few simple Petri nets are sufficient to cover a significant part of the behavior produced by the log.

In this chapter, Section 6.1 will discuss the objectives of this method, while Section 6.2 will highlight it in terms of the related work. The proposed approach will be given an overview in Section 6.3, while Sections 6.4, 6.5 and 6.6 detail each one of the main steps. Section 6.7 evaluates the method with a set of benchmarks, and the conclusions are discussed in Section 6.8.

6.1 Motivation

Process mining is used to deliver valuable insight into the execution of real-life processes. Real-life processes, however, are often highly unstructured. Traces obtained from running systems generally show repetitive patterns, but also contain plenty of unrelated events.

Building unique and compact models out of such unstructured behavior results in the so-called spaghetti models. As models are generated by automatically learning causal dependencies between different events, fitting many unrelated behaviors into a single model produces highly complex relations. The model may learn fake dependencies between events that are, in truth, unrelated in the original process.

One possible way for simplifying a model is by over-generalizing the model, i.e. allowing behavior that is not present in the execution traces. This is the method that has been explored in Chapter 5. When the log contains unstructured behavior, however, it may be difficult to find a balance between a complicated model and an underfitting one [5].

In this chapter, we propose a new methodology to mine easy-to-understand process models (Petri nets) from logs, even those containing unstructured data. At the core of this proposal is a new *clustering* approach which automatically separates out behaviors with structural properties that induce visually friendly models. Thereby, our approach generates multiple process models, each of them conceptually simple, but without removing any of the behaviors. For most types of processes, only a few models are required to cover a majority of the behavior.

6.2 Related work

The problem of trying to find simple yet fitting models from process logs has been approached from several areas. The closest approaches are those related to clustering. Our approach is novel in that clustering is performed by examining properties of transition systems, not log properties, thus being more accurate in determining which clusters will be visually friendly. A related approach is presented in [56], where the quality and simplicity of the models obtained by the heuristic miner is used to determine similarity of traces. Our approach does not depend on any miner during the clustering process. Most other clustering methods estimate similarity based on the log structure, such as vector distance [127] or edit distance [27]. Significantly, event ordering information can be used to construct the pattern vectors [14]. [61] also targets complexity reduction, but it does so by using hierarchical clustering, unifying repeated patterns.

Another group of techniques try to mine simple nets directly, by selectively ignoring noise from logs, such as the inductive miner [100] or the region-based FSM miner [5]. When two or more significantly distinct behaviors are present in the input logs, these methods must greatly reduce fitness or precision to generate simple models. Our clustering approach, instead, generates separate models for the separate behaviors.

The rest of simplification approaches reduce the complexity of models already mined, such as the approach proposed in Chapter 5. Like the previous strategies, these approaches also end up generating a single model. For example, the authors of [62] perform a log-based unfolding of the model and compute a new model that only keeps the frequent parts.

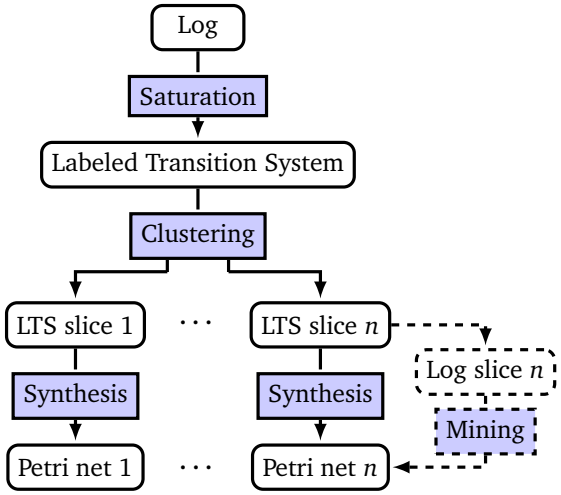


Figure 6.1: Scheme of the proposed flow.

6.3 Structured mining flow

In this section we give an overview of the contributions of this chapter. Figure 6.1 shows the main components of the mining flow.

The most important step is the clustering process, which separates the log into several (possibly non-disjoint) subsets of traces. The classification criteria is based on exploring properties of the labeled transition system (LTS) constructed from the log. The procedure extracts slices from the LTS induced by subsets of transitions satisfying certain properties. Each LTS slice has an associated subset of traces from the log (a log slice).

By ensuring certain properties on each LTS slice, Petri nets with specific structures can be synthesized. This work focuses on *Marked Graphs* and *Free-Choice nets* because of their inherent visually friendly structure. However, the paradigm does not preclude to use other structural properties that may induce other classes of nets. The details on how the slices are generated are described in Sections 6.4 and 6.5.

In the second step, the different log slices are converted into Petri nets (dashed line in Fig. 6.1). Any miner can be used to transform these clusters into a Petri net. However, this work proposes a method (Section 6.6) based on the theory of regions that generates Petri nets based on the LTS slices produced by the clustering step.

6.3.1 Visual example

Figure 6.2 illustrates the different stages of the mining flow with a simple example. Fig. 6.2a contains a fictional event log used as input to the flow.

Fig. 6.2b shows the LTS constructed from the log, where the presence of transition $s_1 \xrightarrow{d} s_{11}$, coloured in Fig. 6.2b, violates certain structural properties (later detailed in Section 6.5). These violations imply that it is not possible to synthesize a visually-friendly Petri net from the LTS. The result is the intricate structure shown in Fig. 6.2c.

The clustering process proposed in this work finds a set of slices of the LTS that satisfy certain structural properties. In this case, the LTS only needs to be split into two slices, shown in Fig. 6.2d. Each slice also fits only a subset of the traces from the log, shown in Fig. 6.2e, and these log slices are the result of the clustering process. In this particular case, the log slices are disjoint. However, overlapping slices can be produced in the most general case.

Fig. 6.2f shows the Petri nets synthesized from the individual LTS slices. The LTS properties give rise to free-choice nets, which are more visually friendly.

6.4 Construction of an LTS from a log

This section details the first step of the mining process, which starts from the input log, constructs a transition system, and performs an initial set of transformations to simplify the relationships between events in the LTS.

Many methods exist to construct an LTS from a log. This work uses a variation of the *prefix multiset* conversion [5] which will detailed below.

In real-life, logs obtained from execution traces often miss information required to fully learn the correct process model. For example, in the log in Fig. 6.3a, events a and b are seemingly concurrent as a is still enabled after firing b and viceversa. The log shows trace $b\,a$, but does not contain $a\,b$. The original prefix multiset technique would generate an LTS where a and b are not concurrent (Fig. 6.3b). The technique presented below generates an LTS where $a\,b$ is also feasible.

For an input log $L \in \mathscr{B}(\Sigma^*)$, the procedure creates a new LTS A as follows:

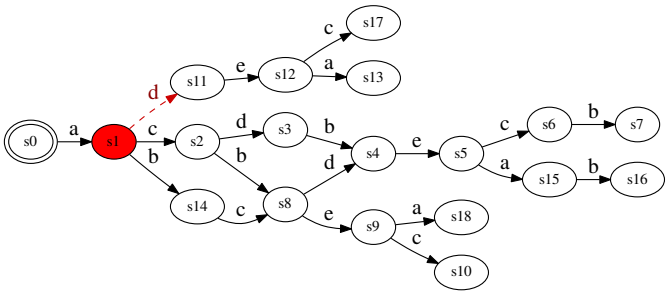
1. The Parikh vector $\psi(\sigma)$ of every possible prefix $\sigma \in \Sigma^*$ appearing in L is computed. A new state is created for every such different Parikh vector.

2. A transition $s_1 \xrightarrow{e_i} s_2$ is inserted in A for every pair of states $s_1, s_2 \in A$ if $\psi(s_1) = (x_0, \ldots, x_i, \ldots, x_n)$ and $\psi(s_2) = (x_0, \ldots, x_i + 1, \ldots, x_n)$.

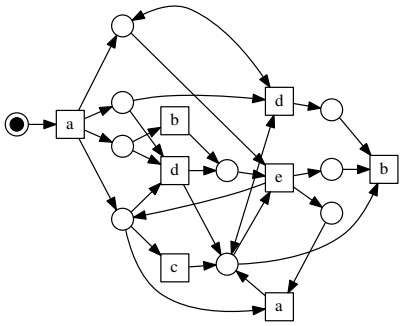
3. The initial state s_0 is the zero Parikh vector.

a b c d e a b
a c d b e c b
a c b d e a b
a c b e a
a d e c
a c b e c
a d e a

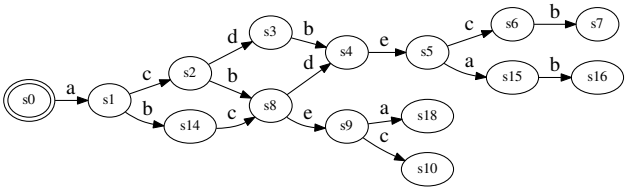
(a) Input log.

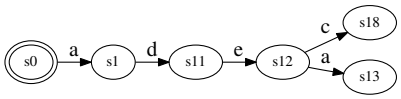


(b) Labeled Transition System.



(c) Petri Net covering the full log.



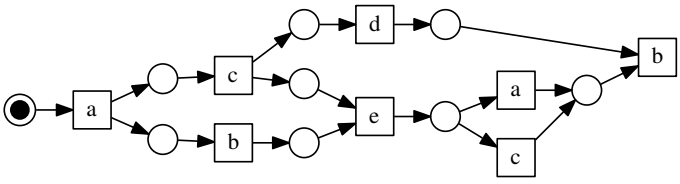


(d) Two slices extracted from the LTS.

a b c d e a b
a c d b e c b
a c b d e a b
a c b e a
a c b e c

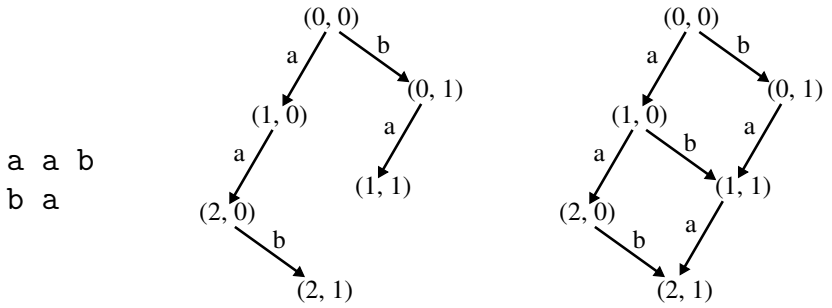
a d e c
a d e a

(e) Corresponding log-slices.



(f) Synthesized Petri nets.

Figure 6.2: Examples at different stages of the mining flow.



(a) Log. **(b)** *Prefix multiset* LTS. **(c)** Arc-completed LTS.

Figure 6.3: Log and steps to construct LTS.

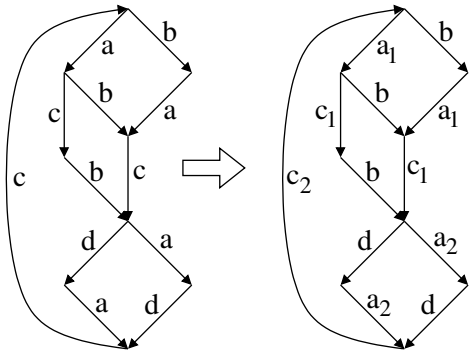


Figure 6.4: Splitting maximal connected ESs.

An example of a log and the LTS generated using this method can be seen in Fig. 6.3c.

The transformations presented in the rest of the section aim at enforcing unique concurrency/conflict relationships between pairs of events.

6.4.1 Splitting events with disconnected Excitation Sets

It is frequent to observe disconnected instances of the same event in different states of the LTS. This transformation considers each instance as a different event.

The transformation splits each event e into a set of events $e_1 \dots e_k$ such that the ESs of each event is maximally connected (only has one connected component). Figure 6.4 illustrates this transformation with an example, in which event a has two maximally connected sets of states in $ES(s)$. Therefore, two new events, a_1 and a_2 are created to substitute the original event a. A similar situation occurs with event c.

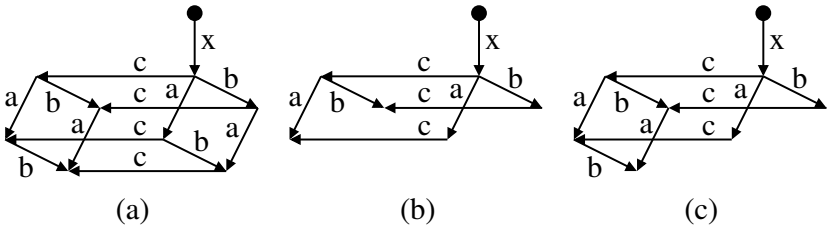


Figure 6.5: Examples to illustrate *c-saturation*.

When synthesizing a Petri net from the LTS, these events may end up by being represented by different transitions in the Petri net if the Free-Choice property requires the splitting. The decision about whether the splitting is necessary is left at the criterion of the synthesis tool [47].

6.4.2 Saturation of concurrency

The goal of this transformation is to define a unique relationship between every pair of events. For every pair a and b, there are two options:

$$ES(a) \cap ES(b) = \emptyset \quad \Rightarrow \quad \text{ordered.}$$
$$ES(a) \cap ES(b) \neq \emptyset \quad \Rightarrow \quad \text{concurrent and/or conflict.}$$

This transformation aims at disambiguating the second case by exclusively choosing between concurrency or conflict. The following definition formally describes what concurrency saturation is.

Definition 6.1 (C-saturation)**.** Two concurrent events are *concurrency-saturated* (c-saturated) if they are concurrent in all states of $ES(a) \cap ES(b)$. An LTS is said to be c-saturated if all pairs of events are c-saturated.

An analogue concept can be defined for the *reversed* LTS, in which the direction of all transitions has been reversed.

Figure 6.5 shows three LTSs with different concurrency properties. The one in 6.5a is c-saturated since all pairs of concurrent events, (a, b), (b, c), and (a, c), are c-saturated. The one in 6.5b is also c-saturated with the pairs (a, c) and (b, c) being concurrent. However, 6.5c is not c-saturated since the pair (a, b) is concurrent, but not in all the states of $ES(a) \cap ES(b)$. Therefore the pair (a, b) is also in conflict.

Any LTS can be transformed into c-saturated by adding states and transitions to complete the missing diamonds of the concurrent events. This process can be applied iteratively, both in the original and reversed LTS, until reaching a fixpoint in which all pairs of concurrent events are c-saturated.

In the example of Fig. 6.5, the LTS in 6.5c would become the one in 6.5a after c-saturation. With this transformation, there is no pair of events that can be concurrent and in conflict simultaneously. This contributes to remove intricate event relations, thus resulting in simpler Petri net structures.

6.5 Extraction of LTS slices

Once an LTS A representing the input log L is constructed and transformed, the following step extracts several LTS slices $A_1, A_2, \ldots, A_k$ satisfying a set of properties than make it amenable for the synthesis of visually friendly Petri nets. Each slice A_i covers a subset of traces (log-slice) L_i of L. The output of the clustering process is a set of log-slices that completely cover L. In the example of Fig. 6.2, two log-slices are delivered, shown in Fig. 6.2e.

We first describe the properties enforced in the LTS slices and then the satisfiability model that extracts the slices.

6.5.1 Properties of the LTS slices

Three properties are desired in the LTS slices: forward persistency, backward persistency and free-choiceness.

Persistency is a property tightly related to the state spaces of Marked Graphs (see Section 2.2.2). An LTS $A = \langle S, \Sigma, T, s_0 \rangle$ is *forward persistent* (FP) if

$$\forall s_1 \xrightarrow{a} s_2, s_1 \xrightarrow{b} s_3 : \ \exists s_3 \xrightarrow{a} s_4.$$

Informally, if two events are simultaneously enabled, they must be concurrent. *Backward persistency* (BP) is an analogous property applied to the reversed LTS (reversing the direction of transitions). It is known that forward and backward persistency are necessary conditions for an LTS to be the state space of a Marked Graph [21].

The third property is free-choiceness (FC). An LTS is said to be free-choice if for every pair of events a and b, the following condition holds:

$$a \text{ and } b \text{ are in conflict} \implies ES(a) = ES(b).$$

FC characterizes the state space for conflicts in Free-Choice Petri nets. Given that two transitions in conflict have the same predecessor places in a free-choice net, the set of markings in which they are enabled is identical for both transitions. This means that both excitation sets will be identical in the corresponding LTS. The FC property is the one used in [47] to split events and guarantee free-choiceness.

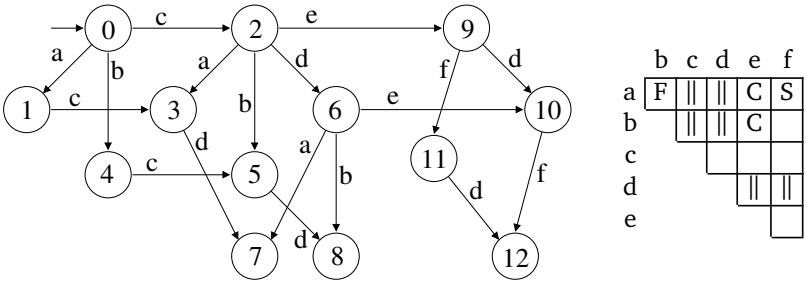


Figure 6.6: LTS with various conflict relations between events.

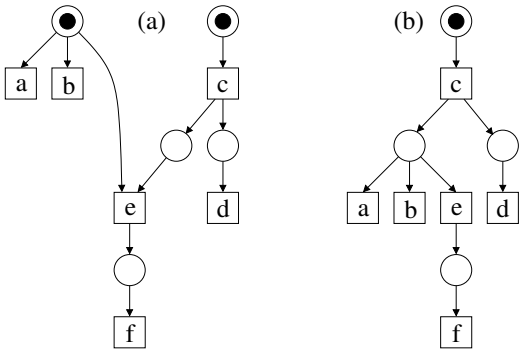


Figure 6.7: Petri nets obtained from the synthesis of the LTS in Figure 6.6: (a) from the the full LTS, (b) from a slice obtained by removing states 1 and 4.

Figure 6.6 shows an LTS and the relationship between pairs of events (||: concurrent, C: conflict, F: free-choice conflict). Blank cells in the table represent ordered events (disjoint ES's). We observe that the only free-choice conflict is between a and b, given that $ES(a) = ES(b) = \{s_0, s_2, s_6\}$. The conflicts for the pairs (a, e) and (b, e) are not free-choice.

Figure 6.7 depicts two Petri nets obtained from the LTS in Fig. 6.6. The one in Fig. 6.6a has been obtained by synthesizing the full LTS, whereas the one in Fig. 6.6b has been obtained by a slice in which states 1 and 4 have been removed. In the latter case, all conflicts for the pairs (a, b), (a, e) and (b, e) become free-choice, and so does the Petri net.

6.5.2 Satisfiability model

An LTS slice is simply a subset of the original LTS. The goal of the approach presented in this chapter is to extract LTS slices with the FP, BP and FC properties. Let us call them *Well-Behaved (WB) slices*.

If each transition $t_i \in T$ of the LTS is represented by a Boolean variable, the set of WB-slices can be characterized by a Boolean formula WB(T) in which every satisfying assignment corresponds to a WB-slice that contains only the transitions t_i for which their variables are asserted.

Fortunately, the function WB(T) can be easily constructed by observing that the FP, BP and FC properties can be formulated locally, i.e., in terms of neighbor transitions. Once the WB formula is constructed, a SAT solver can be used to extract slices. Alternatively, the SAT model can be transformed into an Integer Programming model in which some specific cost function can be optimized.

Let us now see how the FP, BP and FC properties can be characterized with Boolean constraints. Note that after the transformations applied to the LTS, all pairs of events have a unique relationship, i.e., they can only be concurrent or in conflict (but not both) in case their ESs intersect.

Forward persistency (FP)

This property is only applicable to pairs of concurrent events (a, b). For every state $s_1 \in ES(a) \cap ES(b)$, a diamond exists with the following transitions:

$$t_1 = (s_1, a, s_2), t_2 = (s_1, b, s_3), t_3 = (s_2, b, s_4), t_4 = (s_3, a, s_4).$$

Any selected subset of $\{t_1, t_2, t_3, t_4\}$ must preserve the FP property, which means that the selection of t_1 and t_2 must imply the full diamond. The constraint can be formulated as follows:

$$(t_1 \wedge t_2) \implies (t_3 \wedge t_4).$$

Backward persistency (BP)

This property is analogous to the previous one, but reversing the direction of the transitions. A similar Boolean formulation can be constructed.

Free-choiceness (FC)

This property is applied to pairs of events in conflict. The constraints must ensure that both events have the same excitation sets in case both events are present in the LTS slice.

The formalization of the constraints for each pair of events in conflict can be stated as follows:

> Once one of the events is enabled in one state, then the same enabling relation must be maintained in the successor states reachable by other events.

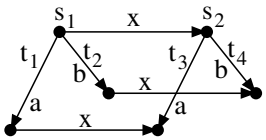


Figure 6.8: Example to illustrate the FC constraint.

Figure 6.8 depicts an scenario in which FC must be applied for two events, a and b, that are in conflict. Notice that the event x with transition $s_1 \xrightarrow{x} s_2$ must be concurrent with a and b, otherwise a, b would not be enabled in s_2, according to the c-saturation transformation applied to the LTS.

Three options are possible with regard to the selection of transitions t_1 and t_2 at state s_1:

1. Both t_1 and t_2 are selected: in this case, the two events must be enforced to have the same ESs and, thus, both or none of t_3 and t_4 must be selected, i.e., $t_3 \Leftrightarrow t_4$.

2. Only one of them is selected, say t_1. t_3 would be present (because of the FP condition) and t_4 cannot be selected, otherwise $ES(a) \neq ES(b)$.

3. None of them is selected. In this case, no constraints are imposed on t_3 and t_4 with regard to state s_1.

Formally, the Boolean constraint applicable to s_1 with regard to events a and b is stated as follows:

$$\begin{aligned} (t_1 \wedge t_2) &\implies (t_3 \Leftrightarrow t_4) \\ (t_1 \wedge \neg t_2) &\implies \neg t_4 \\ (\neg t_1 \wedge t_2) &\implies \neg t_3 \end{aligned}$$

The previous constraints can be simplified if a and b are in conflict but not all transitions of Fig. 6.8 are present in the original LTS. For example, if t_1 is not present ($\neg t_1$), the previous constraints would be simplified and reduced to:

$$t_2 \implies \neg t_3.$$

Generation of choice-free Petri nets.

The previous constraints characterize LTS slices that derive Petri nets in which all choices are free. A simple modification of the model can characterize Petri nets without choices (only causality and concurrency relations). In particular, the FC conditions can be rewritten to disable the selection of two transitions that are in conflict. This would be equivalent to adding the constraint $\neg t_1 \vee \neg t_2$.

6.5.3 Trace coverage

The conjunction of constraints for FP, BP and FC conform the Boolean formula WB(T) that characterizes all WB-slices of the LTS. The question now is: which subset of slices should be extracted? Two properties are desired:

- Every trace of the log must be covered by at least one WB-slice.
- A small number of WB-slices should cover the majority of traces of the log.

At this point, the satisfiability problem becomes an optimization problem that can be modeled as an Integer Programming model with only binary variables.

A log L is a set of traces $L = \{\sigma_1, \ldots, \sigma_n\}$, and each trace σ_i covers a set of transitions $\sigma_i = \{t_{i_1}, \ldots, t_{i_k}\}$ of the LTS, according to the construction described in Section 6.4.

Each trace can be represented by a Boolean variable σ_i that acts as a trace selector. The assertion of σ_i implies the selection of all transitions of the trace in the LTS, i.e.,

$$\sigma_i \implies (t_{i_1} \wedge \cdots \wedge t_{i_k})$$

Including this constraint in the model, a subset of traces can be covered by selecting their variables. The number of covered traces is maximized by incorporating a cost function:

$$\max \sum \sigma_i$$

6.5.4 Algorithm for extracting WB-slices

Algorithm 9 Extraction of WB-slices

1: **Input:** A log L
2: **Output:** A set of pairs (LTS slice, Log slice)
3: $A \leftarrow$ c-saturated LTS from L
4: $R \leftarrow L$ ▷ *Remaining (uncovered) traces from L*
5: $i \leftarrow 1$
6: **while** $|R| > 0$ **do**
7: $A_i \leftarrow$ SOLVEWB(A, R) ▷ *extract new LTS slice*
8: $T_i \leftarrow$ traces from L fitting A_i ▷ *associated traces*
9: $R \leftarrow R \setminus T_i$ ▷ *subtract from remaining traces*
10: $i \leftarrow i + 1$
11: **return** $(A_1, T_1), (A_2, T_2), \ldots, (A_n, T_n)$

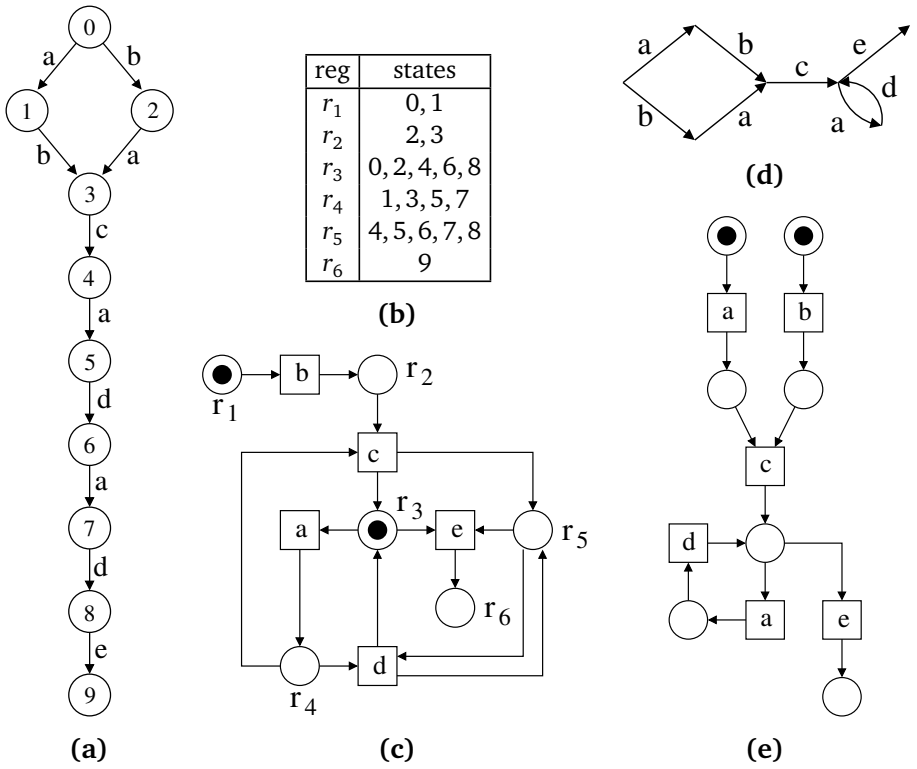


Figure 6.9: Synthesis of Petri nets using regions.

Algorithm 9 shows the main algorithm for extracting LTS and log slices. The algorithm iterates until each trace in log L is covered by at least one of the extracted LTS slices. Variable R stores the list of traces not yet covered.

Function SOLVEWB extracts an slice from A that maximizes the number of covered traces from R. Still, this slice may cover traces already covered by previous slices (in line 8, the traces assigned to T_i are obtained from L, not R).

6.6 Synthesis of Petri Nets

In addition to the main clustering procedure, we propose a method to transform the LTS slices into Petri nets, extending a region-based synthesis tool, *Petrify* [45]. Region-based miners are known for their tendency to construct overfitting models [2]. The proposed modification allows Petrify to construct Petri nets that trade off precision for visualization-friendliness, and implements the ideas of [35].

Any mining tool can be used to obtain process models from the log slices generated by the clustering method. However, the synthesis procedure described

in this section reuses the LTS slices constructed by the clustering algorithm, and thus has inherent benefits in both efficiency and simplicity. For full details on the theory of regions, we refer to [47]. The following section briefly surveys the basics of the theory.

6.6.1 Theory of regions

The theory of regions provides an algorithmic method to derive a Petri net from an LTS. Given an LTS $A = (S, E, T, s_0)$ and a subset of states $r \subseteq S$, a transition $s \xrightarrow{e} s'$ *enters* r if $s \notin r \wedge s' \in r$, *exits* r if $s \in r \wedge s' \notin r$, and *does not cross* r otherwise. r is a *region* if for every event $e \in E$, all transitions $t \in T$ labeled e are in the same relation with r (entering, exiting, or not crossing). A region r is *minimal* if no subset of r is a region. Figure 6.9a shows an example LTS, with 6.9b listing the minimal regions.

Intuitively, a region r corresponds to a place p in the Petri net. p will precondition any of the events exiting r. Thus, it will be marked only in states corresponding to those in r, receiving tokens only from transitions corresponding to events entering r. Straightforwardly, to construct a Petri net from the list of minimal regions, a transition t_i is created for every event $e_i \in A$ and a place p_j is created for every minimal region r_j. If e_i enters r_j, then $p_j \in {}^\bullet t_i$. Conversely, if e_i exits r_j, then $p_j \in t_i^\bullet$. p_j is marked iff $s_0 \in r_j$.

Excitation Closure

Let $\mathscr{L}(A)$ be the *language* of the LTS A, i.e., the set of all traces fitting A, and $\mathscr{L}(N)$ the set of all traces fitting N, where N is a Petri net constructed from A using the above method. Assuming $e \in E$, let ${}^\circ e$ be the set of minimal regions of A where e is exiting. A is *excitation closed* [47] if:

$$\forall e \in E,\ {}^\circ e \neq \emptyset \wedge ES(e) = \bigcap_{r \in {}^\circ e} r$$

$\mathscr{L}(A) = \mathscr{L}(N)$ is only guaranteed if A is excitation closed [47]. If not, then $\mathscr{L}(A) \subseteq \mathscr{L}(N)$ [35], N fits more traces than A.

Figure 6.9c depicts the Petri net N obtained from the minimal regions of the LTS A shown in 6.9a, whereas 6.9d shows the LTS that models $\mathscr{L}(N)$. Since the LTS in 6.9a is not excitation closed, N fits more traces than A (e.g., trace `abce`).

Traditional region-based synthesis tools, including Petrify, transform the LTS to guarantee the excitation closure property and thus enforce $\mathscr{L}(A) = \mathscr{L}(N)$. These transformations often degrade the visualization quality of the resulting Petri nets [35] and result in hard to understand overfitting models which are not ideal for Process Mining. For these reasons, our modification drops the

	Log size		Time [sec.]	
Log	**Unique traces**	**Event types**	**Our proposal**	**ActiTraC**
documentflow	1411	70	90	95
fhmilu	701	12	60	106
fhmn5	693	13	174	38
incidenttelco	212	22	32	17
kim	1174	18	37	20
purchasetopay	76	21	7	55
receipt	116	27	15	7
tsl	1908	42	111	60

Table 6.1: Time required for clustering.

excitation closure requirement to avoid these transformations. Because of this relaxation, the resulting Petri nets are less overfitting.

Finally, event splitting can contribute to further enhance the structure of the Petri net while keeping the recognized language. Figure 6.9e shows a new structure after splitting event `a`. The details on how label splitting may be performed are discussed in Chapter 7.

6.7 Results

We have implemented the algorithms described on this work in Python, using the PMLAB [36] package. 8 logs combining real-life sources [29, 56] and benchmarks [137] have been used as inputs. The logs and our implementation are available at `http://www.cs.upc.edu/~jspedro/pnsimpl/`. Gurobi [72] has been used to solve all ILP models. The quality of the models has been evaluated using the ETConformance plugin in ProM, which computes the best alignment between trace and log before measuring fitness and precision [10, 111].

As for the important issue of visualization-friendliness, there are several studies in the literature [108]. In this work we propose a metric related to the concept of planarity: the minimal number of crossings required to embed the graph on a plane. As described in Section 2.1.1, the *mincross* algorithm from *dot* [66] is used to obtain an estimation of this number.

In order to compare our proposal with the related work we propose two experiments. In the first one, we show how many slices and crossings (Cros.) are required by our mining process, depending on how much behavior of the log is preserved. We also compare it to *ActiTraC* [56], a state-of-the-art clustering procedure that also targets visualization.

In the second experiment, we show how our slicing approach results into visualization-friendly models even when using alternative miners.

Our proposal						
	85%		90%		95%	
Log	**Slices**	**Crossings**	**Slices**	**Crossings**	**Slices**	**Crossings**
documentflow	1	0	1	0	4	8
fhmilu	1	4	1	4	1	4
fhmn5	1	1	1	1	1	1
incidenttelco	1	0	1	0	2	1
kim	1	0	1	0	2	0
purchasetopay	1	0	1	0	1	0
receipt	1	3	1	3	1	3
tsl	1	3	3	3	10	9

ActiTraC						
	85%		90%		95%	
Log	**Slices**	**Crossings**	**Slices**	**Crossings**	**Slices**	**Crossings**
documentflow	1	0	2	14	3	946701
fhmilu	6	4	*Timeout computing fitness* [a]			
fhmn5	3	1	6	2	8	2
incidenttelco	1	3	1	3	2	3
kim	1	2	1	2	2	3
purchasetopay	1	2	1	2	1	2
receipt	2	5	2	5	2	5
tsl	1	25	1	25	2	145

[a] Because of the large number of Petri Nets, replay fitness could not be measured for this benchmark.

Table 6.2: Number of slices and crossings required to reach specific fitness thresholds.

	Using Petrify as miner						Using α miner			
	1st slice (our method)			**Naive method**			**1st slice** (our)		**Naive method**	
Log	**Fit.**	**Prec.**	**Cros.**	**Fit.**	**Prec.**	**Cros.**	**Fit.**	**Cros.**	**Fit.**	**Cros.**
documentflow	92.1%	81.7%	0	97.9%	57.2%	0	37.8%	0	37.5%	2286
fhmilu	97.8%	64.5%	4	*Out of memory*			38.0%	8	16.8%	29
fhmn5	99.1%	49.4%	1	*Out of memory*			40.6%	4	25.5%	116
incidenttelco	92.6%	53.7%	0	99.3%	30.7%	14	63.1%	9	46.7%	53
kim	92.9%	75.1%	0	94.6%	84.6%	38	66.4%	7	56.1%	80
purchasetopay	99.8%	68.2%	0	94.0%	100.0%	0	96.9%	6	100.0%	0
receipt	98.0%	71.3%	3	95.9%	100.0%	0	99.8%	8	76.6%	1
tsl	82.7%	83.9%	3	99.7%	53.1%	1	*Timeout*		75.7%	310

Table 6.3: Comparing the first slice from the proposed clustering method vs. a naive noise removal algorithm (removing 20% least frequent traces), using different miners.

6.7.1 Minimum number of log slices

We configured our implementation to generate as many slices as required in order to obtain a set of slices that include at least 80% of traces from the input log. Petrify was used to generate models for each of the slices. The size of the logs, in number of unique traces and event types, as well as the time spent during the slicing process is shown in Table 6.1. Similarly, *ActiTraC* was configured with the default settings, but a stop criteria of having generated enough clusters to cover 80% of the traces. By default, ActiTraC uses the Heuristic Miner to generate models for the clusters.

The models mined by both tools are generally underfitting, i.e., precision is sacrificed to aid model visualization. Thus, each model allows for more traces than those present in the corresponding log slices. For this reason, measuring entire log replay fitness using ProM usually results in values higher than 80% of the log as configured.

In this experiment, we measure how many slices are actually required to reach a specific level of fitness when replaying the entire input log. For both clustering algorithms, the slices with the largest number of traces are selected first. This number of slices estimates how much behavior a clustering algorithm can fit into a single model. However, the clustering algorithm needs to ensure each model is still visualization-friendly. To estimate this, we measure the total number of crossings present in each of the selected slices.

The results, shown in Table 6.2, indicate that our approach compares positively with ActiTraC. For most examples, the first slice already provides with a model that fits 90% of the original full log. In addition, these first slices have few or almost no crossings.

6.7.2 Metrics of the first slice

In the second experiment, we repeat the proposed slicing procedure on the same logs, but center on the metrics of the slice containing the highest number of traces, which we call the *first slice*. Two distinct miners are used: Petrify (using the modifications described in Section 6.6) and the α-miner [8]. The experiment shows how the slices generated by the proposed algorithm are visualization friendly even when using other types of miners.

As a baseline for comparisons, we also show the metrics of models obtained directly from the log, without applying our slicing approach, but after applying a *naive* noise filtering algorithm. This naive strategy removes the least frequent 20% traces from each log, effectively removing most infrequent behavior. Without this noise removal, attempting to mine the full log would result in huge spaghetti models which would be meaningless to compare with the models

generated after our slicing procedure. Yet, the results show that even when compared to models where most noise has been removed, using our slicing algorithm still results in simpler models with comparable fitness (Fit.) and precision (Prec.).

Table 6.3 shows these metrics on the first slice of each of the logs as well as using the naive algorithm. When using Petrify as miner, some of the models generated after the naive noise elimination still have tens of crossings or low precision. The models generated all have very few crossings despite maintaining similar levels of fitness and precision.

When using the α-miner, low-fitting spaghetti models are discovered even after noise elimination. When applied to the first slices, the α-miner discovers models with a significantly reduced number of crossings, and increased fitness.

6.7.3 Publications

As part of this research topic we have published the following conference article:

- J. de San Pedro and J. Cortadella, *Mining Structured Petri Nets for the Visualization of Process Behavior*, in Proceedings of the 2016 ACM Symposium on Applied Computing (SAC), Pisa, Italy, April 2016.

6.8 Conclusions

In this chapter, we have introduced a new method to mine visualization-friendly Petri nets from real-life process logs. We have shown our method to improve on the visualization quality of other similar slicing methods while being competitive in performance.

This work is just an initial incursion into the study of properties of labeled transition systems with the goal of process model visualization. We envision this effort to be a starting point for further simplifications using other structural properties.

Chapter 7

Discovery of duplicate tasks

In Chapters 5 and 6 we have presented a series of methods to improve the understandability of process models. The methods presented in Chapter 5 trade off quality metrics, such as fitness or precision. In Chapter 6, unstructured event logs are divided into several process models instead of constructing a single model. The methods proposed in this chapter, however, construct a single simplified model while preserving or even increasing fidelity metrics.

The first problem addressed in this chapter is the discovery of duplicate tasks. A new method is proposed that avoids overfitting by working on the transition system generated by the log. The method is able to discover duplicate tasks even in the presence of concurrency and choice.

The second problem is the structural simplification of the model by identifying optional and repetitive tasks. The tasks are substituted by annotated events that allow the removal of silent tasks and reduce the complexity of the model. An important feature of the methods proposed in this chapter is that they are independent from the actual miner used for process discovery.

This chapter is structured as follows. Section 7.1 gives an overview of the topic. In Section 7.2, we discuss the related work. Section 7.3 introduces the concept of a *local excitation set* that will be used during duplicate task discovery. Section 7.4 describes the first proposed technique: a method to discover discover duplicate tasks. The second technique, a set of structural transformations to simplify a Petri net, is shown in Section 7.5. Both techniques are evaluated in Section 7.6. Finally, Section 7.7 presents the conclusions.

7.1 Motivation

This chapter presents a set of techniques to explore the trade off between *simplicity* and *precision*. More specifically, by introducing a small number of

new elements, the proposed techniques result in tangible improvements in precision. They can work in combination with any existing discovery (mining) algorithm. While some of the techniques can be applied to different formal models, this work will focus on Petri nets.

The first technique enables the discovery of duplicate tasks in process models. Duplicate tasks allow several nodes to refer to the same activity in the event log, and are thus ideal to represent different behaviors for the same task in a single event log. While this is not a new concept in Process Mining [7,34,54,126,133], our proposal is novel in that the splitting criteria is based on properties of Labeled Transition Systems, thus allowing more precision than other existing techniques for duplicate tasks.

The second technique performs structural simplifications that do not modify the semantics of the model, thus preserving the quality metrics. We introduce extensions to the formalism that allow single nodes to represent more complex control-flow structures, such as loops or optional tasks.

7.1.1 Example

Figure 7.1 will be used to illustrate the main contributions of this chapter. We start from a simple log, a subset of which is shown in Fig. 7.1a. Figure 7.1b shows the model discovered by the Inductive Miner [101] for this log. This model is highly imprecise (50%): while it is not a pure flower model, almost all the words are recognized.

The reason why many discovery algorithms generate such a low-precision model is because of the presence of *duplicate tasks* in the original process. Several reasons can lead to this scenario. For example, two different tasks may have been improperly tagged with the same label.

Figure 7.1c shows the process model after the discovery of some duplicate tasks. The original process had two different tasks for each of the labels a, b, and e. This information is discovered automatically using the methods proposed in this work. Duplicate tasks also allow the discovery of more precise models. In this particular case, the new model has a precision of 90% and the workflow structure is clearer. However, the model has increased the total number of components, including silent transitions, which unnecessarily increases cognitive load in this example.

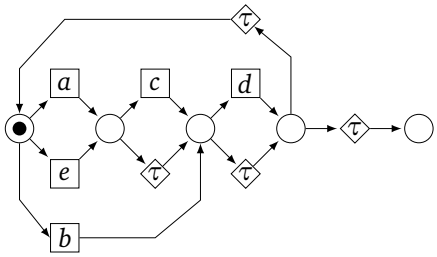
Many of the silent transitions in Fig. 7.1c can be removed without affecting the semantics of the model, as shown in Fig. 7.1d. A method to remove silent transitions will also be presented in this work.

By applying some structural transformations to Fig. 7.1d, further reductions on the structure of the Petri net can be achieved. In this work, the alphabet of labels is enhanced to incorporate *meta-transitions*, which represent control

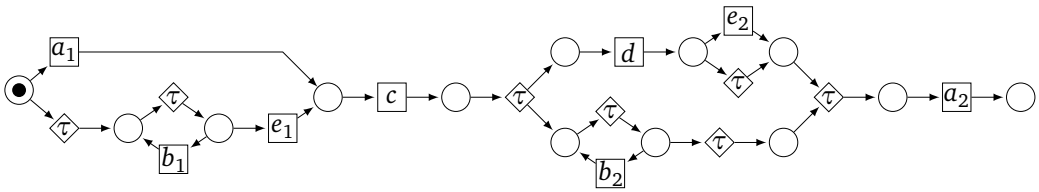
a c d e a
a c d b a
a c b d b e b a
a c b b d b b e b b a

b b b e c d e a
b b b e c d b a
b b b e c b d b e b a
b b b e c b b d b b e b b a

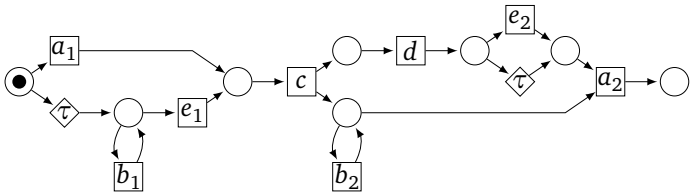
(a) Subset of the example log.



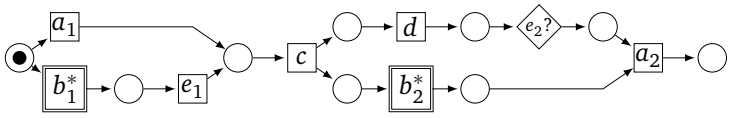
(b) Model discovered by the Inductive Miner.



(c) Model constructed after duplicate task discovery.



(d) Removal of unneeded silent transitions from (c).



(e) Using *meta-transitions* to simplify (d)

Figure 7.1: Applying the method presented in this chapter to a sample model discovered by the Inductive Miner.

flow patterns. For example, an $e?$ meta-transition can replace a choice between e and a silent transition, as in Fig. 7.1d. Similarly, a meta-transition $b*$ can sometimes replace a self-loop transition with label b. We introduce Petri net transformations that, in this particular example, allow the removal of all silent transitions without altering its behavior.

7.2 Related work

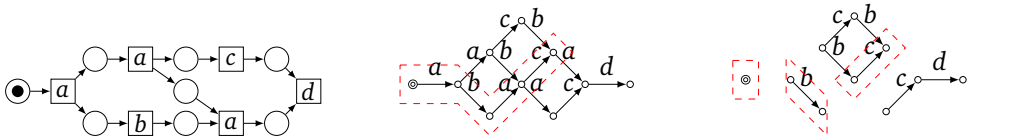
Several methods already exist for duplicate task detection, with the approaches being classifiable into several families. In [133], a set of heuristics creates a candidate set of duplicate tasks, which is then explored by a local search procedure working in tandem with an arbitrary mining algorithm. The method produces high-quality results in combination with advanced miners. However, since the miner influences the direction of the search, it is difficult to predict the runtime of the discovery process. In this work, the miner algorithm is only used to evaluate the set of candidate results. The number of results is exactly bounded by the maximum number of allowed duplicate tasks per event. The work in [126] proposes a clustering approach based on the context of events similar to the one described in this work. However, our work uses excitation sets to identify the context of events, which allows for more accurate detection that using the log directly.

A different family of methods to perform duplicate task detection are tied to specific mining technologies. For example, Fodina [132], Genetic Miner [54], AGNEs [69], InWoLvE [74], region theory [34], α^*-algorithm [103]. The proposal in this work works with any mining algorithm, and does not require e.g. workflow nets or other specific process models.

For the second proposal in this work, structural simplifications, a potential comparable work is the use of other process modeling notations, such as BPMN [140]. However, the formalisms presented in this chapter still allow the full expressiveness of Petri nets, yet hide complexity in the presence of the common flow control operators.

7.3 Local Excitation Sets

Figure 7.2b shows an LTS constructed from the process in Fig. 7.2a. Notice how $ES(a)$ contains the states in which the three duplicate tasks of a are enabled. We now define the concept of *local excitation set* which distinguishes each such instance of a, including the concurrent ones:



(a) Initial process. **(b)** $ES(a)$ in the LTS. **(c)** The three $LES(a)$ after removal of a-transitions.

Figure 7.2: Calculation of Local Excitation Sets.

Definition 7.1 (Local Excitation Set)**.** Given LTS $A = \langle S, \Sigma, T, s_0 \rangle$ and event $e \in \Sigma$, the *local excitation sets* of e, $LES(e)_1, \ldots, LES(e)_k$ are the maximally connected subsets of $ES(e)$ such that, $\forall s_1 \xrightarrow{e} s_2 \in A$, if $s_1 \in LES(e)_i$ and $s_2 \in LES(e)_j$, then $i \neq j$.

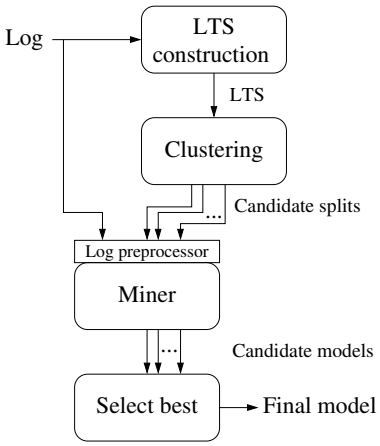
Notice that the definition does not allow both the source and target states of a transition with label e to be in the same $LES(e)_i$. The set of LES of an event can be efficiently computed with a simple algorithm, illustrated in Fig. 7.2c for event a. The algorithm has the following steps: (1) calculate $ES(a)$, (2) remove the transitions with label a from the LTS, (3) identify all $LES(a)$ as the maximally connected subsets of $ES(a)$ after the removal of the a-transitions.

The next two sections describe the two simplification techniques introduced in this work: discovery of duplicate tasks and structural simplifications to represent control-flow patterns.

7.4 Discovering duplicate tasks

This section introduces a method that automatically discovers which events from an event log correspond most likely to duplicate tasks, i.e. should be represented by more than one task in order to enhance the quality of the model. The technique works with the LTS constructed from a log and can be combined with any discovery algorithm. By adding new tasks, the method slightly increases the element count of the model but results in tangible improvements in precision.

Given a log L, the goal of this procedure is to generate, for every activity $a \in L$, a *partition* of all the events in L referring to a. When mining a process model, every different partition will be represented by a different task. We will generally refer to each task by $a_1, a_2, \ldots, a_n$. A partition that, for every activity a, maps all events into a single task a_1 results in a model with no duplicate tasks. Figure 7.3b shows an example partition for the log in Fig. 7.1a.



(a) Overview of the flow.

a_1 c d e_2 a_2
a_1 c d b_2 a_2
a_1 c b_2 d b_2 e_2 b_2 a_2
a_1 c b_2 b_2 d b_2 b_2 e_2 b_2 b_2 a_2

b_1 b_1 b_1 e_1 c d e_2 a_2
b_1 b_1 b_1 e_1 c d b_2 a_2
b_1 b_1 b_1 e_1 c b_2 d b_2 e_2 b_2 a_2
b_1 b_1 b_1 e_1 c b_2 b_2 d b_2 b_2 e_2 b_2 b_2 a_2

(b) Split version of the log from Fig. 7.1a.

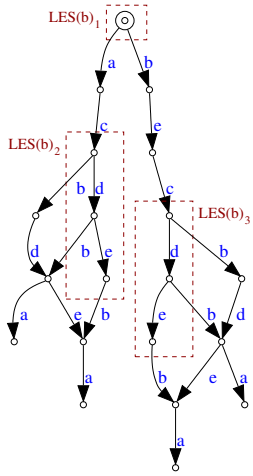
Figure 7.3: Summary of the duplicate task discovery process.

An overview of the proposed method is shown in Fig. 7.3a. At the core of the proposal lies a clustering process that generates a small set of candidate partitions. An existing mining algorithm is used to generate a process model for each of these partitions, and the best model is selected out of these discovered models. This way, the method adapts to the subtleties of the different mining algorithms. Even for miners that automatically discover duplicate tasks, the proposed method may help improving the results.

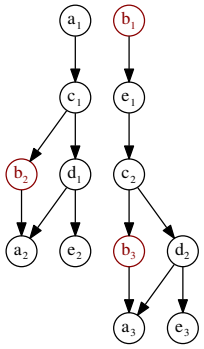
The clustering method uses a bottom-up (*agglomerative*) approach: starting from the trivial partition which maps every event to a different task, the procedure iteratively selects pairs of *similar* events, grouping them into the same task. To find similar events, the algorithm uses causality relationships between events as discovered in a LTS, instead of using log information directly (e.g. direct predecessors or successors of an event). An LTS can be built from the log with a variety of methods [5]. Section 7.4.1 describes how similar events are found in the LTS, while Section 7.4.2 details the actual clustering algorithm.

7.4.1 Partitioning based on Excitation Sets

A significant difference between this proposal and previous approaches to duplicate tasks is that the proposed method works at the Transition System level. The log is first converted into an LTS, and the clustering procedure generates a partition *based on causality relationships between excitation sets in this LTS*, rather than directly using the preceding and successor events in the log. Because of this, the approach is resilient to processes where duplicate tasks are combined with concurrency and choice. The use of clustering-based



(a) Constructed LTS, highlighting all $LES(b)$.



(b) Excitation set graph of the LTS in (a).

LES	Triggers	Triggered by
a_1	$\{c\}$	$\emptyset$
a_2	$\emptyset$	$\{b,d\}$
a_3	$\emptyset$	$\{b,d\}$
b_1	$\{e\}$	$\emptyset$
b_2	$\{a\}$	$\{c\}$
b_3	$\{a\}$	$\{c\}$
c_1	$\{b,d\}$	$\{a\}$
c_2	$\{b,d\}$	$\{e\}$

(c) Trigger relations between LES.

Figure 7.4: Example excitation set graph of a subset of Fig. 7.1a (loops removed).

methods [82] and similarity metrics rather than looking for exact matches also allows the proposed flow to gracefully handle noise and incompleteness in the log.

Let us use an example to show the benefits of using ESs. Figure 7.4a shows the LTS constructed from the log in Fig. 7.1a, with no duplicate task detection performed. For simplicity, loops have been removed (allowing one iteration only). As per the definition of LES, there are 3 LESs for activity b, shown in Fig. 7.4a.

Notice how the LESs of b provide an intuitive view of the correct partition for activity b (as shown in Fig. 7.3b): $LES(b)_1$ corresponds to the events of task b_1, while $LES(b)_2 \cup LES(b)_3$ would correspond to b_2. Our proposal classifies these LES by their relationships with other LES. The *excitation set graph* represents all the LES of a TS as well as the causality relationships between those:

Definition 7.2 (Excitation Set Graph)**.** Given a LTS $A = \langle S, \Sigma, T, s_0 \rangle$, the excitation set graph of A is a graph $ESG(A)$ where:

- The set of vertices $V(ESG(A))$ corresponds to the set of LES of A.
- For every pair $(LES(a)_i, LES(b)_j)$ of A, with $a, b \in \Sigma$, there is an edge $(LES(a)_i, LES(b)_j) \in E(ESG(A))$ iff for any $s_1 \in LES(a)_i$ and $s_2 \in LES(b)_j$, $s_1 \xrightarrow{a} s_2$ triggers b.

Figure 7.4b shows the corresponding excitation set graph of the example LTS, while 7.4c summarizes the immediate trigger relations. Notice how the information on 7.4c allows us to trivially distinguish between $LES(b)_1$ and $\{LES(b)_2, LES(b)_3\}$, since $LES(b)_1$ triggers a different set of events.

Compare this to using predecessor and successor information from the log directly, without constructing an LTS first. It is difficult to distinguish events of b by looking at the immediately following event. For example, an event b followed by e may indicate an instance of task b_1 as discovered in the previous section, but it may also be caused by an instance of b_2, since it is concurrent with e. Thus, using log information only, it would be difficult to construct an accurate partition for b. The use of excitation sets avoids this problem.

Even when using excitation sets, the combination of choice, loops and/or incomplete logs may introduce LES that have related but slightly different sets of predecessors/successors, yet should be mapped to the same task. For this reason, the proposed flow includes a clustering method that combines *similar* LES. This method is described in the following section.

7.4.2 Hierarchical clustering algorithm

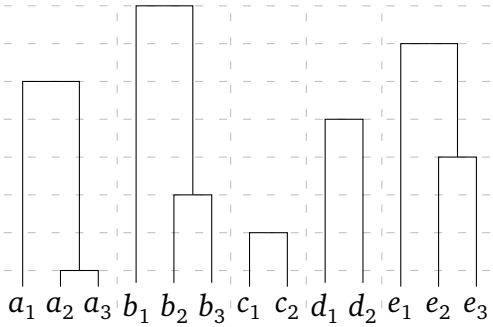
This section describes the method used by our proposal to classify local excitation sets into groups with similar causality relationships. The described clustering method is *agglomerative* [82], discovering clusters using a bottom up approach: the algorithm starts by assuming every that, for every activity a, every $LES(a)_i$ belongs to its own cluster. In this initial solution, each LES maps to its own duplicate task. Then, the algorithm considers the pairwise similarity of all the LES, and combines the two closest $(LES(a)_i, LES(a)_j)$ (of the same activity a) into the same task a_i. The entire process iterates until no further clustering can be performed. On every iteration, the algorithm explores a solution with exactly one duplicate task less than the previous solution.

The full discovery algorithm can be seen in Algorithm 10. The input is a log L. A is an LTS constructed from L (for example using the methods described in [5]), while G is the initial ESG, constructed using the rules seen in 7.2. The output R is a process model with duplicate tasks.

In every iteration, procedure FINDMOSTSIMILARNODES selects two vertices of G with the most similar *context vectors*, a numeric way to represent their causality relations which will be explained in the following section. The selected vertices are then merged into a single new vertex, representing the new cluster, which inherits the causality relationships of the merged vertices. Note that only vertices with the same activity label will be selected. The loop ends when there is only vertex in G for every activity, i.e. there are no duplicate tasks.

Algorithm 10 Discovery flow with duplicate tasks

```
1: function DUPLICATETASKDISCOVERY(L, M)
   ▷ L is the input log, M is a miner algorithm.
2:     A ← CONSTRUCTLTS(L)
3:     G ← ESG(A)
4:     R ← M(L)  ▷ Stores the best result (process model) discovered so far
5:     while |V(G)| > |Activities(A)| do  ▷ While there is some duplicate task
6:         v_x, v_y ← FINDMOSTSIMILARNODES(G)
7:         merge v_x, v_y into single node in G
8:         L_i ← TAGLOG(L, G)  ▷ Tag events in the log according to current partition
9:         N_i ← M(L_i)  ▷ Discover a temporary model for evaluation
10:        if N_i is better than R then
11:            R ← N_i
12:    return R
```



Dup. tasks	Model size	Precision	
0	17	0.53	
1	24	0.60	
2	25	0.65	
3	**22**	**0.95**	←
4	23	0.86	
5	26	0.88	
6	33	0.88	
7	36	0.95	
8	38	0.95	

Figure 7.5: Dendogram showing clustering of LTS in Fig. 7.4a.

To select which partition of tasks will be returned by our procedure, we construct a temporary process model N_i at every iteration. The provided miner is called using a log where events have been tagged according to the currently evaluated partition. The details of how models are compared will be described in a later section. Note that the total maximum number of models to evaluate (i.e. the number of iterations in the procedure) is limited by the number of excitation sets in the LTS. However, most processes contain only a few duplicate tasks. Limiting to 4 or 5 tasks per activity reduces the number of models that need to be evaluated to a few, depending on the number of different activities.

Figure 7.5 visualizes the clustering procedure. The initial solution, where every LES is partitioned into its own duplicate task, is shown at the bottom

row. The following row represents one iteration of the clustering process, in which a_2, a_3 were the most similar LES and were merged. Thus, the number of duplicate tasks, in the first column, is reduced by 1. The top row shows the result after all nodes have been merged and thus there are no duplicate tasks left. Columns 2 and 3 show sample metrics of the evaluation model for each row: Petri net size and precision. The selected model has the best precision and smallest size.

Representing excitation set relations in vector space.

In order to find the closest two groups of LES, a distance metric capable of evaluating the similarity of the relationships of two LES is required. For this, we will first provide a way to represent, as a numeric vector, the causality relationships of a given vertex (representing a LES or cluster of LES) in a ESG.

This representation needs to satisfy several requirements: a) it needs to be normalized, allowing meaningful comparisons between different vertices, b) it needs to distinguish vertices by their immediate predecessors/successors, but also more distant neighbors. Otherwise, duplicate tasks sharing the same set of immediate predecessors or successors would not be distinguishable. However, similarity of closer neighbors should have more weight than distant neighbors.

Definition 7.3 (Context vector)**.** Given LTS A, $ESG(A)$, and a vertex $v \in ESG(A)$, the *forward context vector* of v, $\overrightarrow{C_v}$, is a function $E \mapsto \mathbb{R}$ that maps an activity $e \in \Sigma$ to

$$\overrightarrow{C_v}(e) = \frac{|\operatorname{Succ}(v,e)|}{2|\operatorname{Succ}(v)|} + \frac{\sum_{v' \in \operatorname{Succ}(v)} \overrightarrow{C_{v'}}(e)}{4|\operatorname{Succ}(v)|}$$

where $\operatorname{Succ}(v)$ is the set of immediate successors of v and $\operatorname{Succ}(v,e)$ is the set of immediate successors of v of activities with label e. Similarly, we can define the *backwards context vector*, $\overleftarrow{C_v}$, using predecessors instead of successors.

For a given vertex v and event e, the value of $\overrightarrow{C_v}(e)$ depends on the number of e-successors of v relative to the total number of successors of v. Notice the function gives decreasing weight to more distant successors using the pattern $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$. Thus, the function is normalized between $[0 \ldots 1)$, allowing for numeric comparisons between different vectors.

Imposing a limit k to the recursion depth, context vectors are easy to compute with a single pass over the graph. As the weight of successors decreases with distance, this limit does not impact the quality of the metric. An example list of context vectors for the graph in Fig. 7.4b is shown in Table 7.1, assuming $k = 2$.

Table 7.1: Context vectors for the ESG in Fig. 7.4b.

	Forward					Backward				
LES	a	b	c	d	e	a	b	c	d	e
a_1	0	$1/8$	$1/2$	$1/8$	0	0	0	0	0	0
a_2	0	0	0	0	0	0	$1/4$	0	$1/8 + 1/8$	$1/4$
a_3	0	0	0	0	0	0	$1/4$	0	$1/8 + 1/8$	$1/4$
b_1	0	0	$1/4$	0	$1/2$	0	0	0	0	0
b_2	$1/2$	0	0	0	0	$1/4$	0	$1/2$	0	0
b_3	$1/2$	0	0	0	0	0	0	$1/2$	0	$1/4$
c_1	$1/16 + 2/16$	$1/4$	0	$1/4$	$1/16$	$1/2$	0	0	0	0
c_2	$1/16 + 2/16$	$1/4$	0	$1/4$	$1/16$	0	$1/4$	0	0	$1/2$

Distance function.

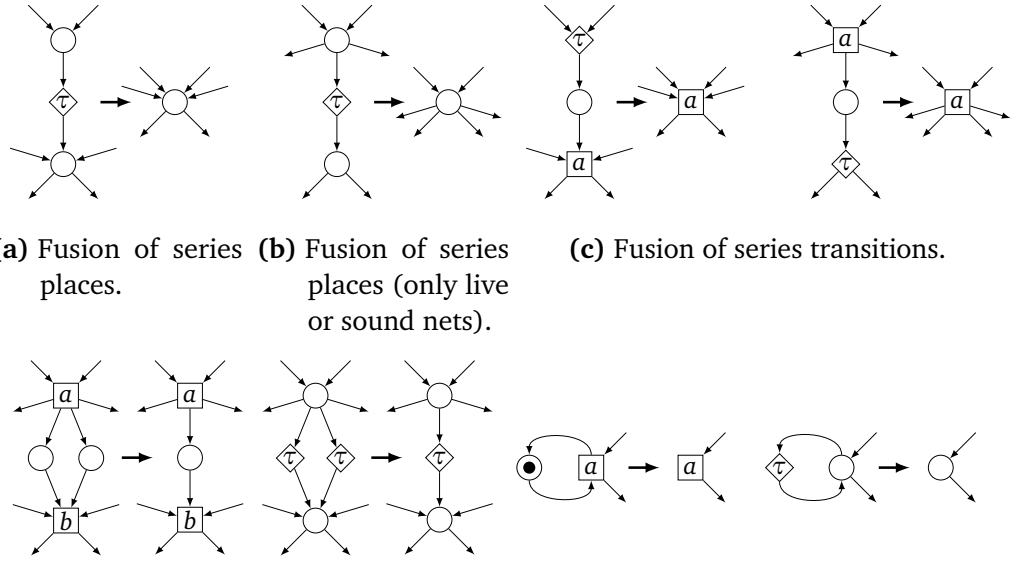
To measure the similarity (distance) between two vertices $v_1, v_2 \in ESG(A)$, the following formula is used, where d is the Euclidean distance:

$$\text{dist}(v_1, v_2) = \min(\text{d}(\overrightarrow{C_{v_1}}, \overrightarrow{C_{v_2}}), \text{d}(\overleftarrow{C_{v_1}}, \overleftarrow{C_{v_2}}))$$

Using the minimal distance between the forward and backward vectors allows proper detection of duplicate tasks in the first and last iterations of loops. For tasks in a loop, several LESs may exist in the LTS for different iterations of the same task. The causality relations of the LESs corresponding to the first and last iterations will be different of those from inner iterations. For example, only the LES corresponding to the last iteration will not trigger other LESs of the same task. By centering on either the backward or forward context vector, depending on which pair is the closest, these LESs will still be clustered into a single task.

Comparing candidate models.

Traditional hierarchical clustering algorithms use various criteria to determine which clustering solution is more suited to the data, such as for example the *elbow* criteria [83]. However, the flow proposed in this work produces more than one candidate model, allowing the exploration of the trade-off between precision and simplicity. By limiting the maximum number of allowed duplicate tasks, the set of candidate models can be kept under manageable sizes. Therefore, conventional conformance checking strategies may be used to accurately compare the candidate models, e.g. measuring fitness, precision, generalization or simplicity. Generally, a combination of these parameters will be used, depend-



(a) Fusion of series places. **(b)** Fusion of series places (only live or sound nets). **(c)** Fusion of series transitions.

(d) Parallel places. **(e)** Identical trans. **(f)** Identity place. **(g)** Identity trans.

Figure 7.6: Reduction rules for behavior-preserving removal of silent transitions.

ing on user preference. For example, maximizing precision while constraining the simplicity to a minimum threshold value.

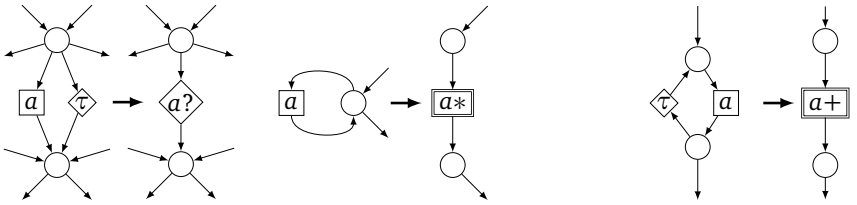
Figure 7.5 shows that the precision increases with every duplicate task until 95% with 3 duplicate tasks, and then decreases, revealing that more duplicate tasks introduce unnecessary choices and are not necessary for this process. The result, with 3 duplicate tasks, exactly matches the model shown in Fig. 7.1c.

7.5 Meta-transitions

This section introduces the structural simplifications proposed in this work: substituting common control flow patterns with special *meta-tasks* that represent optional or iterative behavior.

The simplifications are especially suitable for Petri nets. They reduce the complexity of the net while still allowing the full expressiveness of Petri nets. In addition, the proposed simplifications exactly preserve the semantics of the models, and thus, conformance metrics such as fitness and precision.

The simplifications center on two aspects. First, the removal of unnecessary silent transitions. While silent transitions are a useful construct, many mining algorithms or conversions from other modeling languages often generate silent



(a) Optional task. **(b)** Loop. **(c)** Loop with at least one iteration.

Figure 7.7: Rules for transformation using *meta-transitions*.

transitions that may be unnecessary [3]. Second, we introduce a series of *meta-transitions* which extend the language of Petri nets and represent simple flow control operations such as optional or iterative behavior.

Removal of silent transitions.

Our proposal removes unnecessary silent transitions by following the transformations shown in Fig. 7.6. The objective of these transformations is to eliminate as many silent transitions as possible without impacting the semantics of the Petri net, so that the set of traces fitting the original net is identical to the traces fitting the transformed Petri net. The transformations proposed are similar to the liveness and safeness-preserving transformations proposed in [112], that have been already used in previous work [3, 58] also with the goal of removing silent transitions. However, the existing set of transformations is not exhaustive. For example, it is not possible to remove all the silent transitions from the model in Fig. 7.1c using only the rules defined in [112].

By centering on a commonly used structural type of Petri nets, sound workflow nets [1, 6], we are able to introduce additional transformations covering the removal of more silent transitions. For example, Fig. 7.6b proposes that fusion of serial places can be performed even if the first place has other outgoing arcs. However, this transformation does not fully preserve the behavior of general Petri nets, as it may remove deadlocks present in the original Petri net. Full preservation of behavior, including liveness, is only guaranteed in the case of live Petri nets or nets with deadlocks only on specific states, such as sound workflow nets. For the former subtype of Petri nets, deadlocks only appear in states where the output sink place is marked [6], and the output place will never be modified by the transformation rule.

Meta-transitions.

A *meta-transition* replaces common a Petri net substructure (e.g., a self-loop) with a single transition that is defined to have identical behavior. By transforming a Petri net, replacing instances of these structures by meta-transitions, the element count of a Petri net can be reduced while completely preserving its behavior. The transformed net will fit exactly the same traces as the original net. In addition, the transformation may open the door to further simplifications such as removal of additional silent transitions.

In Figure 7.7 we show the proposed 3 new meta-transitions, as well as the behavior represented by each meta-transition. These specific patterns have been selected because of their high frequency in real-life processes. In addition, the well-known regular expression-like syntax used in the meta-transitions makes their meaning familiar.

The first meta-transition, $a?$, models an *optional* event: it is equivalent to a choice between the empty label τ and trace a. The other two meta-transitions represent iterative behavior. $a*$ is equivalent to a self-loop. Thus, it fits the empty trace, but also $\{a, aa, aaa, \ldots\}$. Meta-transition $a+$ similarly represents a loop of a, but requires at least one iteration.

7.6 Results

The algorithms described in this work have been implemented using PM-LAB [36]. To construct an LTS from the input log, the *multiset* abstraction from [5] is used. Our implementation of the clustering procedure uses the centroid linkage functionality of [83] to avoid recomputing context vectors on every iteration.

For a set of benchmarks, we compare the quality metrics of the models obtained with and without the proposed duplicate task discovery algorithm, as well as the reduction in complexity after the structural simplifications and use of meta-transitions. Our tool as well as all benchmarks are available at `http://www.cs.upc.edu/~jspedro/pnsimpl/`. In order to demonstrate the ability of our tool to work with multiple miners, two different miners will be used: Inductive Miner [101] (IM) and Petrify [35]. While the current version of the Inductive Miner does not support duplicate tasks, Petrify contains some support for automatic discovery of duplicate tasks [34]. Thus, models discovered by Petrify may already contain duplicate task before the clustering method proposed in this article takes place.

Precision and generalization are measured using the available ProM plugins [11, 30]. To measure complexity, we will show the size of the Petri nets. For

non-workflow Petri nets, such as those generated by Petrify, we will also use a complexity metric closely related to the concept of planarity: the minimal number of *crossings* required to embed the graph on a plane. This number is estimated using *dot* [66].

The method used to select a model from the list of candidates produced by the duplicate task discovery method depends on the miner used. When using the IM, the smallest model (in terms of places and transitions) out of all models with highest precision will be selected. When using Petrify, the model with lowest number of crossings, out of those with highest precision, is used instead.

7.6.1 Artificial benchmarks.

To evaluate our duplicate task discovery workflow and compare to the results presented by previous work, we reuse an existing dataset comprising a combination of small logs [74, 126, 133] whose source processes are well-known and reproduce behavior commonly found in real-life. Because these benchmarks have no noise, the miners were configured to generate perfectly fitting models.

Table 7.2 summarizes the results. For every benchmark, there are three different runs: in the first one, the log is mined with the default miner configuration. In the second run, the flow with duplicate task discovery as presented in this work is used. In the third result, we apply structural simplifications (silent transition elimination and meta-transitions) on top of the model discovered on the second run. For each run, we evaluate the size of the model (number of places, transitions and silent (τ) transitions) as well as its precision and generalization.

The proposed method significantly increases the precision on all the benchmarks. In some examples, generalization is reduced, yet still shows that the method results in models that are not overfitting. In tests with the Inductive Miner, using duplicate tasks allows removing most of the silent transitions, and thus the overall complexity of the model decreases. Using meta-transitions, additional silent tasks can be removed. On the other hand, when combining our discovery flow with Petrify, the discovery of duplicate tasks allows for models with fewer crossings. However, results after simplification are not as remarkable as with the IM, since Petrify does not discover silent transitions.

For the majority of benchmarks, the partition of tasks discovered by the proposed flow exactly matched the duplicate tasks in the original process. The exceptions are marked with †. These cases are usually situations where, e.g., duplicate tasks are concurrent with themselves. Despite the fact that the partition is not exactly correct, the increase in quality metrics is still significant.

	Inductive Miner					With duplicate tasks						After simpl.		
	$\|P\|$	$\|T\|$	$\|\tau\|$	Prec.	Gen.	$\|P\|$	$\|T\|$	$\|\tau\|$	Prec.	Gen.		$\|P\|$	$\|T\|$	$\|\tau\|$
alpha	11	17	6	68%	100%	11	16	4	70%	100%	†	9	13	1
betaSimpl	14	21	8	62%	86%	14	16	1	94%	73%		14	15	0
Fig5p19	9	14	6	67%	89%	12	14	5	85%	76%		12	12	3
Fig5p1AND	9	8	3	83%	28%	10	8	2	100%	0%		9	7	1
Fig5p1OR	5	6	1	70%	33%	6	6	0	100%	0%		6	6	0
Fig6p10	15	24	13	63%	100%	19	25	10	77%	100%		18	19	4
Fig6p25	22	35	14	76%	100%	24	35	12	84%	100%		23	27	4
Fig6p31	6	10	1	63%	72%	9	11	0	100%	42%		9	11	0
Fig6p33	7	11	1	67%	70%	10	12	0	100%	38%		10	12	0
Fig6p34	17	24	12	58%	100%	19	20	4	93%	100%		17	18	2
Fig6p38	13	11	4	62%	84%	12	14	6	66%	87%		11	11	3
Fig6p39	12	12	5	90%	94%	12	12	5	90%	94%	†	10	9	2
Fig6p42	7	18	4	23%	100%	26	32	12	75%	96%	†	24	29	9
Fig6p9	10	15	8	67%	82%	9	12	3	83%	72%		9	9	0
flightCar	10	14	4	67%	64%	10	14	4	67%	64%	†	11	9	1
RelProc	21	28	12	71%	100%	21	28	11	74%	100%	†	19	21	4

	Petrify					With duplicate tasks						After simpl.		
	$\|P\|$	$\|T\|$	Cros.	Prec.	Gen.	$\|P\|$	$\|T\|$	Cros.	Prec.	Gen.		$\|P\|$	$\|T\|$	Cros.
alpha	13	11	11	92%	100%	12	12	1	92%	100%	†	12	12	1
betaSimpl	11	13	1	80%	77%	14	15	0	97%	39%		15	15	0
Fig5p19	8	8	2	100%	74%	9	9	1	100%	58%		9	9	1
Fig5p1AND	8	5	0	100%	0%	7	6	0	100%	0%		7	6	0
Fig5p1OR	5	5	3	100%	0%	5	6	0	100%	0%		5	6	0
Fig6p10	7	11	1	39%	100%	13	15	1	91%	100%		13	15	1
Fig6p25	14	21	6	80%	100%	14	23	0	80%	100%		18	23	0
Fig6p31	7	9	12	100%	42%	8	11	0	100%	42%		8	11	0
Fig6p33	8	10	7	100%	38%	9	12	0	100%	38%		9	12	0
Fig6p34	9	12	4	39%	100%	14	16	0	89%	100%		14	16	0
Fig6p38	8	7	0	71%	85%	10	8	0	100%	64%		10	8	0
Fig6p39	6	7	0	72%	98%	7	8	1	86%	86%	†	8	8	0
Fig6p42	11	14	20	37%	98%	21	23	3	96%	94%	†	21	23	3
Fig6p9	9	7	9	100%	54%	8	9	0	100%	54%		8	9	0
flightCar	6	8	0	58%	72%	6	8	0	58%	72%	†	7	8	0
RelProc	16	16	11	87%	100%	15	17	2	87%	100%	†	15	17	2

Table 7.2: Comparison using artificial benchmarks.

7.6.2 Logs with noise.

An additional experiment shows the resilience of the proposed method to noise. We used Process Log Generator (PLG) [32] to generate a set of 3 random processes using a process depth of 3 and uniform probabilities for all control flow operators. Then, for each of these processes, we generated 10 logs containing 1000 traces each. In each log a different amount of random control-flow noise was injected using PLG, ranging from 0% to 10%.

Figure 7.8 compares the precision of the models obtained using the Inductive Miner – infrequent [101] (IMi) miner, configured with a 20% threshold, with the models obtained by the combination of our duplicate task discovery flow and the IMi. For the 3 evaluated processes, our flow can discover duplicate tasks and thus increase the precision even when confronted with noise. The differences in fitness were always smaller than 5% between both versions.

On a Intel Core i5-2520m, our implementation of the clustering procedure is able to provide a set of candidate partitions in less than 4 seconds, even for the largest of these logs. The runtime of the miner, required to evaluate each candidate, is usually much larger than the clustering process. However, the number of candidates to be evaluated can be limited by setting an upper bound to the number of allowed duplicate tasks per label.

7.6.3 Publications

The methods proposed in this chapter have been accepted in a conference:

- J. de San Pedro and J. Cortadella, *Discovering duplicate tasks in transition systems for the simplification of process models*, in Business Process Management (BPM), Rio de Janeiro, Brazil, September 2016.

7.7 Conclusions

This chapter has presented methods for simplification of process models that improve the quality of discovered models, in both simplicity and precision, while using different mining algorithms.

As future work, we envision methods that work even in the presence of concurrent duplicate tasks, which are currently handled with unsatisfactory results. In addition, the language of structural tasks can be extended, for example, to allow simple regular expressions in nodes, e.g., $(a|bc)*$.

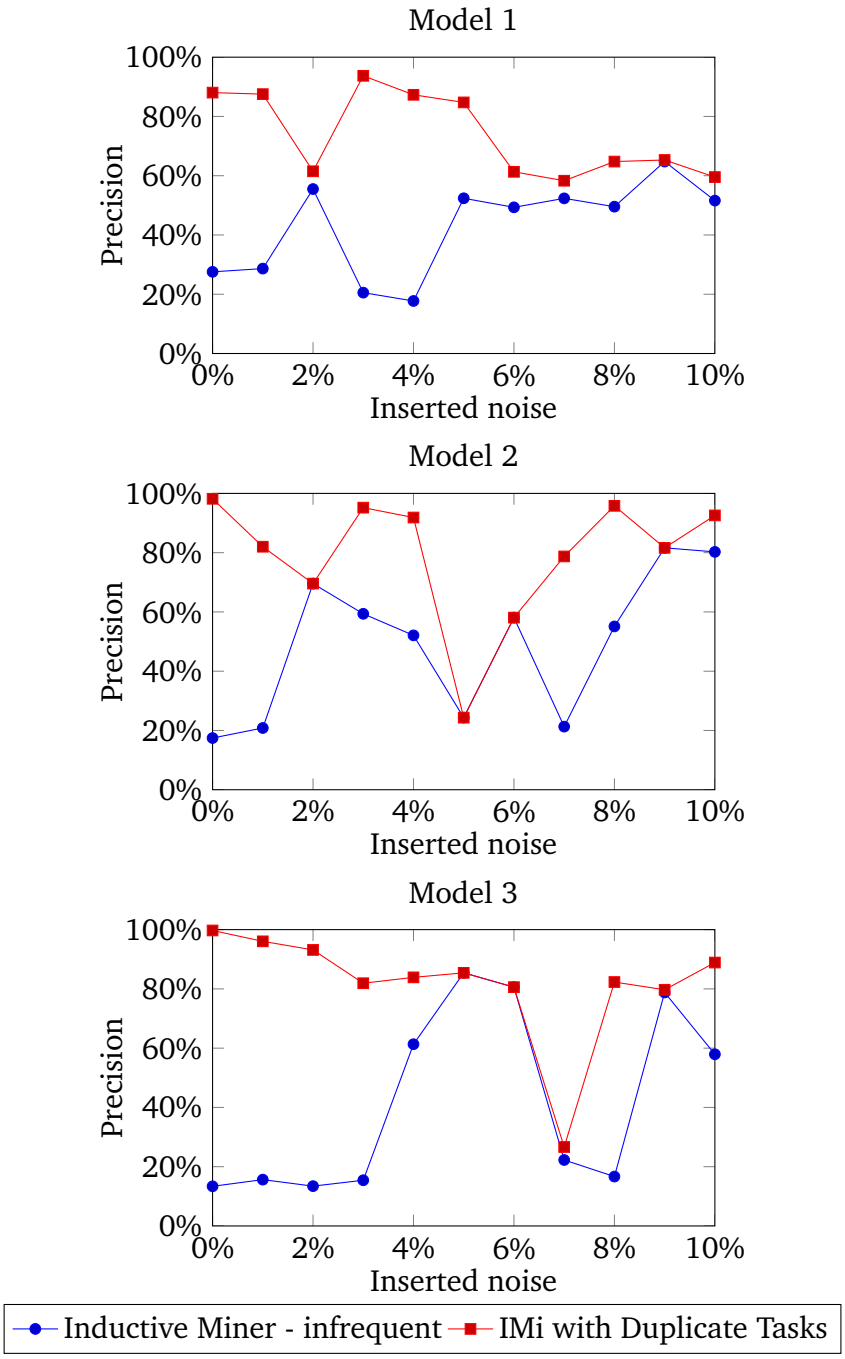


Figure 7.8: Resilience of duplicate task discovery to different artificial noise levels.

Chapter 8

Specification mining of asynchronous controllers

The chapter presents a first effort at exploring a novel area in the domain of asynchronous controllers: specification mining. Rather than synthesizing circuits from specifications, we aim at doing reverse engineering, i.e., discovering safe specifications from the circuits that preserve a set of pre-defined behavioral properties (e.g., hazard freeness). The specifications are discovered without any previous knowledge of the behavior of the circuit environment.

This area may open new opportunities for re-synthesis and verification of asynchronous controllers. For example, the specifications obtained by the proposed flow may be used to decompose the behavior of a controller under test for more efficient verification using formal approaches.

The effectiveness of the proposed approach is demonstrated by mining concurrent specifications (Signal Transition Graphs, STG) from multiple implementations of 4-phase handshake controllers, and a selection of controllers with choice.

This chapter is structured as follows. Section 8.1 provides a brief overview of the topic and a visual example to motivate it. In Section 8.2, we perform a review of the related literature. Section 8.3 gives an initial foray into the behavioral properties that are preserved in the discovered specifications. The specification mining algorithm itself is detailed in Section 8.4. In Section 8.5, we introduce an extension that allows the mining flow to also consider properties specific to the specification language used, and detail the specific constraints for different structural subtypes of STGs. Finally, Section 8.6 shows the results of applying the proposed mining flow to a set of testcases, while in Section 8.7 we discuss future work and conclusions of this topic.

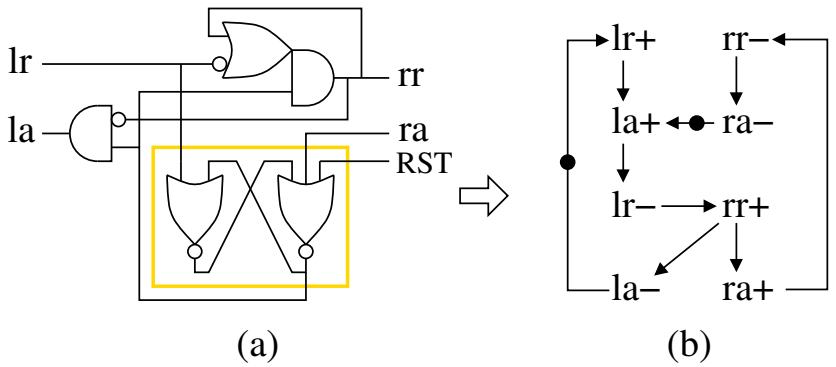


Figure 8.1: Specification mining of a handshake controller.

8.1 Motivation

The design automation efforts in the area of asynchronous circuits have been mostly focused on two problems: synthesis and formal verification. The synthesis problem consists of obtaining a circuit from a specification, e.g., a gate netlist from a Signal Transition Graph (STG). The verification problem consists of checking the conformance of a circuit with regard to a specification.

In this chapter we study a new problem for asynchronous circuits: *Specification Mining*. The problem consists of discovering formal specifications from implementations [13], without any previous knowledge of their original specifications.

The problem can be illustrated using the example in Fig. 8.1(a). The circuit implements a handshake controller in which only the initial state is known (all handshake signals at 0, RST=1). The reset signal (RST) is assumed to be silent during the normal operation of the circuit and the cross-coupled NOR gates are assumed to have an atomic behavior (negligible internal delay). We pose the following challenge:

> *Can we discover a specification of the interface that guarantees a speed-independent behavior of the circuit?*

The answer to this question is not unique. Several interfaces could exercise the circuit without producing any hazard. In particular, an empty interface (no events) would guarantee such behavior. Our interest is to find maximally concurrent interfaces that honor the desired properties of the circuit.

Fig. 8.1(b) shows one possible safe interface. This interface has been discovered automatically by the approach proposed in this work and coincides with the L440R2044 4-phase controller (according to the nomenclature in [25]).

Specification mining can define not only properties that must be preserved in the circuit, but also properties of the interface. For example, it is possible to enforce that the interface is choice-free, i.e., no conflicts in the environment.

Specification mining opens a new research direction in the area of asynchronous controllers that could potentially have applications in different domains, e.g.,

- Reverse engineering, to discover the behavior of some intricate controllers for which no specification is known.
- Re-synthesis of asynchronous controllers, since the discovered interfaces can be used as specifications for synthesis tools that can produce higher-quality solutions and substitute the existing ones [94].
- Compositional verification, by substituting some components of a large circuit by the mined specifications. In this way, an assume/guarantee scheme could be applied to verify the circuit by using the mined interfaces while hiding the internal signals of the components [40, 118].

The main goal of this chapter is to demonstrate that specification mining is feasible for a variety of controllers. The application of this paradigm to specific problems in asynchronous design and verification is out of the scope of this work.

8.1.1 Example

This section gives an informal overview of the approach proposed for specification mining, using the example shown in Fig 8.2. The goal is to obtain a specification for the environment of the circuit shown in the Fig. 8.2(a) in such a way that certain behavioral properties are guaranteed.

In this particular case, we would like the circuit to be speed-independent (SI) and have a delay-insensitive (DI) interface. Speed-independence is guaranteed when the circuit is *output persistent* (only input signals can disable each other). A circuit has a DI interface if its behavior does not depend on the arrival order of the inputs.

The labeled transition system (LTS) shown in Fig. 8.2(b) shows all possible behaviors of the circuit under a free environment, i.e., all input signals can switch at any time instant. Every state corresponds to a binary vector that represents the value of the signals at that state. We will identify each state by its binary vector $\langle abxy \rangle$.

The labels x^+ and x^- represent rising and falling transitions of signal x. The double arcs $\overset{a^*}{\longleftrightarrow}$ represent alternating a^+ and a^- transitions between a

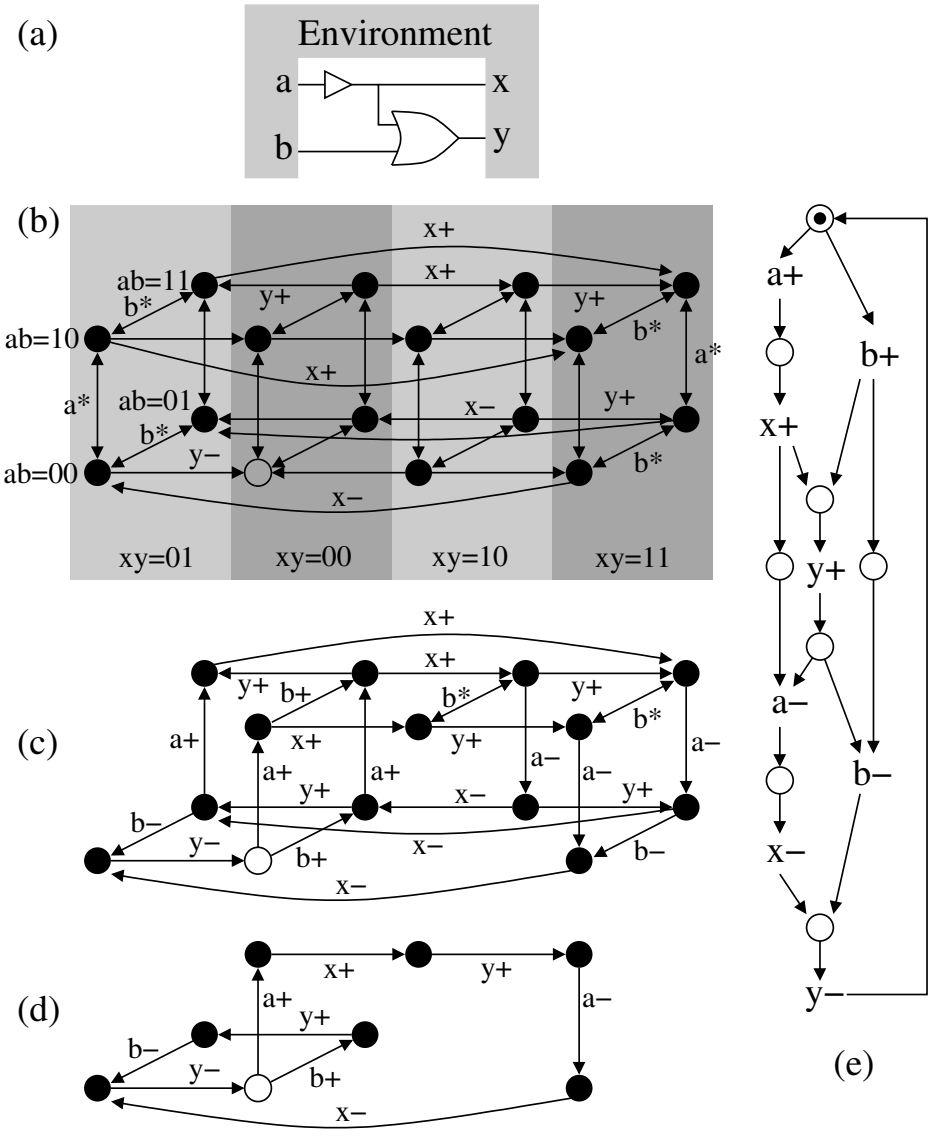


Figure 8.2: Simple circuit for specification mining.

pair of states and are used to model the switching of input signals in a free environment. The transitions of x and y are depicted in the horizontal direction, whereas the transitions of a and b are depicted in the vertical and diagonal directions, respectively. For the sake of clarity, not all the labels are shown in the picture, although they can be easily deduced from the depicted information.

We will assume that the initial (reset) state is also known. In the example, the initial state is $\langle 0000 \rangle$ (unfilled circle).

A free environment leads to many circuit malfunctions (hazards). For example, transition $\langle 0010 \rangle \xrightarrow{x^-} \langle 0000 \rangle$ produces a violation of output persistence that may be manifested as a glitch in signal y, since y^+ is enabled in $\langle 0010 \rangle$ and disabled in $\langle 0000 \rangle$.

The goal of specification mining is to *discover* one or several specifications for the environment that:

- have good properties, e.g., guarantee a hazard-free behavior of the circuit, and
- are general enough to cover a large set of behaviors.

As an example, the cyclic behavior $(b^+y^+b^-y^-)^*$ is hazard free. However, we might be unsatisfied by the fact that signals a and x are not exercised.

Fig. 8.2(c) depicts an LTS with output persistence, i.e., no output transition can be disabled by another transition. The largest LTS fulfilling this property can be uniquely obtained from the one in Fig. 8.2(b) by deleting the transitions that produce violations of output persistence.

Still, Fig. 8.2(c) does not model a delay-insensitive (DI) interface (see Section 2.4.1 for a formal definition). For example, the transitions a^- and b^+ are enabled in state $\langle 1011 \rangle$. The arrival of a^- (leading to $\langle 0011 \rangle$) disables b^+, whereas the arrival of b^+ (leading to $\langle 1111 \rangle$) does not disable a^-.

Unfortunately, there is no unique solution when trying to find a subset of the LTS that fulfills the DI interfacing conditions. Given the fact that the circuit cannot be modified, the only chances are reduced to constraining the environment by deleting some input transitions. In the previous example, a DI interface can be obtained in different ways, e.g., by removing either transition a^- or b^+ from state $\langle 1011 \rangle$. Other deletions are also required in other parts of the LTS to guarantee a complete DI interface.

Fig. 8.2(d) depicts an LTS that models an SI circuit with a DI interface. An STG modeling the same behavior is shown in Fig. 8.2(e). Obtaining this specification is the most challenging problem tackled in this chapter. We will show how the problem can be modeled with a Boolean formula and solved using SAT or Integer Linear Programming.

In the rest of the chapter we propose the methodology used to obtain specifications from a circuit that fulfill a set of properties defined a priori.

8.2 Related work

In this chapter we tackle a novel area, the specification mining [13] of asynchronous controllers.

Specification mining is becoming popular in the software engineering community as a machine-learning approach to infer properties from the observable behavior of the systems [96]. One example is the work presented in [73] in which safe and permissive interfaces (sequences of library calls) are synthesized for software systems in such a way that the interfaces do not violate the internal invariants of the system. Another example is [98] where specifications represented as Message Sequence Graphs are synthesized from the traces observed from the execution of a concurrent system.

In [104], a method is presented which automatically infers high-level descriptions from circuits, representing them as a combination of instances of abstract functional blocks from a predetermined library.

8.3 Circuits with constrained environment

The main purpose of specification mining is to find a specification of the environment of the circuit that fulfills certain properties. In other words, finding a constrained environment that prevents the circuit from reaching states in which the desired properties are violated. This section formalizes the concepts of the LTS of a circuit under a free/constrained environment, and then introduces some examples of desirable behavioral properties in a LTS. For a formal definition of an LTS, we refer to Section 2.2.2.

Definition 8.1 (LTS under free environment)**.** Given a circuit $C = \langle X, G, s_0 \rangle$, we define $\text{LTS}(C) = \langle S, \Sigma, T, s_0 \rangle$ as the LTS associated to C and generated by a free environment. Formally:

- $S = \{0,1\}^n$, with $n = |X|$, the set of binary vectors representing all possible states of the signals.
- $\Sigma = X \times \{+,-\}$, where x^+ and x^- distinguishes the rising and falling transitions, respectively, of signal $x \in X$.
- $T = T_{env} \cup T_g$, where T_{env} and T_g are the transitions produced by the environment and the circuit, respectively (defined later).

The set of transitions T_{env} in a free-environment circuit is defined as follows:

$$T_{env} = \{s \xrightarrow{x} s^{\neg x} \mid s \in S \wedge x \in I\},$$

representing the fact that any input signal can switch at any state. The set of transitions T_g produced by the circuit is defined as follows:

$$T_g = \{s \xrightarrow{x} s^{\neg x} \mid s \in S \wedge x \in (O \cup Z) \wedge s(x) \neq f_x(s)\},$$

representing all transitions of non-input signals produced by the logic gates when the value at the output of the gate is different from the function computed by the gate.

The goal of specification mining, to discover a constrained environment E that satisfies all desired properties, is equivalent to finding a subset of transitions of T_{env} that prevent the circuit from reaching states in which the desired properties are violated.

Definition 8.2 (LTS under constrained environment)**.** Given a circuit C, its free-environment $\text{LTS}(C) = (S, X, T, s_0)$ and a subset of transitions $E \subseteq T_{env}$ representing a constrained environment, we denote by $\text{LTS}(C, E)$ the LTS obtained from $\text{LTS}(C)$ after deleting the transitions in $T_{env} \setminus E$. Formally, $\text{LTS}(C, E) = (S', X, T', s_0)$ is the maximal LTS such that:

- $S' \subseteq S$ is the subset of reachable states from s_0.
- T' is the maximal set of reachable transitions from s_0 such that $T' \subseteq (E \cup T_g)$.

$\text{LTS}(C, E)$ can be computed from $\text{LTS}(C)$ by deleting the transitions in $T_{env} \setminus E$ and iteratively deleting unreachable states and transitions until a greatest fixed point is reached.

8.3.1 Properties of an LTS

Let $\text{LTS}(C, E) = \langle S, \Sigma, T, s_0 \rangle$ be the LTS associated to circuit $C = \langle X, G, s_0 \rangle$ with environment E. We say that x is *enabled* in state s if $s \xrightarrow{x} s^{\neg x} \in T$. We say that x *disables* y in state s if $s \xrightarrow{x} s^{\neg x} \in T$, y is enabled in s and not enabled in $s^{\neg x}$. Finally, we say that x *triggers* y in state s if $s^{\neg x} \xrightarrow{x} s \in T$, y is not enabled in $s^{\neg x}$ and enabled in s. For more details on these definitions we refer to Section 2.2.2 and Section 2.4.

Definition 8.3. In an LTS a signal x is *persistent* if no signal $y \neq x$ disables it. If a signal x disables another signal y in any state s, then there is a *conflict* between x and y.

Example. Let us consider the LTS in Fig. 8.2(b), where the initial state is $s_0 = \langle 0000 \rangle$ (unfilled circle). Let us consider the state $s_1 = \langle 1000 \rangle$ and the transition $s_0 \xrightarrow{a^+} s_1$. We can say that a^+ triggers x^+ in s_1, since x^+ is not enabled in s_0 but it is in s_1. Let us now consider the transition $s_1 \xrightarrow{a^-} s_0$. We can see that a^- disables x^+ in s_1 since x^+ is enabled in s_1 but not in s_0. Notice how Fig. 8.2(c) and (d) show LTSs where both x and y are persistent signals while a, b are in conflict.

Speed-independence

In this work, we deal with circuits with unbounded gate delays, i.e., any gate of the circuit can switch at any time as long as it is enabled. A circuit whose behavior does not depend on the delay of its gates is called *speed-independent*.

Proposition [46, 89] Given a circuit C and an environment E, C under E is speed-independent iif in LTS(C, E) all pairs of signals x, y are in conflict only if both x, y are input signals.

This property implies every non-input signal is persistent.

Delay insensitive interfacing

Another desired property is that the behavior of the circuit is insensitive to the arrival order of the input transitions. This property is called *delay-insensitive (DI) interfacing* and is formally defined as follows.

Proposition [121]. The LTS associated to a circuit satisfies the DI interfacing conditions if no input transition triggers another input transition.

Rather than dealing with pure delay insensitivity, DI interfacing assumes that wire delays can be kept under control within the circuit and only tolerance to delay variability at the interface is required.

Multi-environment interfaces

In some scenarios for re-synthesis and compositional verification, a system may have been split into different components (circuits). From the point of view of the circuit of interest, the surrounding components can be considered as a set of independent environments, $E_1 \ldots E_n$, that interact with the circuit, as shown in the example of Fig. 8.3.

When mining specifications for a circuit, we might want to consider multi-environmental scenarios where independence between different environments is to be preserved. Informally, this means that, given two environments E_i and E_j, signals from E_i cannot directly trigger or disable inputs from E_j. Such a

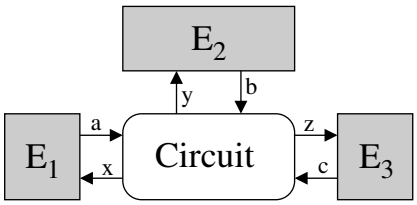


Figure 8.3: Multi-environment interfacing.

x	y	x triggers y	x disables y
Input	Input	Violates DI	*Only if* $x, y \in E_i$
Output	Output	Allowed	Violates SI
Input	Output	Allowed	Violates SI
Output	Input	*Only if* $x, y \in E_i$	Violates SI

Table 8.1: Allowed relations in a circuit with multiple environments.

causality relation would imply a connection between E_i, E_j outside of the circuit of interest, making E_i and E_j dependent from each other. However, an input from E_i may excite an output in a different environment E_j, via the circuit.

Definition 8.4. Let us consider the LTS(C) associated to a circuit C. Let the set of signals of C be

$$X = X_1 \cup \ldots \cup X_n \cup Z$$

where Z is the set of internal signals and $\{X_1, \ldots, X_n\}$ is a partition of the set of input/output signals ($I \cup O$), with each X_i corresponding to a different environment. The LTS preserves the *multi-environment interface* for partition $\{X_1, \ldots, X_n\}$ if:

$$\forall a \in X_i, b \in X_j \cap I,\ i \neq j : a \text{ cannot trigger or disable } b.$$

In the example of Fig. 8.3, the preservation of the multi-environment interface would not allow $\{x, a\}$ to trigger or disable $\{b, c\}$, $\{y, b\}$ to trigger/disable $\{a, c\}$ and $\{z, c\}$ to trigger/disable $\{a, b\}$.

Note that, by this definition, a circuit where each environment has one input only, i.e. $\forall E_i$, $|E_i \cap I| \leq 1$, any LTS preserving the multi-environment interface would be input-persistent, as no signal would be allowed to disable an input.

Table 8.1 summarizes the previous properties, showing the allowed causality relations between two different signals x, y depending on the type (input or output) of each signal. For example, x cannot trigger y if both x, y are inputs, since that would violate the delay-insensitive interfacing property. However, x may disable y, but only if both x, y are in the same environment E_i.

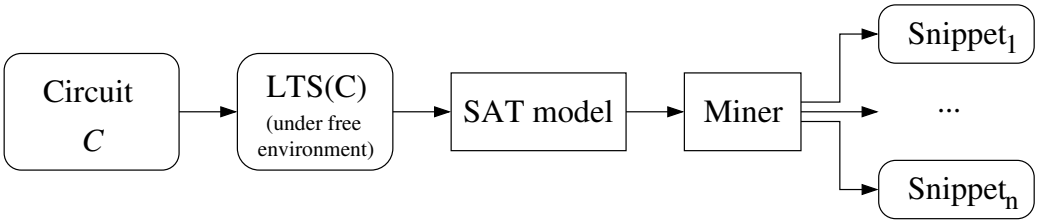


Figure 8.4: Overview of the specification mining flow.

8.4 Specification mining

This section describes the main contribution in this chapter: a process used to mine a specification from a circuit C, while guaranteeing a set of properties for both the circuit and the interface. The process works by starting from a free environment E, and then constraining this environment until both the environment and the circuit under such environment satisfy all properties.

Note, however, that for certain properties there may be more than one specification satisfying all the properties. The environments in each specification may only exercise a small subset of the full circuit behavior. In these situations, our flow will discover each of these specifications, which we call *snippets*. The original specification of the circuit will be contained in one of these snippets. Our mining flow will give priority to the most general snippet, containing the environment exercising most behavior from the circuit. An example of this will be discussed in Section 8.6.1.

A summary of the proposed mining flow can be seen in Fig. 8.4. The first step constructs LTS(C) containing all the behaviors of the circuit under a free environment, starting from the circuit netlist. For the circuit in Fig. 8.2, this would correspond to the LTS in (b).

The desired properties for the mined specifications are then specified into a set of constraints on top of this LTS. In this section, we will specify these constraints as satisfiability (SAT) formulae. Every truth assignment of the formula represents a valid environment under which circuit C satisfies all the desired properties, and from which a snippet may be synthesized.

The most interesting snippets can be then synthesized into different types of specifications, such as Signal Transition Graphs (STGs). For most circuits, only the snippet containing the most general behavior will be of interest. However, the secondary snippets may provide an insight into alternative behaviors of the system.

The following sections describe this flow in more detail, including examples that show how the most common circuit properties are modeled.

8.4.1 Satisfiability model for behavioral properties

Many of the most interesting circuit properties imply constraints on the causality/concurrency/choice relations between events of the LTS. Table 8.1 shows the causality constraints to guarantee speed independence, delay insensitive interface and multi-environment properties.

In this section, we show how different circuit and environment properties can be mapped into constraints between different signals, and how these constraints can be implemented on a SAT model.

Let $\mathrm{LTS}(C) = \langle S, \Sigma, T, s_0 \rangle$ be the LTS associated to a circuit C constructed using the method defined in the previous section. The SAT model extracts a subset of $\mathrm{LTS}(C)$, $\mathrm{LTS}(C, E)$, satisfying the required properties. In our formulation, for every transition $t_i \in T$, we define a variable with the same name indicating whether t_i is selected, i.e. whether $t_i \in \mathrm{LTS}(C, E)$. Definitions of the most typical constraints are as follows:

x cannot trigger y

For every pair of states $s_1, s_2 \in S$, with a transition $t_2 = s_1 \xrightarrow{x} s_2$, x triggers y if y is enabled in s_2 but not in s_1. i.e., when $t_3 = s_2 \xrightarrow{y}$ exists but $t_1 = s_1 \xrightarrow{y}$ does not. This pattern is illustrated in Fig. 8.5(b).

To guarantee this property, if both t_2 and t_3 are selected, then t_1 must exist and be selected:

$$t_2 \wedge t_3 \implies t_1.$$

In case t_1 does not exist in $\mathrm{LTS}(C)$, then the previous constraint must be rewritten accordingly, forbidding selection of both t_2 and t_3:

$$\neg(t_2 \wedge t_3).$$

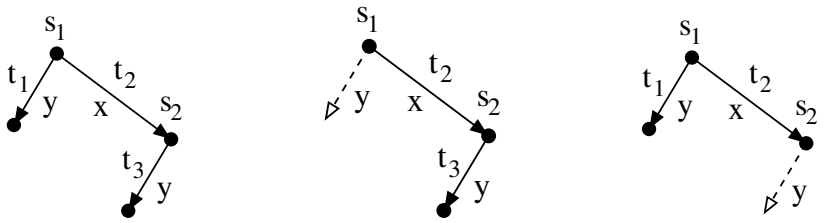
These constraints may be used for example to enforce the delay-insensitive interfacing property when applied to pairs of inputs signals, as shown in Table 8.1.

x cannot disable y (persistence of y)

For every pair of states $s_1, s_2 \in S$, with a transition $t_2 = s_1 \xrightarrow{x} s_2$, x disables y if y is enabled in s_1 but not in s_2. This is similar to the trigger definition above, except that the roles of t_1 and t_3 are reversed: x disables y when $t_1 = s_1 \xrightarrow{y}$ exists but $t_3 = s_2 \xrightarrow{y}$ does not, as seen in Fig. 8.5(c).

Thus, to satisfy the property, selecting t_1 and t_2 implies t_3 must exist and be selected:

$$t_1 \wedge t_2 \implies t_3.$$



(a) Persistence of y. **(b)** x triggers y. **(c)** x disables y.

Figure 8.5: Causality patterns in LTS.

Similar to the previous constraint, this constraint can be simplified if any of t_1, t_2, t_3 is not present in LTS(C).

Preservation of non-input signals

The SAT model searches for a subset LTS(C, E) representing the behavior of the circuit under a constrained environment E. For this reason, the model should only remove transitions of input signals, and potentially all transitions from unreachable states under environment E.

However, it must always select all non-input transitions from an state if it is reachable. Thus, for every state s, the following constraint must be added, which forces the selection of all non-input transitions if any incoming transition is selected:

$$\forall s \in S: \quad \bigvee_{t_i = s_1 \xrightarrow{x} s \in T} t_i \implies \bigwedge_{\substack{t_j = s \xrightarrow{y} s_2 \in T \\ y \notin I}} t_j.$$

Strong connectedness

This property ensures that the initial state is reachable from every other reachable state. There are several known methods to require conectedness with SAT or ILP models [41]. To identify the initial state, we assume that the values of the output signals are known at reset time, and that it is stable, i.e. no output transitions are enabled.

Additional constraints

Many controllers impose additional properties on the environment that can be modeled as constraints between different inputs. For example, in Section 8.6.2 we show a circuit with a mutual exclusion requirement between two different input signals, i.e. the two signals cannot be enabled simultaneously.

These properties can be enforced by removing states from LTS(C). For example, guaranteeing mutual exclusion between two input signals x, y is equivalent to removing every state s where $s(x) \vee s(y)$. In the proposed formulation, this can be achieved by prohibiting the selection of any t_i incident to s.

8.4.2 Algorithm for specification mining

Algorithm 11 Extracting LTS snippets

```
1: Input: Circuit C and a set of desirable properties P
2: Output: LTS_1, ..., LTS_n under which C satisfies P
3: L ← LTS(C)  ▷ construct full LTS from C
4: R ← T(LTS(C))  ▷ transitions not yet in any snippet
5: i ← 1
6: while |R| > 0 do
7:     LTS_i ← SOLVE(LTS(C), P, maximize |t_i ∈ R|)
8:      ▷ extract subset of LTS(C) satisfying P
9:     if LTS_i = ∅ break
10:    R ← R \ T(LTS_i)  ▷ subtract from remaining transitions
11:    i ← i + 1
12: return LTS_1, ..., LTS_n
```

Algorithm 11 describes the procedure to mine specifications using the SAT model described in the previous section. At the start of the procedure, the complete LTS(C) is built. The algorithm iterates, generating a new snippet on each cycle, until all transitions from LTS(C) appear on at least one snippet or it is impossible to create new ones without violating P. To account for the former, R contains all transitions not yet included in any $\text{LTS}_1, \ldots, \text{LTS}_i$.

Procedure SOLVE uses the SAT model to find the subset of LTS(C) satisfying P that contains the largest subset of transitions from R. Thus, every iteration discovers a snippet containing the largest behavior from C not yet covered in any previous snippet. Different strategies may be used to solve the SAT model with the cost function, such as MaxSAT or ILP.

8.5 Properties of the specification model

In the previous section we have shown a method to mine snippets in which both the circuit and the environment satisfy a set of properties. These snippets are provided in the form of LTSs. However, it is often desirable to use more succinct representations, such as Signal Transition Graphs (STGs). An STG may be obtained from an LTS using Petri net synthesis tools such as petrify [45].

This section shows that, by adding some constraints during the mining process, properties of the specification model can be enforced. For example, structural Petri net properties, such as marked graphs or free-choiceness, can be modeled in this way. These extra properties may contribute to enhance the visualization and analysis of the specification models.

This section is focused on structural properties of Petri nets, generating two different types of STGs: marked graphs and free choice.

Marked Graphs

A marked graph is a Petri net in which all places have exactly one predecessor and one successor transition [112].

Forward and backwards persistence are necessary conditions for a strongly-connected LTS to model the space state of a marked graph [21]. Thus, to obtain a marked graph, it is necessary to extend the constraints of the mining flow to prevent conflicts between all pairs of signals. Backwards persistence can be guaranteed using a similar set of constraints.

Free-choiceness

A Petri net is free choice is for any two transitions x and y that share a predecessor place p, then x and y have only one predecessor [112]. While there is a choice in p between x and y, we say the choice is *free* because on any marking where x can be fired, y can be fired too, and vice versa.

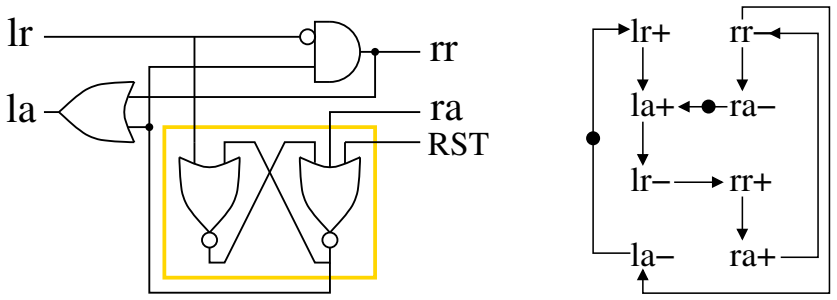
In a LTS, this is equivalent to guaranteeing that if x, y are in conflict, then x must be enabled in all the states where y is enabled, and y must be enabled in all states where x is. We model this by introducing a new Boolean variable, $\text{choice}_{x,y}$, which indicates whether the snippet contains a conflict between x and y, and a new set of constraints which relate these variables to the relationship between x and y.

The first set of constraints removes all conflicts between x, y (identical to the constraint described in section 8.4.1), unless $\text{choice}_{x,y}$ is asserted:

$$(t_1 \wedge t_2 \implies t_3) \vee \text{choice}_{x,y}.$$

In addition, for every state s where either x or y is enabled, a constraint is added forcing both to be enabled or disabled if $\text{choice}_{x,y}$ is asserted. With $t_1 = s \xrightarrow{x} s_1$ and $t_2 = s \xrightarrow{y} s_2$:

$$\text{choice}_{x,y} \implies t_1 = t_2.$$



(a) Circuit **(b)** Mined specification

Figure 8.6: Example of circuit and mined specification for *L440oR2264*.

As in previous constraints, the formula is simplified appropriately if there is no t_1 or t_2 in s. For example, if there is a state s where t_1 exists but t_2 does not, the constraint becomes:

$$\text{choice}_{x,y} \implies \neg t_1.$$

8.6 Results

In this section we demonstrate the effectiveness of the proposed flow with a series of selected benchmarks.

Configuring our proposed flow to search for environments where the circuit is speed-independent and with delay-insensitive interfacing, we were able to discover the correct specifications for the basic asynchronous building blocks: C-element, handshake decouplers (B and D-element [33]), etc.

Section 8.6.1 centers on one of these basic blocks: the 4-phase latch controller. In 8.6.2, designs with free choices in the environment are also included.

8.6.1 Mining specifications for 4-phase latch controllers

The 4-phase latch controller is at the core of the data paths of many asynchronous designs. A 4-phase latch controller is composed of 4 handshake signals controlling 2 channels: left (*lr*, *la*) and right (*rr*, *ra*), as shown in the example of Fig. 8.1.

In [25], the design space of 4-phase controllers is studied. While these designs all have the same external interface, they vary on the level of concurrency allowed by the protocol. Each variation cuts away some states of the controller.

Every variation is given a name depending on the number of states that are removed. *L000oR0000* is the version with all states, and thus, the most concurrent of all variations. The circuit in Fig. 8.1 represents *L440oR2044*, which

removes 18 states, and results in a more constrained protocol. *L440oR2264* removes an additional 4 states, resulting in a slightly simpler circuit also shown in Fig. 8.6

In this case study, we will focus on the 137 controllers presented in [25] which are speed independent and deadlock-free. In the experiment we will synthesize a circuit for each one of these controllers, and then rediscover the specifications from each one using the proposed mining flow.

Environment setup

The input to our mining flow is a netlist. To generate circuits for each of the 137 controllers, we used petrify [45] to synthesize gate netlists from the specifications. After generating the LTS with free environment from the circuit, we transformed the SAT models into ILP, and used Gurobi [72] to mine the most general specification for each controller.

We configured our mining flow to ensure the following circuit and environment properties:

- Speed independence and delay-insensitive interfacing.
- Strong connectedness.
- Multi-environment interface to ensure the independence between the left and right channels. This will be further discussed in a subsection below.

No other information was given to the miner.

Results

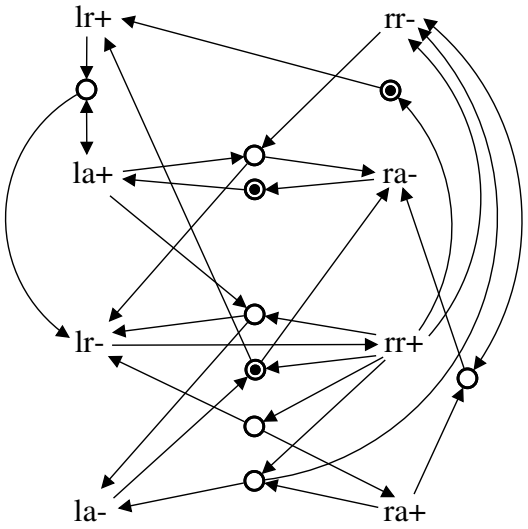
The 137 specifications were mined from the circuits in 177 seconds (Intel Core i5-2520M). Each run of the ILP model took less than 1 second on average, with the rest of time spent in generating the model and preparing the environment.

The size of the ILP model is $O(|T|^2)$ where T is the set of transitions in the LTS of the circuit. However, our implementation performs a preprocessing in which many redundant constraints are removed before generating the ILP model. The most concurrent circuit, with 256 states, also resulted in the largest model, with 267 variables and 996 constraints.

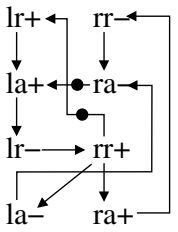
For each one of the 137 circuits, the first snippet obtained from the mining flow was always bisimilar to the original specification of the controller.

	Snippets	**Circuits**
Identical behavior	1	25
Additional behavior	1	59
	2	48
	3	5

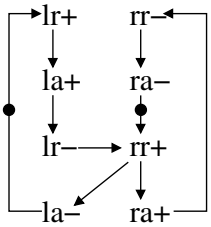
Table 8.2: Circuit implementations allowing additional behavior after removing constraints.



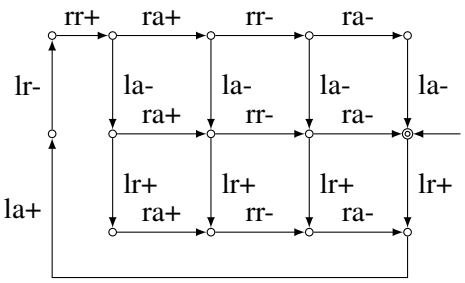
(a) Combined into single STG.



(b) First marked graph.



(c) Second marked graph.



(d) LTS corresponding to (b).

Figure 8.7: Mined specification for L440oR2044.

Relaxing constraints to discover additional behavior

We also experimented our mining flow by discovering additional behaviors when relaxing some of the environmental constraints imposed in previous section. In particular, we allowed the left and right environments to be dependent from each other. In practice this means that the output of one channel can trigger the input of the other channel (i.e., *la* can trigger *ra* and *rr* can trigger *lr*). However, we still preserved the SI and DI interfacing properties.

With this reduced set of constraints, our tool discovered more general specifications for 113 out the 137 controllers. All the specifications still include the original specifications. In addition, out of the 113 specifications with additional behavior, 53 required a minimum of two snippets. That is, no single snippet was able to model the entire behavior of these 53 snippets without violating DI or SI. Table 8.2 shows the total numbers of circuits that exhibited additional behavior and/or required more than one snippet.

An example of a 4-phase protocol where additional behavior is discovered is *L440oR2044*, whose original specification and circuit are shown in Fig. 8.1. An STG showing the additional behavior is represented in Fig. 8.7a. To aid legibility, our mining flow was configured to enforce the marked graph property described in section 8.5, which divides this specification into the two snippets shown in Fig. 8.7b and c.

Notice that the snippet in Fig. 8.7b, however, assumes an environment where the left and right channels are not independent. For example, output $rr+$ triggers input $lr+$, which is on a different channel. Thus, the multi-environment constraints used in the last section would allow only the behavior described by snippet c. This snippet is bisimilar to the original specification of *L440oR2044*.

8.6.2 Mining controllers with choice

This section shows the results of applying the proposed mining flow to a selection of asynchronous controllers from well-known benchmarks. In these examples, we introduce environments with input conflicts, i.e. inputs may disable other inputs.

As in the previous case study, the properties of SI and DI interfacing are enforced. When possible, we also enforce multi-environment interfacing as well as any required mutual exclusion between pairs of input signals.

Table 8.3 reports the total number of snippets discovered by our mining flow, as well as whether the original specification of the circuit was included in one of the discovered snippets. The rest of this section delves into the details of some of the test cases with interesting properties.

Benchmark	Number of snippets	Original specif. included in	ILP runtime [sec.]
SM-latch [95]	1	Snippet #1	0.10
RLM [39]	4	Snippet #4	0.14
1-bit variable [19]	1	Snippet #1	0.31
alloc-outbound [46]	3	Snippet #1	0.73
vmebus [46]	4	Snippet #1	0.12
A/D converter ctrl. [46]	1	Snippet #1	0.43
tsend-csm [65]	–	–	> 1 h.

Table 8.3: Summary of free-choice results.

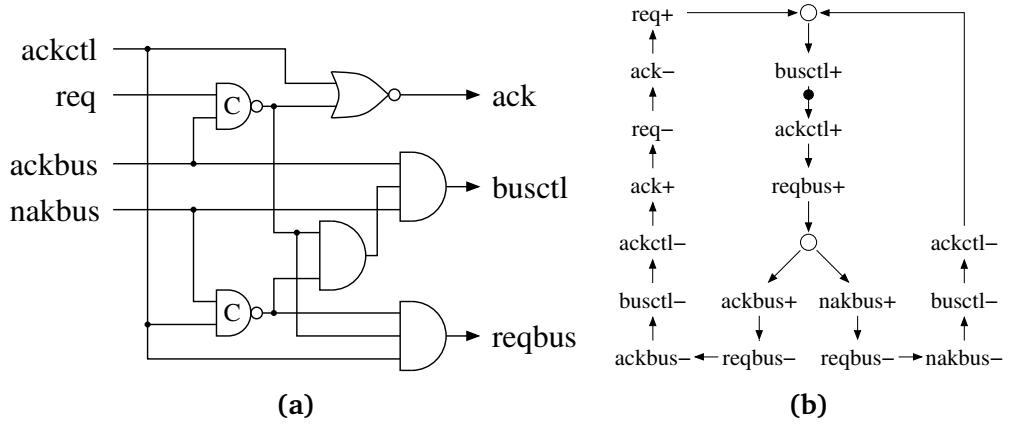
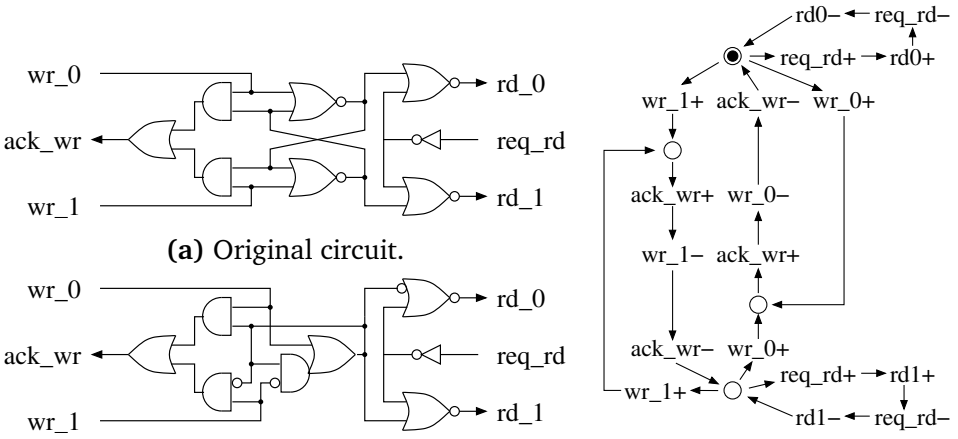


Figure 8.8: Circuit and mined specification for *alloc-outbound*.

alloc-outbound

alloc-outbound is part of a set of well-known academic benchmarks [46], representing part of an HP bus controller. The circuit used as input for the mining flow is shown in Fig. 8.8. The interface is composed of three different environments: 1) $reqbus, ackbus, nakbus$, 2) $busctl, ackctl$, 3) ack, req. Notice only environment 1 has more than one input, with only signals $ackbus$ and $nakbus$ allowed to be in conflict.

Without this constraint, the number of possible SI and DI snippets grows to 12. The ILP runtime also rises up to 1 hour, showing the effectiveness of the multi-environment constraint in restricting the size of the state space. With this constraint, there are only 3 valid snippets, with the original specification being the first discovered snippet, shown in Fig. 8.8b.



(a) Original circuit.

(c) Resynthesized circuit.

(b) Mined specification.

Figure 8.9: Circuit and mined specification for a 1-bit variable.

1-bit variable

In this example we use a simple implementation of a 1-bit variable with a single read and write port [19]. The original circuit is shown in Fig. 8.9a. The write port includes the input signals *wr_0*, *wr_1* as well as the write acknowledgment signal *ack_wr*. The read port includes *req_rd* as well as the response signals *rd_0*, *rd_1*.

Of significance is that the original circuit, in Fig. 8.9a, may go into metastability if both *wr_0* and *wr_1* are asserted. Our mining flow, thus, discovers environments in which *wr_0* and *wr_1* are mutually exclusive.

Hazards may also be produced when simultaneous read and write requests are asserted. In this particular implementation, hazards only occur when the read value is different from the one being written (e.g., read a 1 while writing a 0). Hazards are not produced when both values are the same. The mined specification (not shown in this work because of its complexity) accepts concurrent read/write requests of the same value.

Yet, because of higher-level environmental conditions, it may be desirable to enforce the mined behavior to have mutually exclusive inputs, i.e.,:

$$wr\_0 + wr\_1 + req\_rd \leq 1$$

When configured to honor this property, the mining flow generates the specification shown in Fig. 8.9b.

Figure 8.9c shows an alternative implementation of the circuit obtained by synthesizing the mined specification. Interestingly, this implementation has no metastability problems when both *wr_0* and *wr_1* are asserted, although

glitches may be observed when such situation occurs. This feature, however, is irrelevant for a well-behaved environment that would never allow both inputs to be asserted simultaneously.

Negative results

Not all the experimental results are as attractive as the ones presented in previous sections. One of the major challenges of specification mining is to deal with state explosion. The runtime of the ILP models grows exponentially with the number of states present in LTS(C).

As seen with *alloc-outbound*, constraining the environment is an effective approach to handle state spaces that are initially too large to explore. However, some designs are not amenable to this type of constraints. For example, in *tsend-csm*, separate environments cannot be assumed. Our approach explores the full state space, resulting in large ILP models.

To obtain a reasonable runtime in these situations, further environment constraints are necessary. This is an area for further research. For larger circuits, divide-and-conquer approaches, as in compositional verification scenarios [40, 118], may be necessary.

8.6.3 Publications

The results of this research have been published in the following conference article:

- Javier de San Pedro, Thomas Bourgeat and Jordi Cortadella, *Specification mining for asynchronous controllers*, in Proceedings of the 2016 IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Porto Alegre, Brazil, May 2016.

8.7 Conclusions

The intricate structure of asynchronous controllers makes their design error-prone. Discovering safe specifications contributes to understanding the implicit protocols behind them and their properties.

This chapter has presented a novel approach for behavior discovery that can offer useful mechanisms for re-synthesis and verification.

As new challenges for the future we envision two directions: applying specification mining to compositional verification and mining specifications with bounded delays [77], relative timing [130] and other delay models.

Chapter 9

Conclusions and future work

This chapter concludes this thesis by first summarizing its contributions and their relevance (Section 9.1). Then, we list several of the directions in which the research in this area could be continued, and discuss some potential applications (Section 9.2).

9.1 Summary of contributions

The primary goal behind this thesis has been the research of structure mining approaches to solve a variety of problems in the areas of circuit design and process model visualization. The contributions in this work show that applying graph and structure mining techniques to these challenges results in quantifiable improvements when compared to the existing approaches. Below is a short summary of each contribution.

- *Physical planning for regular layouts:*
  - Chapter 3 has shown the importance of considering physical information during early architectural exploration of CMPs, and introduced a framework for early physical planning of CMPs.
  - Chapter 4 introduced HiReg, a new floorplanning tool that uses frequent subgraph discovery to generate regular floorplans, and shown that highly regular layouts can be produced automatically without incurring excessive area and wire length costs.
- *Visualization of process models:*
  - Chapter 5 presented several methods for simplifying existing process models by computing the usefulness of various parts of the model

with information from the log. Using a set of real-life models, it also experimentally shown how these methods handle the trade-off between simplicity and the other quality metrics.

- Chapter 6 introduced a new discovery flow that mines a series of visualization-friendly Petri nets from logs, rather than centering on a single model covering all behavior. It also shown that only a few models are required to cover significant chunks of the behavior of real-life processes, and that the models generated by the proposed flow have high simplicity when compared to other mining strategies.
- Chapter 7 proposed a novel method for the discovery of duplicate tasks that can be combined with existing process discovery algorithms. In addition, it introduced a set of extensions to the Petri nets formalism. The experiments illustrated how these transformations can be used to generate highly simplified models with minimal loss in precision and generalization.

- *Specification mining for asynchronous circuits:*

  - Chapter 8 introduced the topic of reverse engineering asynchronous circuit specifications, and discussesed potential applications of specification mining in the verification and resynthesis of asynchronous circuits. It also proposed a flow for mining specifications that uses graph mining techniques on the state graph, and demonstrated its effectiveness with a set of benchmarks.

9.2 Future work

In this section we propose a few directions in which the presented research may be continued.

9.2.1 Physical planning for regular layouts

The proposed architectural exploration flow proposes using physical planning flow to enhance the phase of early system-level design during large-scale chip design. In a similar vein, the availability of physical planning information would allow future work the use of accurate power-performance models to further improve the quality of the predictions.

The world of interconnect design for CMPs is also constantly evolving. Addressing the issues presented by the physical planning for alternative in-

terconnect topologies, such as crossbars, would allow for more exhaustive architectural exploration of the large design space of modern CMPs.

In addition, the ideas of regularity and hierarchy discovery (Chapter 4) can be extended into other stages of VLSI design, such as placement. While the ideas could lead to huge increases in the design regularity at cell-level in addition to the block-level, future work in this direction would need to tackle new challenges such as the increased scalability requirements.

9.2.2 Visualization of process models

The methods presented in Chapter 5 are appropriate in scenarios where there is an existing process model, discovered using an existing process mining tool, that needs to be simplified for visualization. On the other hand, Chapter 6 proposes a different technique where multiple models are generated starting directly from the log. We envision this technique as a starting point in the use of transition system properties with the goal of process model visualization.

One of the potential areas for improvement for both techniques is allowing further constraints on the structural type of the simplified Petri nets, such as extended free choice [57] or workflow nets [1]. In addition, further research in the study of the properties of labeled transitions [22] could allow for more efficient mining of other structural classes.

In Chapter 7, a method to discover duplicate tasks was presented. Properly discovering concurrent duplicate tasks, however, is left as future work since changes would be required to many of the underlying assumptions in the formalisms (e.g. by introducing indeterminate transition systems).

Many of the techniques proposed can also be adapted to other formalisms such as BPMN [140]. The study of additional visualization metrics could lead to better understanding of how to evaluate process models. The language of structural tasks proposed in Chapter 7 could also be extended, for example, to allow simple regular expressions in nodes, e.g., $(a|bc)*$.

9.2.3 Specification mining for asynchronous circuits

Chapter 8 introduces the area of specification mining for asynchronous designs, opening many potential applications in the analysis and verification of already implemented designs. Using our approach, a series of snippets describing multiple aspects of the behavior of a system could be obtained from a working circuit, allowing for the compositional verification of a complex system. In addition, the existing implementations could be resynthesized, obtaining circuits with better quality metrics, perhaps using newer toolchains.

The proposed specification and behavioral properties were illustrated using circuits operating under the input/output mode [110]. A future direction for research would be mining specifications of circuits operating under burst mode [53], or with bounded delays [77].

An additional area of improvement for the mining flow is the ability to extract specifications from circuits designed under relative timing [130] assumptions. A relative timing constraint indicates an expected ordering between two events (e.g. when both are enabled, $x+$ always fires before $y+$). Synthesis tools [45] can use these assertions to simplify logic complexity. However, from an analysis tool that is unaware of these timing assumptions, the implemented circuit can appear to violate the delay model constraints (e.g., persistence). To ensure that our mining flow discovers complete specifications, it will be necessary to relax the delay model constraints according to the relative timing assumptions.

Bibliography

[1] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[2] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 1st edition, 2011.

[3] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Technol.*, 8(3):13:1–13:30, May 2008.

[4] W. M. P. van der Aalst, A. K. A. d. Medeiros, and A. J. M. M. Weijters. Genetic process mining. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 48–69, 2005.

[5] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[6] 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*, 23(3):333–363, 2011.

[7] W. M. P. van der Aalst and A. J. M. M. Weijters. Process mining: a research agenda. *Computers in Industry*, 53(3):231–244, 2004.

[8] W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE Trans. on Knowledge and Data Engineering*, 16(9):1128–1142, Sept. 2004.

[9] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core CMPs. In *Proc. of the Annual International Symposium on Computer Architecture*, pages 451–461, 2009.

[10] A. Adriansyah. *Aligning observed and modeled behavior*. PhD, Technische Universiteit Eindhoven, 2014.

[11] A. Adriansyah, J. Muñoz-Gama, J. Carmona, B. van Dongen, and W. M. P. van der Aalst. Measuring precision of modeled behavior. *Information Systems and e-Business Management*, 13(1):37–67, 2015.

[12] S. N. Adya and I. L. Markov. Fixed-outline floorplanning: enabling hierarchical design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):1120–1135, 2003.

[13] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[14] A. Appice and D. Malerba. A co-training strategy for multiple view clustering in Process Mining. *IEEE Trans. on Services Computing*, 2015. Early online access.

[15] F. Balasa and K. Lampaert. Symmetry within the sequence-pair representation in the context of placement for analog design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(7):721–731, Nov. 2006.

[16] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 187–198, New York, NY, USA, 2006. ACM.

[17] P. A. Beerel, R. O. Ozdag, and M. Ferretti. *A designer's guide to asynchronous VLSI*. Cambridge University Press, 2010.

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, Feb. 2008.

[19] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[20] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):113–129, 2005.

[21] E. Best and R. Devillers. Characterisation of the state spaces of live and bounded marked graph Petri Nets. In *Language and Automata Theory and Applications*, volume 8370 of *LNCS*, pages 161–172. Springer, 2014.

[22] E. Best and R. Devillers. The Power of Prime Cycles. In F. Kordon and D. Moldt, editors, *Proceedings of the 37th International Conference on Application and Theory of Petri Nets and Concurrency*, pages 59–78, Torun, Poland, June 2016. Springer.

[23] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[24] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS Press, 2009.

[25] G. Birtwistle and K. Stevens. Modelling mixed 4phase pipelines: Structures and patterns. In *20th IEEE Int. Symp. on Asynchronous Circuits and Systems (ASYNC)*, pages 27–36, 2014.

[26] S. Boettcher and A. G. Percus. Extremal optimization: methods derived from co-evolution. In *Proceedings of the Genetic and Evolutionary Computing Conference*, Orlando, FL, July 1999. Morgan Kaufman.

[27] R. P. J. C. Bose and W. M. P. van der Aalst. Context aware trace clustering: Towards improving process mining results. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 401–412, 2009.

[28] J. Buijs. *Flexible Evolutionary Algorithms for Mining Structured Process Models*. PhD thesis, Technische Universiteit Eindhoven, 2014.

[29] J. Buijs. Receipt phase of an environmental permit application process (âĂŸwaboâĂŹ), coselog project, 2014.

[30] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. On the role of fitness, precision, generalization and simplicity in Process discovery.

In *On the Move to Meaningful Internet Systems*, pages 305–322, Rome, Italy, 2012. Springer.

[31] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. Quality dimensions in process discovery: The importance of Fitness, Precision, Generalization and Simplicity. *International Journal of Cooperative Information Systems*, 23(01), 2014.

[32] A. Burattin and A. Sperduti. PLG: A framework for the generation of business process models and their execution logs. In *Business Process Management Workshops*, volume 66 of *Lecture Notes in Business Information Processing*, pages 214–219, Hoboken, NJ, USA, Sept. 2010. Springer.

[33] A. Bystrov, D. Shang, F. Xia, and A. Yakovlev. Self-timed and speed independent latch circuits. In *6th UK Asynchronous Forum*. University of Manchester, July 1999.

[34] J. Carmona. The label splitting problem. In *Transactions on Petri Nets and Other Models of Concurrency VI*, volume 7400 of *LNCS*, pages 1–23. Springer, 2012.

[35] J. Carmona, J. Cortadella, and M. Kishinevsky. A region-based algorithm for discovering Petri Nets from event logs. In *Business Process Management*, volume 5240 of *LNCS*, pages 358–373. Springer, 2008.

[36] J. Carmona and M. Solé. PMLAB: An scripting environment for Process Mining. In *Proceedings of the BPM Demo Sessions 2014*, pages 16–21, Oct. 2014.

[37] H. H. Chan and I. L. Markov. Practical slicing and non-slicing block-packing without simulated annealing. Technical report, University of Michigan, May 2004.

[38] X. Chen, J. Hu, and N. Xu. Regularity-constrained floorplanning for multi-core processors. *Integration, the VLSI Journal*, 47(1):86–95, 2014.

[39] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT, June 1987.

[40] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 331–346. Springer-Verlag, 2003.

[41] N. Cohen. Several graph problems and their Linear Program formulations. Technical report, INRIA, July 2010.

[42] J. Cong, A. Jagannathan, G. Reinman, and M. Romesis. Microarchitecture evaluation with physical planning. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 32–35, New York, NY, USA, 2003. ACM.

[43] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[44] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct. 2004.

[45] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, Mar. 1997.

[46] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[47] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from Finite Transition Systems. *IEEE T. on Comp.*, 47(8):859–882, Aug. 1998.

[48] Stanford CPU DB. `http://cpudb.stanford.edu`.

[49] W.-M. Dai. Hierarchical placement and floorplanning in BEAR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(12):1335–1349, 1989.

[50] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[51] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan, and C. R. Das. Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs. In *Proceedings of HPCA 2009*, HPCA 2009, pages 175–186, 2009.

[52] J. Davies and F. Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, SAT'13, pages 166–181. Springer, 2013.

[53] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact asynchronous control circuits. In *Proceedings of the IFIP WG10.5 Working Conference on Asynchronous Design Methodologies*, pages 193–207, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[54] A. K. A. de Medeiros. *Genetic Process Mining*. PhD, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2006.

[55] J. de San Pedro. A simulation framework for hierarchical Network-on-Chip systems. Master's thesis, UPC, June 2012.

[56] J. De Weerdt, S. vanden Broucke, J. Vanthienen, and B. Baesens. Active trace clustering for improved process discovery. *IEEE Trans. on Knowledge and Data Engineering*, 25(12):2708–2720, Dec. 2013.

[57] J. Desel and J. Esparza. *Free Choice Petri nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[58] B. F. van Dongen, A. K. A. Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM Framework: A new era in process mining tool support. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets*, pages 444–454. Springer, June 2005.

[59] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4(1):92–98, Jan. 1985.

[60] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. *Acta Informatica*, 27(4):343–368, 1990.

[61] C. C. Ekanayake, M. Dumas, L. García-Bañuelos, and M. La Rosa. Slice, mine and dice: Complexity-aware automated discovery of business process models. In *Business Process Management*, volume 8094 of *LNCS*, pages 49–64. Springer, 2013.

[62] D. Fahland and W. M. P. van der Aalst. Simplifying mined process models: An approach based on unfoldings. In *Business Process Management*, volume 6896 of *LNCS*, pages 362–378. Springer, 2011.

[63] D. Fahland and W. M. P. van der Aalst. Simplifying discovered process models in a controlled manner. *Information Systems*, 38(4):585–605, 2013.

[64] G. Faust, R. Zhang, K. Skadron, M. Stan, and B. Meyer. ArchFP: rapid prototyping of pre-RTL floorplans. In *Proceedings of the 2012 IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, pages 183–188, 2012.

[65] R. M. Fuhrer and S. M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Kluwer Academic Publishers, 2001.

[66] E. R. Gansner, E. Koutsofios, S. C. North, and K. Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993.

[67] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[68] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[69] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens. Robust process discovery with artificial negative events. *J. Mach. Learn. Res.*, 10:1305–1340, June 2009.

[70] T. Gschwind, J. Pinggera, S. Zugal, H. A. Reijers, and B. Weber. A linear time layout algorithm for business process models. *J. Vis. Lang. Comput.*, 25(2):117–132, 2014.

[71] C. Günther. *Process Mining in Flexible Environments*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[72] Gurobi Optimization. Gurobi Optimizer reference manual, 2015.

[73] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. *SIGSOFT Softw. Eng. Notes*, 30(5):31–40, Sept. 2005.

[74] J. Herbst and D. Karagiannis. Workflow mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004. Process / Workflow Mining.

[75] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[76] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. Van Der Wijngaart. A 48-core IA-32 processor in

45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173–183, 2011.

[77] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, Mar. 1954.

[78] W. N. N. Hung, X. Song, T. Kam, L. Cheng, and G. Yang. Routability checking for three-dimensional architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(12):1371–1374, 2004.

[79] IEEE Task Force on Process Mining. Process Mining Manifesto. In *Business Process Management Workshops*, pages 169–194, Clermont-Ferrand, France, 2011.

[80] International Technology Roadmap for Semiconductors. `http://www.itrs.net/reports.html`.

[81] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(01):75–105, 2013.

[82] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, Sept. 1967.

[83] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-03-18].

[84] I. Jonyer, L. B. Holder, and D. J. Cook. Hierarchical conceptual structural clustering. *International Journal on Artificial Intelligence Tools*, 10(1-2):107–136, 2001.

[85] A. B. Kahng. Classical floorplanning harmful? In *Proceedings of the 2000 international symposium on Physical design*, ISPD '00, pages 207–213, New York, NY, USA, 2000. ACM.

[86] A. B. Kahng, I. L. Markov, J. Hu, and J. Lienig. *VLSI physical design: from graph partitioning to timing closure*. Springer, USA, 4 edition, Feb. 2011.

[87] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[88] G. Keller, M. Nüttgens, and A. W. Scheer. Semantische prozeßmodellierung auf der basis ereignisgesteuerter prozeßketten. *Veröffentlichungen des Instituts für Wirtschaftsinformatik*, 89, 1992.

[89] R. M. Keller. A fundamental theorem of asynchronous parallel computation. In *Parallel Processing: Proceedings of the Sagamore Computer Conference*, pages 102–112. Springer, 1975.

[90] N. S. Ketkar, L. B. Holder, and D. J. Cook. Subdue: compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, OSDM '05, pages 71–76, New York, NY, USA, 2005. ACM.

[91] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[92] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[93] V. Klee and G. J. Minty. How good is the simplex algorithm? In *Proc. Third Sympos. Inequalities*, pages 159–175, Los Angeles, 1969. Academic Press.

[94] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar. 1996.

[95] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, Jan. 1998.

[96] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–189, 2014.

[97] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, 2005.

[98] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Mining message sequence graphs. In *33rd International Conference on Software Engineering (ICSE)*, pages 91–100, 2011.

[99] M. Lai and D. Wong. Slicing tree is a complete floorplan representation. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE '01, pages 228–232, Piscataway, NJ, USA, 2001. IEEE.

[100] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *LNCS*, pages 311–329. Springer, 2013.

[101] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering block-structured process models from incomplete event logs. In *Application and Theory of Petri Nets and Concurrency*, volume 8489 of *Lecture Notes in Computer Science*, pages 91–110. Springer, 2014.

[102] F. T. Leighton. New lower bound techniques for VLSI. *Mathematical systems theory*, 17(1):47–70, 1984.

[103] J. Li, D. Liu, and B. Yang. Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In *Advances in Web and Network Technologies, and Information Management*, pages 396–407, Huang Shan, China, June 2007. Springer.

[104] W. Li, Z. Wasson, and S. Seshia. Reverse engineering circuits using behavioral pattern mining. In *IEEE Int. Symp. on Hardware-Oriented Security and Trust (HOST)*, pages 83–88, June 2012.

[105] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big Data: the next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, June 2011.

[106] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 302–311. Springer, New York, NY, 1990.

[107] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177, 1993.

[108] J. Mendling, G. Neumann, and W. M. P. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *On the Move to Meaningful Internet Systems*, volume 4803 of *LNCS*, pages 113–130. Springer, 2007.

[109] M. Monchiero, R. Canal, and A. Gonzalez. Power / performance / thermal design-space exploration for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):666–681, 2008.

[110] D. E. Muller and W. S. Bartky. Theory of asynchronous circuits. Technical report, University of Illinois, Graduate College, Digital Computer Laboratory, Dec. 1955.

[111] J. Muñoz-Gama and J. Carmona. A fresh look at precision in process conformance. In *Business Process Management*, volume 6336 of *LNCS*, pages 211–226. Springer, 2010.

[112] T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, Apr. 1989.

[113] N. Nikitin. *Automatic Synthesis and Optimization of Chip Multiprocessors*. PhD thesis, UPC, Apr. 2013.

[114] N. Nikitin, J. de San Pedro, J. Carmona, and J. Cortadella. Analytical performance modeling of hierarchical interconnect fabrics. In *Proceedings of the Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS)*, pages 107–114, 2012.

[115] N. Nikitin, J. de San Pedro, and J. Cortadella. Architectural exploration of large-scale hierarchical chip multiprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1569–1582, 2013.

[116] R. H. J. M. Otten. Efficient floorplan optimization. In *Proc. International Conf. Computer Design (ICCD)*, pages 499–502, 1983.

[117] J. Rissanen. *Stochastic complexity in statistical inquiry*, volume 15. World scientific, 1998.

[118] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Asynchronous Design Methodologies*, pages 129–137. IEEE Computer Society Press, May 1995.

[119] F. Rubin. The lee path connection algorithm. *IEEE Trans. Comput.*, 23(9):907–914, Sept. 1974.

[120] S. M. Rubin and R. Reddy. The locus model of search and its use in image interpretation. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, volume 2 of *IJCAI'77*, pages 590–595, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

[121] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev. What is the cost of delay insensitivity? In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 316–323, Nov. 1999.

[122] K. Samadi. *Accurate estimators and optimizers for networks-on-chip*. PhD thesis, UC San Diego, 2010.

[123] K. Sankaranarayanan, S. Velusamy, M. Stan, C. L, and K. Skadron. A case for thermal-aware floorplanning at the microarchitectural level. *Journal of ILP*, 7, 2005.

[124] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27 – 64, 2007.

[125] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, Norwell, MA, USA, 2nd edition, 1995.

[126] J.-L. Song, T.-J. Luo, S. Chen, and W. Liu. A clustering based method to solve duplicate tasks problem. *Journal of University of Chinese Academy of Sciences*, 26(1):107, 2009.

[127] M. Song, C. W. Günther, and W. M. P. van der Aalst. Trace clustering in Process Mining. In *Business Process Management Workshops*, volume 17 of *LNBIP*, pages 109–120. Springer, 2009.

[128] D. Sreenivasa Rao and F. J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1198–1208, Aug. 1993.

[129] K. Srinivasan and K. S. Chatha. A low complexity heuristic for design of custom network-on-chip architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '06, pages 130–135, Leuven, Belgium, 2006. European Design and Automation Association.

[130] K. S. Stevens, R. Ginosar, and S. Rotem. Relative timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, Feb. 2003.

[131] X. Tang, R. Tian, and M. D. F. Wong. Minimizing wire length in floorplanning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1744–1753, 2006.

[132] S. K. L. M. vanden Broucke. *Advances in Process Mining*. Phd, Katholieke Universiteit Leuven, 2014.

[133] B. Vázquez-Barreiros, M. Mucientes, and M. Lama. Mining duplicate tasks from discovered processes. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data*, volume 1371, pages 78–82, Brussels, Belgium, June 2015. CEUR.

[134] B. Vázquez-Barreiros, M. Mucientes, and M. Lama. ProDiGen: Mining complete, precise and minimal structure process models with a genetic algorithm. *Information Sciences*, 294:315–333, 2015.

[135] R. Wang and N. Shah. Scalable hierarchical floorplanning for fast physical prototyping of systems-on-chip. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, ISPD '12, pages 187–192, New York, NY, USA, 2012. ACM.

[136] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, July 2003.

[137] A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible Heuristics Miner (FHM). In *Computational Intelligence and Data Mining*, pages 310–317, 2011.

[138] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process discovery using Integer Linear Programming. In *Applications and Theory of Petri Nets*, volume 5062 of *LNCS*, pages 368–387. Springer, 2008.

[139] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[140] S. A. White and D. Miers. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies Inc., 2008.

[141] D. F. Wong and C. L. Liu. A new algorithm for floorplan design. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, DAC '86, pages 101–107, Piscataway, NJ, USA, 1986. IEEE Press.

[142] B.-S. Wu and T.-Y. Ho. Bus-pin-aware bus-driven floorplanning. In *Proceedings of the 20th Great Lakes Symposium on VLSI*, GLSVLSI '10, pages 27–32, New York, NY, USA, 2010. ACM.

[143] J. Z. Yan and C. Chu. DeFer: deferred decision making enabled fixed-outline floorplanning algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):367–381, Mar. 2010.

[144] B. Yao, H. Chen, C.-K. Cheng, and R. Graham. Floorplan representations: Complexity and connections. *ACM Trans. Des. Autom. Electron. Syst.*, 8(1):55–80, Jan. 2003.

[145] T. T. Ye and G. D. Micheli. Physical planning for on-chip multiprocessor networks and switch fabrics. In *Proceedings of the 14th IEEE International*

Conference on Application-Specific Systems, Architectures, and Processors, ASAP 2003, pages 97–107, The Hague, The Netherlands, June 2003.

[146] E. F. Y. Young, C. C. N. Chu, and M. L. Ho. Placement constraints in floorplan design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):735–745, 2004.

[147] F. Y. Young and D. F. Wong. How good are slicing floorplans? In *Proceedings of the 1997 International Symposium on Physical Design*, ISPD '97, pages 144–149, New York, NY, USA, 1997. ACM.