University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Data-aware Soundness Verification and Repair of Data Petri-Nets

Supervisor
Prof. Davide Bresolin
University of Padova

Co-supervisor
Dr. Matteo Zavatteri
University of Padova

Master Candidate
Aurelo Makaj

Student ID
2044583

Academic Year
2022-2023

"Concern should drive us into action and not into a depression. No man is free who cannot control himself."
— Pythagoras

Abstract

Business processes are one of organizations' core assets, impacting directly on the quality of products and services and on the revenue of corporations. A failure in such processes can negatively affect the core of the organization and its production chain. For such reasons, various tools, techniques, and methods have been gathered from different disciplines and collected in what is called Business Process Management (BPM). Petri Net, or Place/Transition (P/T) Net, is one such tool: it is a powerful modeling formalism combining a well-defined mathematical theory with a graphical representation of the dynamic behavior of systems. Given the endless heterogeneity of cases and the extensive combinations between the various resources that interact in a process, additional Petri Net extensions were designed through the years, each aiming to provide more expressive power to describe a broader range of systems. One is Data Petri Net (DPN). This Petri Net-based modeling formalism allows adding a data dimension to the traditional control flow, which permits both the description of how data evolve through the lifecycle of the process and the decisions based on such data.

The presence of unreachable activities or steps without the possibility to proceed further falls under the concept of soundness, and its verification has been a central topic for different research activities. The following work hooks into soundness verification and provides an approach to repair Data Petri Nets concerning data and decision perspective. It provides an extensive and accurate description of unsound DPNs that helps the reader understand the base cases and their resolution. Due to the presence of false-positive cases within the existing soundness verification techniques in the literature, this work studies such issues and proposes a patch aimed at the correct identification of these unsound DPNs. The verification and repair techniques rely on the Constraint Graph structure, a finite symbolic abstraction of the possibly infinite traces of a DPN. The algorithms proposed assume that the underlying Petri Net is sound, thus focusing solely on the repair of data that determines the behavior of the net, aiming to restore its soundness. They seek to provide a solution with minimal changes from the original DPN, which helps the process designer verify its correctness and plan further developments and optimizations.

Contents

Abstract v

List of figures ix

List of tables xi

1 Introduction 1

2 Business Process Management 3
- 2.1 Business Processes . . . 3
- 2.2 History of BPM . . . 5
- 2.3 Business Process Modeling . . . 6
  - 2.3.1 Event-driven Process Chain (EPC) . . . 7
  - 2.3.2 Unified Modeling Language (UML) . . . 7
  - 2.3.3 Business Process Model and Notation (BPMN) . . . 8

3 Petri Nets 11
- 3.1 Petri Net Definition . . . 11
  - 3.1.1 Mathematical formalism . . . 12
  - 3.1.2 Graphical representation . . . 13
  - 3.1.3 Transition firing . . . 13
- 3.2 Expressive power and properties . . . 14
  - 3.2.1 Modeling primitives . . . 14
  - 3.2.2 Properties . . . 16
- 3.3 Data Petri Nets . . . 18
  - 3.3.1 Definition . . . 18
  - 3.3.2 Execution Semantics . . . 19

4 Soundness Verification 21
- 4.1 System of Difference Constraints . . . 21
- 4.2 Constraint Graph . . . 23
- 4.3 Data-aware Soundness Verification . . . 27
  - 4.3.1 Soundness Definition . . . 27
  - 4.3.2 Verification Procedure . . . 29
- 4.4 Satisfiability Modulo Theories (SMT) . . . 30
- 4.5 Co-reachability Analysis . . . 32
  - 4.5.1 False-positive Sound DPN . . . 32
  - 4.5.2 Co-reachability . . . 34
  - 4.5.3 Co-reachability procedure . . . 37

5 Data-aware Process Repair 41
- 5.1 Repair definition and prerequisites . . . 41

5.2 Acyclic DPNs 43
5.2.1 Dead nodes 43
5.2.2 Missing transitions 49
5.3 Cyclic DPNs 50
5.4 Tests and evaluation 53

6 Concluding Remarks 57

References 59

Acknowledgments 63

Listing of figures

2.1 Business Process example . . . 4
2.2 Work evolution . . . 5
2.3 Business Process Model Languages . . . 6
2.4 EPC example . . . 7
2.5 UML Example . . . 8
2.6 BPMN Example . . . 8

3.1 Petri Net example . . . 12
3.2 Enabling Rule cases . . . 14
3.3 Sequential . . . 14
3.4 Choice . . . 15
3.5 Parallelism . . . 15
3.6 Synchronization . . . 15
3.7 Extended Petri Net example . . . 16
3.8 Reachability example . . . 17
3.9 Unbounded Petri Net . . . 17
3.10 Non-live Petri Net . . . 18
3.11 Data Petri Net example . . . 19

4.1 DBM subtraction . . . 25
4.2 Constraint graph . . . 27
4.3 Unsound DPN for P_1 . . . 28
4.4 Unsound DPN for P_2 . . . 28
4.5 Constraint graph for P_3 . . . 29
4.6 SMT correlation with Constraint Graph . . . 33
4.7 False-positive sound DPN . . . 35
4.8 Difference Constraints Set . . . 39

5.1 Cyclic DPN with infinite Constraint Graph . . . 42
5.2 Cyclic DPN with finite CG . . . 43
5.3 ForwardRepair example . . . 47
5.4 ForwardRepair and BackwardRepair example . . . 48
5.5 BackForwardRepair example . . . 49
5.6 Co-reachable cyclic solution . . . 54

Listing of tables

4.1 Co-reachability analysis 39

5.1 Tests summary 55

1 Introduction

Business processes define one of the core aspects of organizations. In the current times, where the market is constantly changing, and each day new business arises, it is mandatory to monitor, analyze, and improve the production lines to keep up. The ability to model and obtain suitable abstract representations of such processes helps detect anomalies or other factors that may prevent maximal or cost-efficient solutions from being found. Given this increasing importance, the scientific community developed a series of techniques and tools grouped under the name of *Business Process Management* through the years.

Petri Nets is one of the numerous formalized models for describing different kinds of processes. Thanks to their flexibility, they proved to be helpful in the context of business processes. Different variants of Petri Nets are present today. Still, the one utilized in this work is Data Petri Nets (DPN), which extends the traditional representation by adding a data dimension to the underlying model. Thanks to a solid mathematical background and a simple yet powerful graphical visualization, (Data) Petri Nets offers different properties for understanding the workflow and discovering irregularities.

Soundness verification is the field concerned with analyzing a business process model to detect possible anomalies within the process. In fact, "*well-formed business processes correspond to sound workflow nets*"[1]. Soundness verification leverages the properties offered by the model representation of a process. It allows discovering problems in the flow that would have been unnoticed by looking at the concrete case.

The following work hooks into soundness verification and provides an approach to repair Data Petri Nets concerning data and decision perspective. It offers a detailed explanation of what defines a sound Data Petri Net and the algorithm required for its verification based on existing techniques in the literature. It draws attention to a false-positive case, where an unsound DPN is incorrectly detected as sound by current solutions. It explores such a case, allowing the reader to understand the causes leading to a false-positive identification. Such causes are resolved by introducing a new soundness verification strategy and the corresponding procedure. The joint combination of the existing solutions with this new algorithm permits the correct identification of unsound Data Petri Nets.

The last part of this work covers the repair of unsound Data Petri Nets. The repair process assumes that the underlying dataless Petri Net is sound (that is, on the workflow level), thus focusing solely on the repair of the data dimension. Two different algorithms are presented, one for repairing acyclic DPNs and one for cyclic DPNs. Moreover, due to the presence of cycles and possible infinite executions, cyclic nets must satisfy specific restrictions to enable a feasible repair.

The rest of the work is structured as follows. Chapter 2 gives an overview about *Business Process Managament*. It introduces *business process* and describes the elements composing them, then proceeds to explore the historical background and events that lead to the current approach concerning work management. The last section enumerates different modeling tools developed by the scientific community through the years and used in *Business Process Mangament*. Chapter 3 focuses on Petri Nets, explaining the characteristics that made them so famous and valuable in the representation of business processes. After an overview of its expressive power and properties, it introduces Data Petri Net, the Petri Net variation on which this work is based. Chapter 4 starts by introducing the mathematical foundations and structures used by the soundness verification procedures. It proceeds by defining a *data-aware sound* DPN, the conditions that must be satisfied to be so, and the existing procedure in the literature for its detection. It investigates a false-positive sound DPN, explaining how and why it was detected as sound, and, after a brief but necessary detour on Satisfiability Modulo Theories, it concludes by presenting the new approach for detecting these false-positive cases. Finally, Chapter 5 covers the repair of unsound Data Petri Nets. It explains what it means to repair a DPN and the prerequisites to be satisfied by the models to apply the repair. It describes the algorithms for repairing acyclic and cyclic DPNs and concludes with selected case studies.

2

Business Process Management

The steady and ever-increasing number of businesses and organizations worldwide has forced managers to pay more and more attention to internal structure analysis and evaluation to optimize processes and reduce costs. Business processes are one of the organizations' core assets, impacting directly on the quality of products and services and on the revenue of corporations. A failure in them can negatively affect the core of the organization and its production chain. This critical aspect fueled the exploration of new tools and methodologies to verify and correct business processes effectively.

This chapter aims to give an introductory perspective on the Business Process Management (BPM) discipline. The first section is focused on describing what business processes are, which elements compose them, and why they are important. The second section goes through BPM's history, starting from the roots and continuing with the evolution of the working processes, to allow the reader to understand why BPM was born in the first place. Finally, the last section describes some modeling languages designed through the years to describe real-life cases.

2.1 Business Processes

Processes are everywhere and wildly different, from the inside of the human body to more external ones, like computer processes. The definition of process can change from field to field. Still, generally, it can be defined as a "*a series of actions or operations conducing to an end*" [2]. Through the years, with the industrialization and evolution of the production sector, factories and corporations increasingly replaced individual, autonomous workers. Instead of having a single person, multiple workers are assigned to specific and smaller tasks, all cooperatively operating to reach a common goal. Later in the years, the concept of *business process* was born.

Business processes don't have a unique definition: in the literature review [3] are listed two, "*a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer*" and "*a specific ordering of activities across time and place, with a beginning and an end with clearly defined inputs and outputs*".

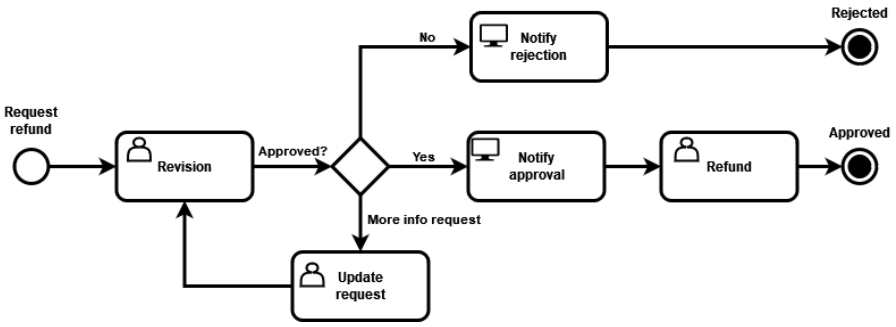


Figure 2.1: An example of a business process representing a refund procedure, from the request to its approval or rejection.

While structured differently, both definitions refer to activities executed to produce one or more outputs. Moreover, the following elements constitute a business process [4]:

- *Events*, atomic things that happens without duration.
- *Activities*, units of works that last in time. When the activities are relatively small, the term *tasks* can be used.
- *Decision points*, points in time that change the execution of the process.
- *Actors*, which includes both humans and systems working on behalf of them. Moreover, it can also be divided into internal and external
- *Objects*, physical and informational, representing the resources used in the process.
- *Outcome(s)*, the result of the process's execution

The refund procedure is a simple business process example that can help understand them. The practice starts with a refund request by someone external to the organization, which is immediately handled by an employee whose job is to revise it. From there, the request can take three different paths depending on the approval result; if there is a need for more information, a step requires the requester to provide it. Otherwise, an automatic system notifies the user of the result. If the request is approved, an operator supplies the refund, concluding the procedure. The example provides different business process elements mentioned above, like events (fund request), activities (revision or refund tasks), decisions (approval after the revision), *internal* and *external operators*, and the objects (notification system). Figure 2.1 gives a graphical procedure representation.

Business processes are fundamental for the development and growth of a company. As stated in [3], "*improving the efficiency of the organization directly relates to improving the core business processes*". Being able to discover, select, categorize, and describe all the different parts of a system or production chain already gives an overview, which can be further analyzed for additional improvements. Consequently, a new discipline called ***Business Process Management*(BPM)** arose. BPM can be defined as "*supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information*" [5]. Business Process Management, though, didn't come up out of the blue, but after many years and the evolution of the economic sector and the related research field.

2.2 History of BPM

Business Process Management's origins are dated around 1990, with different roots in computer and management sciences, making it difficult to state the starting point. The path to BPM was not straightforward, as this discipline revolves around work organization in processes composed of several intermediary tasks and interactions. Even if this idea may seem intuitive and natural at first glance, how civilizations have structured their production workflow has changed at different times through the centuries. Figure 2.2 gives a concise but highly informative picture of the workers' focus and capabilities through the years. At the beginning of human history, the focus was on the entire process, meaning that people usually produced the food cultivated by them and crafted the tools by themself. Thus, there was a general knowledge of the production of daily life goods. Through the years, people started grouping from smaller nuclei to villages and from villages to cities, incrementally bigger and more populated. The need to produce all the goods themselves wasn't necessary anymore; thus, people could specialize in a single or few items, like the locksmith, the butcher, and so on. The advent of the Industrial Revolution and the development of the assembly line manufacturing process added a further specialization to individual work. Instead of partaking in all the production steps, workers were assigned only to a specific phase or a few, creating an environment where people could reach a higher level of specialization.

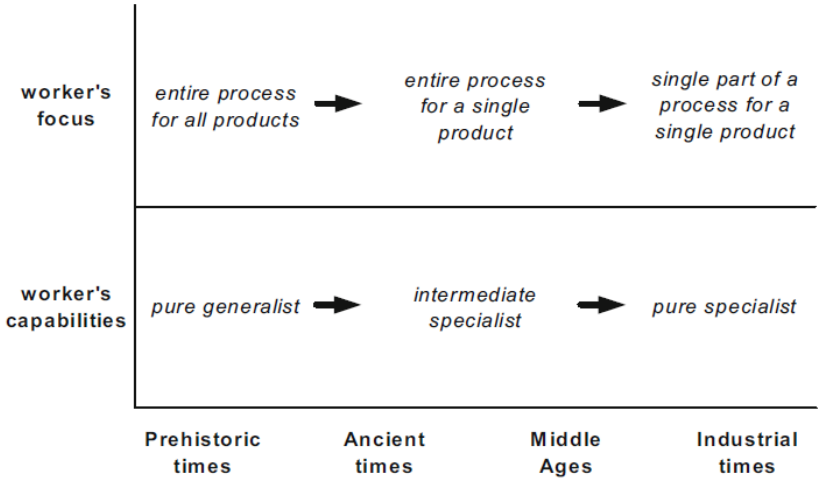


Figure 2.2: The evolution of work and the individual level of specialization [4]

Adam Smith, economics and philosopher of the 18th Century, gave the following example to highlight the importance of division of labor: "*One man draws out the wire, another straights it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving the head; to make the head requires two or three distinct operations; to put it on, is a peculiar business, to whiten the pins is another; it is even a trade by itself to put them into the paper; and the important business of making a pin is, in this manner, divided into about eighteen distinct operations, which, in some manufactories, are all performed by distinct hands*". The example proceeds by saying that the number of pins produced by ten workers using this organization could reach up to 48,000 units daily, while a single person could make no more than 20 units. Even combining ten workers' results cannot be compared with the amount

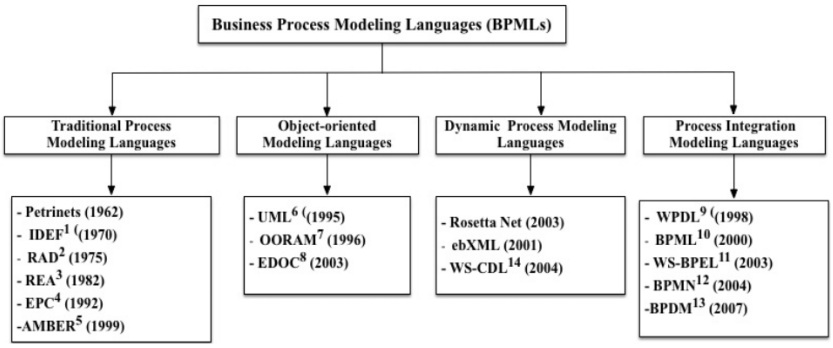


Figure 2.3: Classification of Business Process Modeling Languages (Fig. 1 [8])

produced by the assembly line. The example shows how focusing the effort of each worker on specific fixed tasks can drastically boost the overall production process.

With the passing of the years and the ever-increasing number of factories, the assembly line proved the effectiveness of the division of labor, thus pushing the working scenario into a more specialized effort. A name that helped to drive the world in this direction is Frederick Winslow Taylor (1856 – 1915), an American mechanical engineer whose focus was on the improvement of industrial efficiency, posing the basis of what is called "*scientific management*" (or *Taylorism*, from his name). The continuous evolution of the economic sector and expansions of businesses from factories to companies and later corporations required more attention to the production process and its organization. New roles and positions arose, like *managers*, to oversee and coordinate the work of the individuals partaking in the same process line to create a synergic, harmonious, and continuous flow. The joint effort and research in these new sciences led to new concepts and methodologies, like the *business process* idea described in the previous section and their management, which have been collected in what is currently known as *Business Process Management*.

2.3 Business Process Modeling

Business Process Management obtained extensive attention through the years, aiming to discover new methodologies and expand the tools and techniques available for analyzing and improving business organizations. The ability to represent business processes became vital for redesigning work and distributing responsibilities between the different resources partaking in it [6]. Moreover, sole modeling is not enough but also using the proper annotation for specific goals or aspects of the process that must be analyzed. As stated in [7]: "*Process modeling is widely used within organizations as a method to increase awareness and knowledge of business processes, and to deconstruct organizational complexity. It is an approach for describing how businesses conduct their operations and typically includes graphical depictions of at least the activities, events/states, and control flow logic that constitute a business process*".

The academic world gave birth to multiple models through the years, as different notations have different capabilities. For example, understanding processes usually requires the use of pragmatic approaches, while for their analysis, rigorous paradigms are preferred [9]. Figure 2.3 shows how the numerous business process modeling

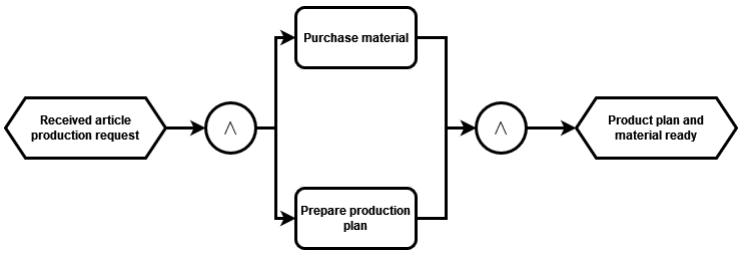


Figure 2.4: An EPC model describing a simple process

languages can be classified. Following is the description of three modeling solutions to give the reader a general overview and a visual aid for understanding the similarities between them. Not listed below, Petri Nets are treated more in-depth in Chapter 3.

2.3.1 Event-driven Process Chain (EPC)

Event-driven Process Chain (EPC) is a traditional process modeling language that aims to "*describe processes on the level of their business logic, not necessarily on the formal specification level, and to be easy to understand and use by business people*" [10]. Given the lack of formalism and the goal of being easily understandable, the diagram is quite simple and composed of a few elements: functions, events, and logical connectors. Functions are the basic building blocks that describe executable activities or tasks, events represent the situation before or after a function, and finally, the logical connectors are the glue to combine the previous two and define the control flow. Figure 2.4 depicts a simple process modeled with EPC. The diagram is composed of two events and two functions connected by a pair of *AND* ($\wedge$) logical connectors. When an order is received, this triggers the parallel execution of two tasks: purchasing the material and preparing the production plan. The conclusion of both triggers an additional event, which describes the situation after the tasks have finished. Other logical connectors are the *OR* ($\vee$) and *XOR* connectors, which give an additional degree of flexibility in the model representation.

2.3.2 Unified Modeling Language (UML)

From the *UML Reference Manual* [12] it can be read: "*The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such systems*". Even though UML was born to describe software systems, it also found practical implementation for modeling business processes. It offers different types of diagrams to represent the numerous entities present in an organization and their production lines; *activity diagrams* allow to model processes, while *class* and *object* diagrams permit to define the underlying structural organization [11]. Figure 2.5 shows an example of a UML activity diagram that models the selling process of computer hardware products. It is composed mainly of two types of nodes: the rounded-edge blocks are *actions*, defining the executable tasks of the process, while the smaller, diamond-shaped boxes are control blocks, which assume different names based on the number of incoming and outgoing arcs: when one arc enters and multiple

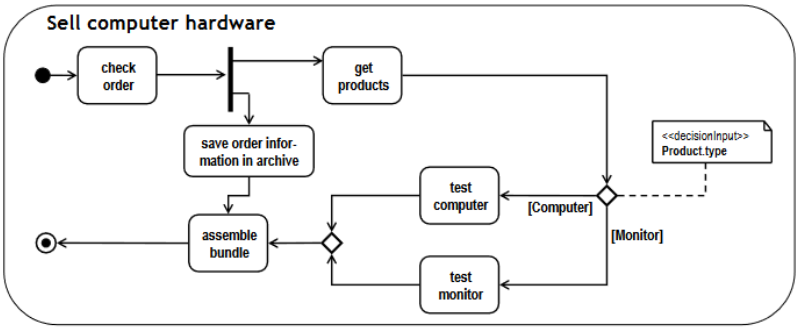


Figure 2.5: UML example describing the selling of computer hardware using an activity diagram (Fig. 5.1 [11])

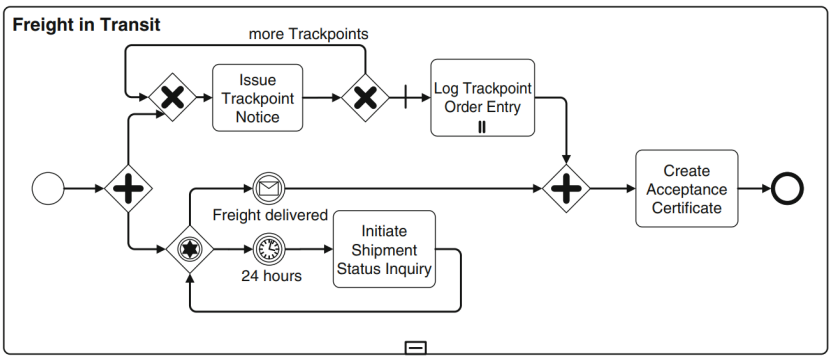


Figure 2.6: Example of a BPMN model describing a freight in transit (Fig. 13.1 [13])

leaves, it is called *decision node*, while on the contrary, when multiple enters and one leaves, it is called *merge node*. Moreover, when multiple flows are possible, defining a *guard* to specify the condition for enabling a path is possible. In the example, this is shown in by the words *Computer* and *Monitor* enclosed in square brackets.

2.3.3 Business Process Model and Notation (BPMN)

Business Process Model and Notation is a younger modeling language compared to the previous ones. Still, through the years, it has become the de-facto standard for business process diagrams [14]. It aims to provide a notation that could be easily understandable by business users of different levels, starting from the business analysts in charge of the initial designs of the process to the technical developers who are going to implement them, and finally by the remaining staff in charge of the deploy and monitoring [15]. BPMN offers many tools to describe business processes and all the details that characterize them. A BMPN process comprises BPMN elements, which can be categorized as *objects, sequence flows, and message flows*. An object can be an *event*, *activity*, or *gateway*, while a sequence flow defines a control flow relation by linking together two objects. Message flows, instead, allow to capture the interaction between processes [16]. Figure 2.6 depicts the process of a freight in transit. The diamond-shaped boxes are gateways, splitting or joining the flow, while the rounded-edge rectangles are activities.

The round icons, the events, have more expressive power than the model languages seen before: they can define a message being sent (*Freight delivered*), an amount of time to wait before proceeding (*24 hours*), and other types. BPMN provides other types of elements, but it is not in the scope of this work to dig deeper into them.

3
Petri Nets

When studying and analyzing phenomena and activities, it could be challenging and not practical to do so by looking at them directly as they are. In many fields of study, using a model for their representation can help to highlight and understand key features of the underlying object of study. Given the large variety of phenomena, different modeling systems were designed through the years to allow researchers to create the most correct and faithful representations of real-life cases.

In this chapter, Petri nets are introduced to give the reader all the necessary foundations to proceed further in the work. First, the mathematical perspective of this modeling tool is provided, starting from its formal definition and continuing with the properties that characterize it. Then, the graphical representation is handled, showing which are the basic building blocks available for the modeling of systems. Different examples are illustrated throughout the chapter so the reader can become comfortable with this representation and the use cases involved. Finally, Data Petri-nets, a Petri net extension used in this work, is described and formalized.

3.1 Petri Net Definition

Petri nets are a historical and widely used modeling tool, *"combining a well-defined mathematical theory with a graphical representation of the dynamic behavior of systems"* [17]. They are attributed to Dr. Carl Adam Petri, who in 1962 gave an early and extensive description in his doctorate dissertation *Kommunikation mit Automaten* [18]. The mathematical foundations allow systematically analyzing the model's behavior, while the graphical representation helps visualize and keep track of its dynamic changes. These characteristics have made Petri nets used across different research fields, like computer science and business management, to describe various event-driven systems, such as computer networks, hardware design, and workflows. In particular, their expressive power makes them a valuable choice to model complex, real-time asynchronous systems where parallel and concurrent activities are involved.

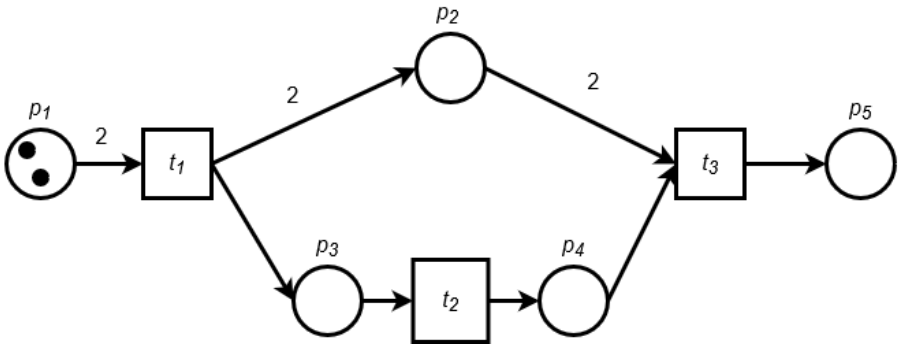


Figure 3.1: An example of a Petri Net. Two tokens are placed initially at place p_1. It can be observed how a transition can generate two (or more) branches, which can have a different number of places themselves. The numbers on the arcs define the tokens required to proceed (from place to transition) and the tokens generated (from transition to place).

Petri Nets fall under the family of bipartite graphs, composed of three elements: *places*, *transitions*, and *directed arcs*. In the modeling of a system, places represent the states or intermediary steps, while transitions describe the events or available actions. Instead, directed arcs work only as a connector, connecting a place to one or more transitions and a transition to one or more places. Connecting elements of the same type is not allowed, so finding a place directly connected to another one is impossible. A feature that characterizes Petri Nets is that places may contain zero or a positive number of *tokens*: in the context of event-driven systems and dynamic behaviors, tokens allow tracking and visualizing the system's evolution.

3.1.1 Mathematical formalism

Petri Nets benefit from a sound mathematical foundation, described through mathematical terms.

Definition 3.1.1 (Petri Net)**.** A Petri Net (PN) is a 5-tuple $\mathcal{N} = (P, T, I, O, m_0)$, where:

- P is a finite set of places
- T is a finite set of transitions
- $P \cap T = \emptyset$
- $I : (P \times T) \to \mathbb{N}$ is a function defining arcs from places to transitions
- $O : (T \times P) \to \mathbb{N}$ is a function defining arcs from transitions to places
- $m_0 : P \to \mathbb{N}$ is the *initial marking*

A *marking* is a function $M : P \to \mathbb{N}$ that defines how many tokens each place contains [19]. Thus, *initial marking* m_0 will describe how the system is at the starting point, where tokens can be either all in the same place or distributed across the model. A place $p_i \in P$ is an *input place* of a transition $t_j \in T$ if $I(p_i, t_j) \geq 1$, while instead it is an *output place* of t_j if $O(t_j, p_i) \geq 1$, where the resulting number specifies the number of tokens of the arc (also called weight of the arc).

Figure 3.1, which shows a graphical representation of a Petri Net, can be mathematically described as a 5-tuple $N = (P, T, I, O, m_0)$, where:

- $P = \{p_1, p_2, p_3, p_4, p_5\}$
- $T = \{t_1, t_2, t_3\}$
- The possible values of the input function I are $I(p_1, t_1) = 2, I(p_2, t_3) = 2, I(p_3, t_2) = 1, I(p_4, t_3) = 1$.
- The possible values of the output function O are $O(t_1, p_2) = 2, O(t_1, p_3) = 1, O(t_2, p_4) = 1, O(t_3, p_5) = 1$
- Initial marking can be described as a vector, each position being a place containing some tokens. Thus, $(2, 0, 0, 0, 0)$

3.1.2 Graphical representation

Graphically, places are represented through circles, while transitions are vertical bars or boxes. Arcs are the joining part, visualized as a directed arrow and smaller circles inside places represent tokens. With such elementary elements, Petri Nets have enough expressive power to model many systems with multiple concurrent activities. Taking again into consideration Figure 3.1, place p_1 contains two tokens, and the number above the arc between p_1 and transition t_1 defines the number of tokens required by the transition to proceed. Typically, the number is omitted when only a single token is required. The fork of two branches from a transition is observable, and each can behave independently with different activities. This form of modular composition ties into Petri Nets' flexibility, allowing focusing and working on smaller parts of the model.

3.1.3 Transition firing

As stated before, token distribution defines the system's evolution and helps to understand its dynamic behavior. [20] Petri Net changes its current state by *firing* transitions, which are dictated by the following two rules:

1. *Enabling Rule*: a transition t is said to be *enabled* when each input place p contains the number of tokens defined by the weight of the arc that connects p to t, that is, $\forall p \in P, M(p) \geq I(p, t)$.
2. *Firing Rule*: once and only when a transition is enabled, the firing of a transition t removes $I(p_i, t)$ tokens from each input place p_i, and puts $O(t, p_o)$ tokens into each output place p_o.

Figure 3.2 gives a graphical representation of the possible cases within the *Enabling Rule*. Given these rules, the firing of a transition t from the current marking M leads to a new marking M' such that:

$$M'(p) = M(p) - I(p, t) + O(t, p), \forall p \in P. \tag{3.1}$$

The notation $M[t\rangle M'$ is used to show that marking M' is reached from marking M by firing t. Further discussion about markings is present in the next section.

(a) Enabled transition **(b)** Not enabled transition

Figure 3.2: Enabling Rule possible cases. In the first case, transition t_1 can fire since two tokens are required, and place p_1 contains three. In the second one, transition t_1 is not enabled since place p_1 doesn't have enough tokens

3.2 Expressive power and properties

3.2.1 Modeling primitives

The number of systems and process models can vary significantly by form or size. Besides the naive cases where only sequential activities are involved, it is not so uncommon that some sub-processes of a system need to run in parallel, or they may be concurrently working with a shared resource. Petri Nets have enough flexibility and expressive power to describe accurately the flow and evolution of such cases. Petri Net's primitives are:

Sequential This is the simplest and most basic primitive. It represents the causal relation between states and transitions. Here, transitions are executed in the same order they appear, allowing thus to define a precedence operation between them. Figure 3.3 shows a sequential execution, where t_2 will indeed be fired after transition t_1 has been fired previously.

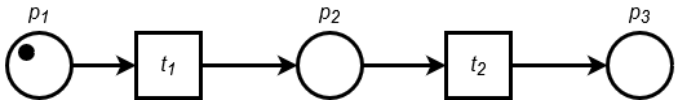


Figure 3.3: Sequential

Choice A *choice* primitive (sometimes called *conflict*) comprises multiple transitions with a shared *input place*. Place's tokens enable all the transitions involved, but they are sufficient for the firing of only one of them. Here, the system chooses one transition or another, either non-deterministically or probabilistically. In Figure 3.4, there is a choice between t_1 and t_2: both of them are enabled by place p_1, which has one token, but only one of them will fire, because after the firing the token will be removed, and this operation will disable the other transition. The choice primitive describes a system or a portion of it where a resource is needed for multiple processes but can be used once at a time.

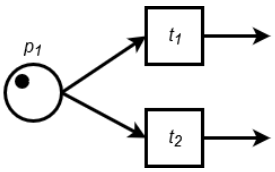


Figure 3.4: Choice

Parallelism As the name suggests, this primitive allows modeling multiple processes executed in parallel. A parallel execution starts with a transition having multiple *output places*; it is essential to keep in mind that the tokens "*consumed*" by a transition don't have to be the same as the tokens generated by the same. Consequently, when the transition is enabled and fires, it will put in each output place the number of tokens defined by the weight of the arc connecting the transition to the place. For example, in Figure 3.5, transition t_1 consumes one token and generates two, one put in place p_1 and one in place p_2, at the same time.

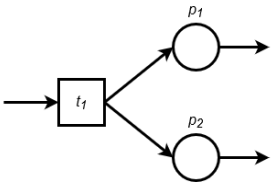


Figure 3.5: Parallelism

Synchronization When a transition requires multiple resources to proceed, it is called synchronization. This scenario is represented by a transition with multiple *input places*, as Figure 3.6 illustrates: transition t_1 will not be enabled until place p_1 and p_2 will have the number of tokens required by each arc. In a more real-life vision, synchronization is used when events or tasks require more resources coming from different sources. For example, the output from a cuisine of a plate consisting of chicken and potatoes will happen when both ingredients have been cooked. Nevertheless, it is not required that the sub-processes terminate at the same time.

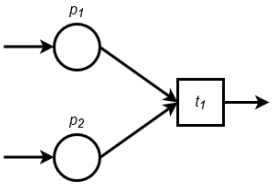


Figure 3.6: Synchronization

Other two forms of primitives that are not relevant for this work but are worth mentioning are *mutual exclusive* and *priority*. *Mutual exclusive* defines two processes that cannot be performed simultaneously due to a shared resource, and *priority* is an addition to the expressive power of Petri Nets, allowing defining a priority between transitions. Petri Net portrayed in Figure 3.7 is a more complex example, showing how the primitives seen above can be put together differently.

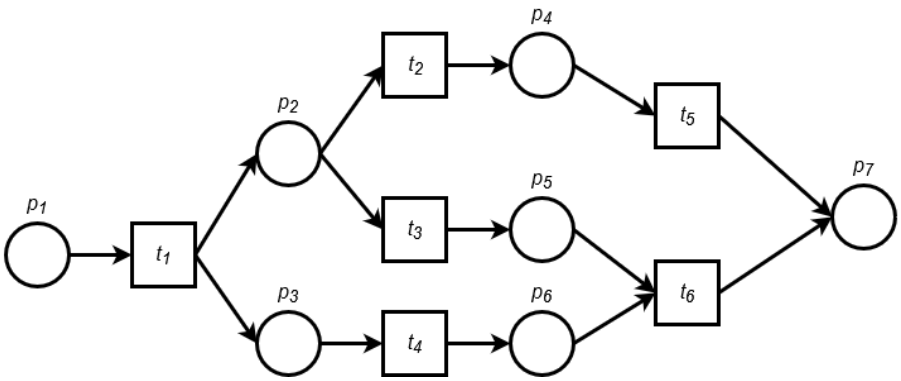
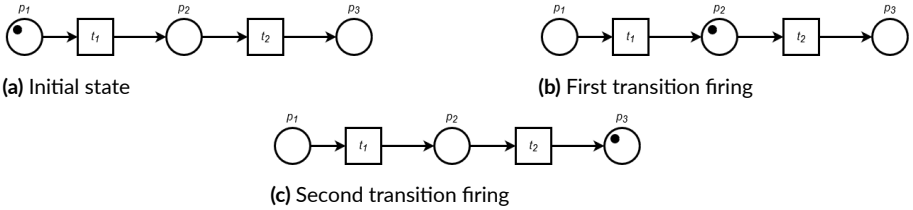


Figure 3.7: A further example of a Petri Net, more complex than the previous one. Multiple branches are present, with different primitives put together. For example, there is a *parallelism* primitive for transition t_1 and a *synchronization* for transition t_6, as well as a *choice* for place p_2

3.2.2 Properties

At this point, Petri Nets have been defined, both from the mathematical perspective and the graphical representation. Designers can leverage previously described building blocks to model increasingly larger and more complex real-life cases. Given the mathematical nature of Petri Nets, they offer a certain number of properties. As Tadao Murata says in [21]: "*A major strength of Petri nets is their support for analysis of many properties and problems associated with concurrent systems*". The properties analyzable from a Petri Net can be divided into two groups, *behavioral* and *structural* properties: *behavioral* properties depend on the initial state of the net, while the structural ones don't, but rely on the topological composition of places and transitions.

Reachability This property helps to understand how a Petri Net evolves after the firing of each transition. As seen in the previous section, each firing moves some tokens from one (or more) place to another one, thus changing the current marking M to a new one M', also written as $M[t\rangle M'$. Consequentially, starting from initial marking m_0, the sequence $m_0[t_1\rangle M_1[t_2\rangle M_2[t_3\rangle \ldots$ defines the evolution of the model, with each marking showing the distribution of tokens after each *transition firing*. A marking M_n is *reachable* from a marking M_i, if there exists a sequence of markings and transitions such that $M_i[t_i\rangle \ldots [t_n\rangle M_n$. The set of markings reachable from a marking M for a Petri Net N is written as $R(N, M)$. The *reachability* property allows understanding the places involved in the execution of the system. If a place in neither one of the markings does contain a token, it may be a superficial activity, or something may need to be corrected in the overall workflow. In the sequential example depicted in Figure 3.8, it is possible to see how the sequence of markings is created through the Petri Net execution. The initial state is $m_0 = (1, 0, 0)$, and after transition t_1 fires, the token is moved in p_2: a new marking M_1 is created, where $M_1 = (0, 1, 0)$. Finally, after transition t_2 fires too, the token is moved again, resulting in the final marking $m_3 = (0, 0, 1)$.



(a) Initial state

(b) First transition firing

(c) Second transition firing

Figure 3.8: Sequential execution of a Petri Net. Initial state is marking $(1, 0, 0)$, evolving in $(0, 1, 0)$ after transition t_1 fires, and concluding in $(0, 0, 1)$ after the last firing.

Boundedness Another relevant property of Petri Nets is *boundedness*. A Petri Net is said to be *bounded* (or *k-bounded*) if the amount of tokens each place contains through the execution of the system doesn't surpass a specific number. More formally, for each place p and marking M reachable from the initial state m_0, $M(p) \leq k$, where k is a positive integer greater than zero. The various examples of Petri Nets seen up until now are all *bounded* (Figures 3.1, 3.7, 3.3). Instead, Figure 3.9 illustrates an *unbounded* Petri Net. From the sequence of markings, it can be seen that places p_1 and p_3 will contain an always-increasing number of tokens. In fact, given an initial state $m_0 = (1, 0, 0)$, transition t_1 generates marking $M_1 = (0, 1, 0)$, while t_2 puts one token in p_3 and two in p_1, thus $M_2 = (2, 0, 1)$. Transition t_1 is enabled and can fire, thus starting the procedure again. The next markings will be $M_3 = (1, 1, 1)$ and $M_4 = (3, 0, 2)$, from where another iteration can start over and over. Petri Nets with a bound $k = 1$ are said to be *safe*. Boundedness property is crucial in those systems where places represent a container since it is certain that no place will contain more than k resources.

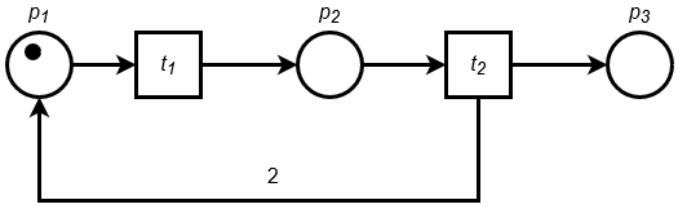


Figure 3.9: An unbounded Petri Net. When the execution reaches the firing of transition t_2, one token is sent to place p_3 and two to p_1. In this way, at each successive iteration, place p_1 and p_3 will contain an always-increasing number of tokens.

Liveness This property is strongly correlated to the presence of deadlocks. Given a Petri Net N with initial marking m_0, $t \in T$ is *live* at m_0 if for all $M \in R(N, m_0)$, there exists a marking $M' \in R(N, m_0)$ such that $M'[t\rangle$. A Petri Net N is *live* if for all $t \in T$, t is *live* at m_0 [22]. More descriptively, a Petri Net is live if, no matter what marking has been reached from m_0, it is possible to fire every net transition, even by passing through some intermediary transitions. Consequently, a *live* net assures a deadlock-free system, meaning there are no intermediary steps where it is blocked. Figure 3.10 shows an example of a non-*live* Petri Net with a deadlock at transition t_3. In fact, t_3 requires a token from both places, but the only token available is either in p_2 or p_3, but never in both. Thus, transition t_3 will never be enabled. *Liveness* property, anyway, due to being very strong and impractical to verify in some systems, has been divided up into multiple levels, each describing a different level of *liveness*. For further in-depth analysis, it is possible to refer to Cassandras and Lafortune's work [23].

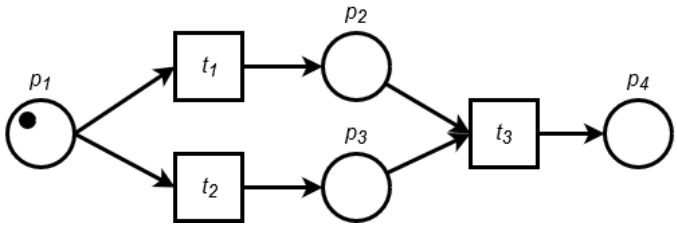


Figure 3.10: A non-*live* Petri Net, where transition t_3 will never be able to fire.

3.3 Data Petri Nets

3.3.1 Definition

Traditional Petri Nets have the power to design many concurrent systems, but they may only be able to represent some of the numerous variations in real life. Different extensions of Petri Nets were formalized through the years to address this issue [24]. One of these, on which the following work has its foundations, is Data Petri Nets. With this extension, adding a data layer to the traditional Petri Net is possible, allowing it to specify how values evolve through its execution. The data perspective is enabled by introducing a new set V containing variables: each variable $v \in V$ is initialized to a fixed value and may change by firing transitions. Moreover, each transition is enriched by a guard: it is not sufficient to have in the input places the required number of tokens by the weight of the arc, but the variables must satisfy the guard, too.

A variable $v \in V$ can be of two types, *read* or *written*, defined respectively as v^r and v^w. Consequently, two further sets are considered: the set of read variables $V^r = \{v^r | v \in V\}$ and the set of written variables $V^w = \{v^w | v \in V\}$. Furthermore, Data Petri Nets always have a special variable Z which is not included in V: the presence of this variable is essential to allow defining guards as *difference constraints*, whose the next chapter gives a more in-depth description. Finally, Data Petri Nets can be defined as follows[25]:

Definition 3.3.1 (Data Petri Net)**.** Let V be a set of variables. Let C_v be the universe of difference constraints over $V^r \cup V^w \cup \{Z\}$. A Data Petri Net (DPN) $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$ is a Petri Net (P, T, I, O, m_0) with additional components describing the new perspectives of the process model:

- V is a finite set of real process variables
- $\alpha_I : V \cup \{Z\} \to \mathbb{R}$ is a function that defines the initial assignments such that $\alpha_I(Z) = 0$
- $guard : T \to C_v$ returns a difference constraint, that is, the guard associated with a transition

As stated in [25], *"Disjunctions are not allowed in the guard for simplicity but without loss of generality"*. Moreover, given a transition $t \in T$, the notation $read(t) = \{v \in V | v \in read(guard(t))\}$ defines a shorthand for the set of variables that are read by the guard of t. The same applies to $write(t)$.

The graphical representation of Data Petri Nets is similar to traditional nets, with the guards above each transition. Figure 3.11 illustrates a basic example of a Data Petri Net, where a single variable a is both read and written.

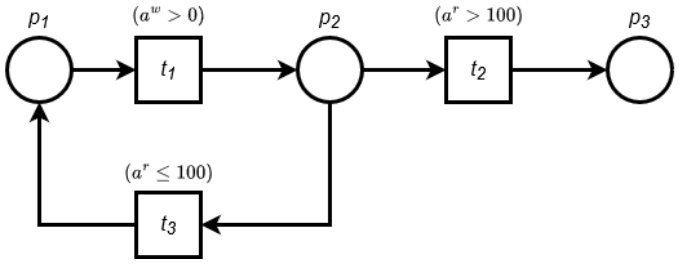


Figure 3.11: A Data Petri Net example, with $\alpha_I(a) = 0$. The variable's value is set greater than zero and can reach place p_3 only if the value is greater than 100.

The guard of transition t_1 means that the variable is changed (a^w) with a value greater than zero. Subsequentially, a *choice* primitive is present: in this case, though, the choice is not made non-deterministically, but it depends on the value that was assigned in the previous step. If the value is greater than the value specified, the flow proceeds to place p_3; otherwise, it starts again from p_1.

3.3.2 Execution Semantics

The introduction of the guards for each transition slightly changes the firing rules and the execution of the Data Petri Net. The central concept remains, enabling transition by the required token amount in each input place. Since there are also variables and constraints over them, the enabling also requires that such conditions are satisfied given the current state. In a Petri Net, a state is represented by a marking M. In a DPN, instead, it is expressed as a tuple (M, α), where M is a marking as previously described, while α is an assignment of variables in V.

Given a state (M, α) and a transition t, the enabling and the firing of the transition t into a new state (M', α') is described as follows:

- transition t is enabled and generates a new marking M' as previously defined in Section 3.1.3
- for each variable $v \in V$, if $v \notin write(t)$ then $\alpha(v) = \alpha'(v)$, that is, the value is unchanged
- the guard is satisfied when read variables are assigned according to α and written variables according to α'.

The notation $(M, \alpha) \xrightarrow{t} (M', \alpha')$ defines a legal firing transition, which can be extended to $(M, \alpha) \xrightarrow{*} (M', \alpha')$ to define a sequence σ of intermediary transitions such that $(M, \alpha) \xrightarrow{\sigma} (M', \alpha')$ or that $(M, \alpha) = (M', \alpha')$. Moreover, given two markings M' and M'', the inequality $M'' \geq M'$ holds if and only if $\forall p \in P$ is true that $M''(p) \geq M'(p)$ and $M'' > M'$ if and only if $M'' \geq M'$ and $\exists p \in P$ such that $M''(p) > M'(p)$. Finally, Petri Nets properties (*reachability*, *boundedness* and *liveness*) explained in Section 3.2.2 apply to Data Petri Nets too.

4

Soundness Verification

As stated previously, Data Petri Nets provide a good expressive power to describe business processes on two dimensions: the first is on the workflow level, showing how the tokens move through the different places of the underlying Petri Net. In contrast, the second one is the data level. Process modeling isn't the goal itself but a starting point of a long and continuous chain of analysis steps aimed at the detection of flaws and activity optimization. One is understanding whether a net is well-formed: in fact, "*Well-formed business processes correspond to sound workflow nets*" [1]. Numerous research activities have discussed soundness verification, proposing different notations, definitions, and algorithms. This work proposes a solution, too, by applying the concept of soundness verification to Data Petri Nets.

The goal of this chapter is firstly to provide the mathematical tools and notations used to develop the verification procedure and, secondly, to describe the solution designed. The first two sections focus on describing *difference constraints*, which are the form the guards in a DPN follow, and the *constraint graph*, a fundamental tool on which both the soundness verification and repair process are built. Then, it follows with a section expressing the definition of a sound DPN and the corresponding algorithm for its verification. After a brief overview of the Satisfiability Modulo Theories (SMT), the last section explains a false-positive sound DPN and the new procedure developed for its detection.

4.1 System of Difference Constraints

A *system of difference constraints* can be defined as follows [25][26]:

Definition 4.1.1 (System of Difference Constraints)**.** A system of difference constraints is a pair $(\mathcal{X}, \mathcal{S})$ where:

- $\mathcal{X}$ is a set of n unknown variables $x_1, x_2, \ldots, x_n$
- $\mathcal{S}$ is a set of m inequalities with form $x_i - x_j \bowtie k$ for $1 \leq i, j \leq n$, where $\bowtie \in \{<, \leq\}$ and $k \in \mathbb{R} \cup \{+\infty\}$

A system of difference constraints is a set of inequalities that must be satisfied altogether. When there exists an assignment of values to the variables such that all constraints are satisfied, the system is said to be *consistent*. In a DPN, each transition guard can be defined only as a difference constraint. Inequalities of the form $x \bowtie k$ and $x = k$ are converted into the correct format by using a new, fresh variable Z that can only hold the zero value. Consequently, guards defined as $x \bowtie k$ are converted into $x - Z \bowtie k$, and $x = k$ are rewritten using two constraints, $x - Z \leq k$ and $Z - x \leq -k$.

Difference constraint sets are often represented through a data structure called *difference bounded matrix*. Given a system of difference constraints $D = (X, S)$ with n variables and m constraints, a *difference bounded matrix* is a $n \times n$ matrix M whose entries are composed by the set $\mathbb{V} = \{(\bowtie_{ij}, m_{ij}) | 0 \leq i, j \leq n, \bowtie_{ij} \in \{<, \leq\}, m_{ij} \in \mathbb{R}\} \cup \{(\leq, \infty)\}$ [27]. Each entry $M_{ij} = (\bowtie_{ij}, m_{ij})$ in the matrix M represents a difference constraint $x_i - x_j \bowtie_{ij} m_{ij}$, that is, the difference constraint between the variables x_i and x_j. If a pair of variables has no corresponding inequality, the entry is specified as $(\leq, \infty)$. Finally, the variable x_0 is the equivalent variable Z described previously.

To give a more practical example of what has been just said, consider the following difference constraints set:

$$Z - x \leq -5 \wedge Z - y < -3 \wedge y - Z < 10 \wedge x - y < 0$$

Such difference constraint set can be represented using the next difference bounded matrix M:

	Z	x	y
Z	$(\leq, 0)$	$(\leq, -5)$	$(<, -3)$
x	$(\leq, \infty)$	$(\leq, 0)$	$(<, 0)$
y	$(<, 10)$	$(\leq, \infty)$	$(\leq, 0)$

Difference bounded matrices are helpful because, with this representation, it is possible to make a crucial operation called *tightening*. The constraint $x_1 - x_2 \leq 3$ is said to be *tighter* than $x_1 - x_2 \leq 5$, and the same can be said of $x_1 - x_2 < 4$ and $x_1 - x_2 \leq 4$. As observed in [28], the sum of the upper bounds on the difference constraints $x_i - x_j$ and $x_j - x_k$ is an upper bound on the difference constraint $x_i - x_k$. Essentially, it holds that $x_i - x_k \bowtie'_{ik} m'_{ik}$, where $m'_{ik} = m_{ij} + m_{jk}$ and

$$\bowtie'_{ik} = \begin{cases} \leq & \text{if } \bowtie_{ij} = \leq \text{ and } \bowtie_{jk} = \leq \\ < & \text{otherwise} \end{cases}$$

The *tightening* operation consists of checking whether $(\bowtie'_{ik}, m'_{ik})$ is a tighter bound than $(\bowtie_{ik}, m_{ik})$, and replacing the entry in the matrix if it is the case. Referring to the example above, the sum of the bounds of $y - Z < 10$ and $Z - x \leq -5$ produces $y - x < 5$, and since $(<, 5)$ is tighter than $(\leq, \infty)$, the entry can be replaced. The result of applying the *tightening* operation to all the matrix entries is the *canonical representation* of the system of difference constraints.

	Z	x	y
Z	$(\leq, 0)$	$(\leq, -5)$	$(<, -5)$
x	$(<, 10)$	$(\leq, 0)$	$(<, 0)$
y	$(<, 10)$	$(<, 5)$	$(\leq, 0)$

Each (consistent) difference constraint set has a unique canonical representation, a fundamental property that brings various benefits. Having more than one bounded matrix representation for the same system is possible. Still, after the tightening operations, all of them are reduced to the same canonical form. This property makes the *canonical form* a crucial key in verifying if two difference constraint sets are equivalent. Two sets of difference constraints are equivalent if and only if they are both inconsistent or have the same canonical form [25]. In this work, the canonical representation is computed by applying a generalized version of the Floyd-Warshall algorithm to the difference constraint set, using the *difference bounded matrix* and applying to each entry the tightening operation.

Algorithm 4.1 describes the procedure utilized. It requires a set of difference constraints as input and gives the canonical form of the system, if consistent, *null* otherwise. It first initializes the difference bounded matrix (lines 3-8), setting all values in the diagonal as $(\leq, 0)$ to represent the constraints of the form $x - x \leq 0$, and $(\leq, \infty)$ to model the absence of constraint between two variables. Then, it fills the matrix with the constraints of the input system (lines 9-12) and applies the Floyd-Warshall algorithm for the tightening operation (lines 13-16). Finally, it checks the system's consistency by verifying that the diagonal's values are all set to $(\leq, 0)$ and, if so, recreates the constraints from the difference bounded matrix into a new set representing the canonical form. The procedure uses two additional functions, *Index*, which maps a variable to an index of the matrix, and *Var*, which, given an index, returns the corresponding variable. The algorithm has a complexity of $\Theta(n^3)$, where n is the number of variables in the system of difference constraints.

Subtraction It is fundamental to describe another operation that works on difference constraints sets and is used by the repair process described in the subsequent chapters. Given two systems of difference constraints, or Difference Bounded Matrices (DBMs), D and E, the subtraction of E from D consists in a new set S satisfying the constraints of D and $\neg E$ [29]. The result $S = D \wedge \neg E$, denoted $D - E$, can be written as:

$$S = D \wedge \neg(\bigwedge_{1 \leq i,j \leq n} e_{ij}) = \bigvee_{1 \leq i,j \leq n} (D \wedge \neg e_{ij}) \qquad \text{(De Morgan Law)}$$

The resulting set S is the union of D constrained by each negated constraint of E. Figure 4.1 visually represents the subtraction between DBMs. The above formula gives a basic, straightforward algorithm to compute the subtraction. David et al., though, in [29] propose various solutions to efficiently execute it, either by exact algorithms or using heuristics. The repair process presented in this work utilizes the straightforward approach, leaving its optimization as a possible future development.

4.2 Constraint Graph

The soundness verification of a Data Petri Net model requires the analysis of the different traces (paths) that belong to the underlying business process. Since such a number may be possibly infinite, previous works [30] have introduced a new structure named *constraint graph* to conveniently describe the traces for the verification procedure. A *constraint graph* is a directed graph that follows the state-transition nature of the DPN, each node composed of a pair of elements: the first one represents the marking of the net, while the second one describes an abstraction of the data, using the canonical form of a system of difference constraints. By using the *constraint*

Algorithm 4.1 Procedure for computing the canonical form of a system of difference constraints

Input: A set of difference constraints C

Output: The canonical form of C if consistent, *null* otherwise

Let $Index$ be a map between variables and indexes in range $[1, n]$
Let Var be the inverse map of $Index$
Let n be the number of variables in C
Let M be a $n \times n$ empty matrix
for $i := 0$ to n **do** ▷ Matrix initialization
    for $j := 0$ to n **do**
        if $i = j$ **then**
            $M[i][j] := (\leq, 0)$
        else
            $M[i][j] := (\leq, \infty)$
for $x - y \bowtie k$ in C **do** ▷ Matrix fill from input constraints
    $(\bowtie', k') := M[Index(x)][Index(y)]$
    if $(\bowtie, k) < (\bowtie', k')$ **then**
        $M[Index(x)][Index(y)] := (\bowtie, k)$
for $k := 0$ to n **do**
    for $i := 0$ to n **do**
        for $j := 0$ to n **do**
            $M[i][j] := min(M[i][j], M[i][k] + M[k][j])$
for $i := 0$ to n **do** ▷ Consistency check
    if $M[i][i] < (\leq, 0)$ **then**
        return *null*
Let C' be an empty set
for $i := 0$ to n **do** ▷ Canonical form construction from matrix
    for $j := 0$ to n **do**
        $x := Var(i)$
        $y := Var(j)$
        $(\bowtie, k) := M[i][j]$
        $C' := C' \cup \{x - y \bowtie k\}$
return C'

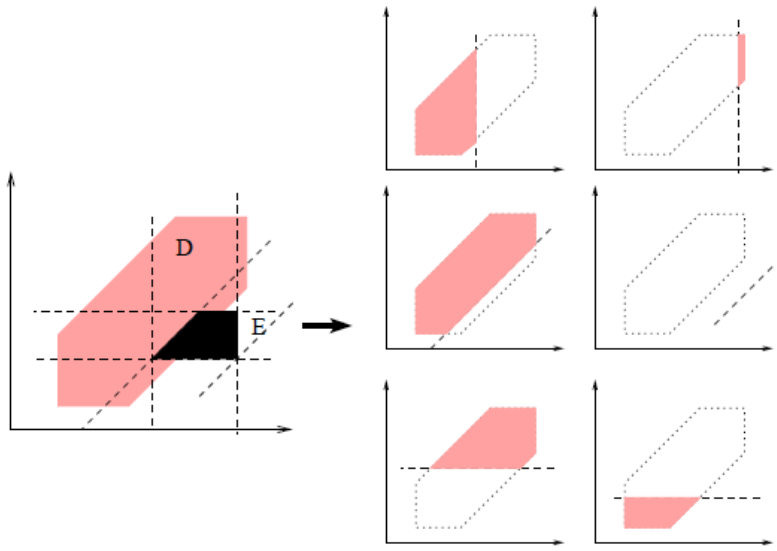


Figure 4.1: Example of DBM subtraction $D - E$. The result consists of the union of all the zones on the right (Fig. 3 [29])

graph, it is possible to obtain a finite-state representation of the possibly infinite traces of the DPN, permitting the assessment of the soundness of the original process.

A crucial operation for constructing the constraint graph is the addition of a difference constraint to an existing set of them. Given a system of difference constraints C, Algorithm 4.2 computes the operation $C \oplus c$, returning the canonical form of the resulting set, if consistent, or *null*, if not. It behaves differently based on the number of variables written in c. If the variables in c are only read, the data do not change; thus, the procedure adds c to the system as is, which imposes a new constraint on the previous set and returns the canonical form of the resulting one. The `CanonicalForm` auxiliary function works as described by Algorithm 4.1, returning the canonical form of the input system if consistent, *null* otherwise. If one or both variables are written, instead, the constraint c is added but with x^w and/or y^w considered as fresh variables, representing the new values. After the computation of the resulting set's canonical form, all the occurrences of x and y are removed, and x^w and y^w are renamed into x and y, to model the fact that the new values take place as the current ones. These last two operations are safe concerning the canonical form previously computed since projection and renaming of variables do not affect minimality [25].

The constraint graph is enriched by a new set of *silent transitions* defined by $\tau_T = \{\tau_t | t \in T\}$, to specify that *guard(t)* does not hold in the current state. A constraint graph is defined as [25]:

Definition 4.2.1 (Constraint Graph of DPN). Let $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$ be a DPN and $\mathcal{M}$ be the set of markings in $\mathcal{N}$. Let $\mathcal{C}_V$ be the universe of canonical forms of system constraints over variables in $V \cup \{Z\}$ and $\tau_T = \bigcup_{t \in T} \tau_t$. The *constraint graph* $\mathcal{CG}_\mathcal{N}$ of $\mathcal{N}$ is a tuple (N, n_0, A), where:

- $N \subseteq \mathcal{M} \times \mathcal{C}_V$ is the set of states of the graph, named *nodes* to distinguish them from the *states* of the DPN
- $n_0 = (m_0, C_0)$ is the initial node, with C_0 being the canonical form of the system of difference constraints $\bigcup_{v \in V} \{v = \alpha_I(v)\}$

Algorithm 4.2 Procedure for computing $C \oplus c$

Input: A set of difference constraints C
Output: The canonical form equivalent to $C \oplus c$ if consistent, *null* otherwise

```
1:  if c = y^r − x^r ⋈ k then                                  ▷ Only read variables
2:      C' := C ∪ {y − x ⋈ k}
3:      return CanonicalForm(C')
4:  else                                                       ▷ At least one variable written
5:      if write(c) = {x} then
6:          C' := C ∪ {y − x^w ⋈ k}
7:      else if write(c) = {y} then
8:          C' := C ∪ {y^w − x ⋈ k}
9:      else                                                   ▷ write(c) = {x, y}
10:         C' := C ∪ {y^w − x^w ⋈ k}
11:     C' = CanonicalForm(C')
12:     C' := C'\{x' − y' ⋈' k' | x' ∈ write(c) or y' ∈ write(c)}
13:     Rename all occurrences of x^w to x and all occurrences of y^w to y in C'
14:     return C'
```

- $A \subset N \times (T \cup \tau_T) \times N$ is the set of arcs such that:
  - a transition $((M, C), t, (M', C'))$ is in A *iff* $M[t\rangle M'$ and $C' = C \oplus guard(t)$ is consistent (i.e., satisfiable);
  - a transition $((M, C), \tau_t, (M, C''))$ is in A *iff* $write(t) = \emptyset$, $\exists M'$ s.t. $M[t\rangle M'$, and $C'' = C \oplus \neg guard(t)$ is consistent

Figure 4.2 shows the constraint graph relative to the Data Petri Net model represented in Figure 3.11. Each node of the graph is composed of a marking, represented by the name of the place where the token is positioned and the system of difference constraints, all connected by the transitions. The first upper-left node represents the initial marking: the place is p_1 with two difference constraints indicating the initialization $\alpha(a) = 0$. In the model, transition t_1 can fire, and since its guard writes variable a, the computation of $C \oplus c$ results in a single inequality, reaching a new node, p_2. It is essential to notice now how multiple arcs and nodes unfold: from place p_2 can fire two transitions, t_2 and t_3, based on the new value of the recently written variable a. If $a > 100$, place p_3 is reached with the corresponding system and connected by transition t_2. Moreover, the condition $write(t_2) = \emptyset$ allows taking into consideration the silent transition τ_{t_2}: the graph reaches a new node with the same marking, but it computes the new set of difference constraint using the negation of the guard of t_2. Note that multiple nodes with the same marking are present: in fact, what makes a node unique is the combination of marking and difference constraints set. For example, in the bottom-right of the figure, there is a node with a self-pointing arc (τ_{t_3}), meaning that the outgoing transition reaches the same node. This representation avoids the creation of an infinite amount of elements, obtaining a finite graph of the model.

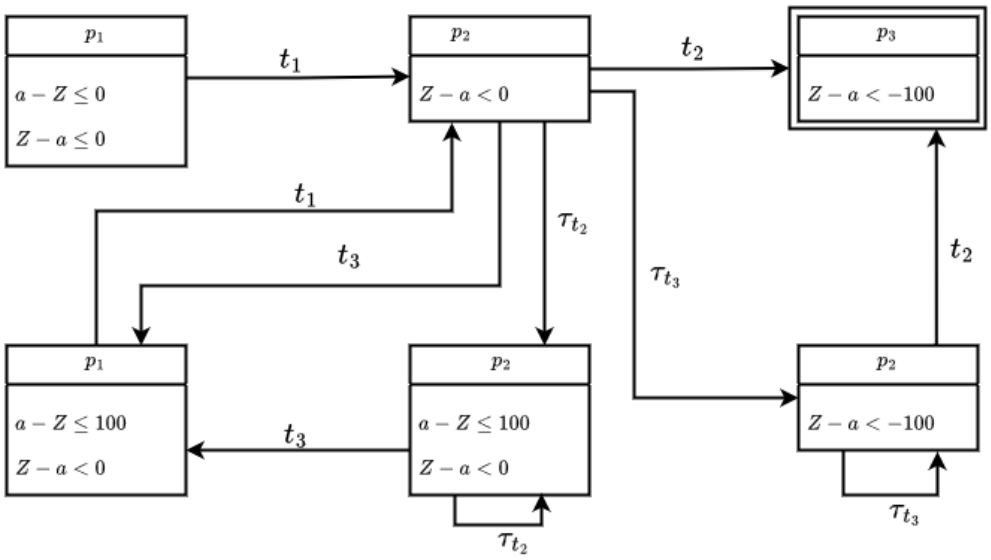


Figure 4.2: Constraint graph of the DPN model at Figure 3.11. The final marking is p_3, denoted with a double frame.

4.3 Data-aware Soundness Verification

Verifying the soundness property of a Petri Net-based model is crucial to understanding whether the net is well-formed, directly affecting the underlying business process. Previous works have already dedicated research to analyzing such property applied to general workflow nets [31], creating the base for further adaptations to Data Petri Nets.

4.3.1 Soundness Definition

De Leoni et al. provide in [32] the lifting of the standard notation of soundness to Data Petri Nets, transitioning from *decision-aware soundness* to *data-aware soundness*, to take into consideration the assignment of variables.

Definition 4.3.1 (Data-aware soundness)**.** Let $\mathcal{N}$ be a DPN with initial marking m_0, final marking M_F, and $Reach_{\mathcal{N}}$ denoting the set of reachable states of $\mathcal{N}$, that is, the set $\{(M,\alpha)|(m_0,\alpha_0) \xrightarrow{*} (M,\alpha)\}$. DPN $\mathcal{N}$ is *data-aware sound* if and only if all the following properties hold:

- ***P*$_1$.** For all reachable states $(M,\alpha) \in Reach_{\mathcal{N}}, \exists \alpha'.(M,\alpha) \xrightarrow{*} (M_F,\alpha')$
- ***P*$_2$.** For all reachable states $(M,\alpha) \in Reach_{\mathcal{N}}, M \geq M_F \implies (M = M_F)$
- ***P*$_3$.** For all transitions $t \in T$, there exists two reachable states (M_1,α_1) and (M_2,α_2) such that $(M_1,\alpha_1) \xrightarrow{t} (M_2,\alpha_2)$

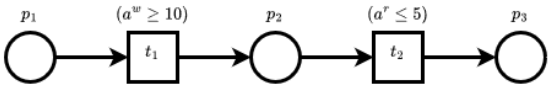


Figure 4.3: A DPN not satisfying P_1

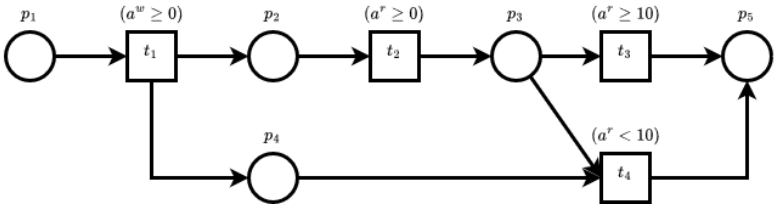


Figure 4.4: A DPN not satisfying P_2

The above definition says that the soundness property is not composed of a single condition, but multiple ones must simultaneously be satisfied by the DPN. The first condition P_1 ensures that no matter the current state, there is at least a sequence of one or more firing transitions allowing it to reach the final marking. It means that a given model must not contain deadlocks or loops that may prevent the final marking from being reached. The need for this condition is straightforward since a process that, from some states, cannot finish appropriately cannot be considered well-formed. Figure 4.3 shows a simple DPN that is not able to reach the final marking p_3; since the guard of transition t_1 sets a new value greater or equal than 10 to variable a, transition t_2 is not enabled since it requires that a is lower or at least equal to 5.

The second condition C_2 ensures that when a process ends, it does so without leaving tokens in places other than the ones specified in the final marking. This condition is needed because, in the model representation of a business process, a token in a place means that the execution has reached that step. Thus, if the execution reaches the final marking, but some tokens are left behind, the process has concluded, but some activities are still being performed elsewhere. Ideally, reaching the final marking should be only done if the previous tasks have all been terminated.

Figure 4.4 depicts a DPN with *initial marking* $m_0 = p_1$ and *final marking* $M_F = p_5$ that does not satisfy condition P_2, that is, there exists a marking M such that $M \geq M_F \wedge M \neq M_F$. The execution starts from place p_1 with transition t_1 writing variable a and putting a token in p_2 and p_4. Then, the only possible step is the firing of transition t_2, which moves the token from p_2 to p_3. Now, at the workflow level, transitions t_3 and t_4 are enabled because the number of tokens required by both are satisfied, thus leaving the choice only at the data level. Notice that when $a^r \geq 10$, t_3 fires reaching the *final marking* p_5. Nevertheless, a token was left behind at place p_4, and it will never reach the final marking since t_4 can no longer become enabled. The current marking $M' = (0, 0, 0, 1, 1)$ does not satisfy condition P_2 because $M' \geq M_F$ but $M' \neq M_F$.

Finally, the third condition P_3 ensures that no dead transitions are present. A transition is dead if it is not possible to enable it during the execution of the process. A well-formed business process needs to satisfy this condition because if a transition cannot fire, there is a problem in the workflow since it doesn't meet the prerequisites for its enabling. Figure 4.5 shows the constraint graph of the DPN modeled in Figure 4.3. Since the guard of transition t_2 cannot be satisfied, it will never fire, thus missing from the constraint graph. Note that here, the condition

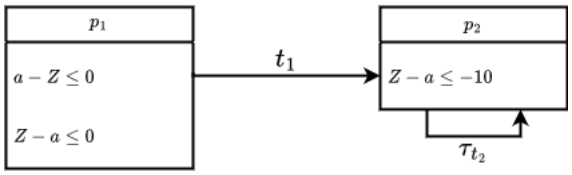


Figure 4.5: Constraint graph of the DPN depicted in Figure 4.3, showing the missing transition t_2

applies only to the transitions that are not silent.

Definition 4.3.1 is based on the reachability graph of a DPN. Felli et al. in [30] propose a variation applied to the constraint graph of a Data Petri Net, as expressed by the following Definition 4.3.2.

Definition 4.3.2 (Data-aware soundness)**.** Let $\mathcal{N}$ be a DPN with initial marking m_0, final marking M_F, and $\mathcal{CG}_\mathcal{N} = (N, n_0, A)$ its constraint graph. The constraint graph $\mathcal{CG}_\mathcal{N}$ is *data-aware sound* if and only if the following properties hold.

- ***P*$_1$.** For every reachable node $(M, C) \in N, \exists C'.(M, C) \xrightarrow{*} (M_F, C')$
- ***P*$_2$.** For every reachable node $(M, C) \in N, M \geq M_F \implies (M = M_F)$
- ***P*$_3$.** For every transition $t \in T$, there exist two reachable nodes (M_1, C_1) and (M_2, C_2) such that $(M_1, C_1) \xrightarrow{t} (M_2, C_2)$

The similarity between the two definitions is straightforward, as the conditions to be respected are the same but applied to a different structure. Nevertheless, it is crucial to remember that the reachability graph of a DPN may be infinite for cyclic DPNs, and the soundness verification would be unfeasible. By using the constraint graph, it is possible to obtain a finite state representation of the DPN instead, on top of which a suitable algorithm may verify that the DPN is data-aware sound. Such an algorithm is presented in the next section.

4.3.2 Verification Procedure

The soundness verification algorithm comprises the constraint graph construction and the soundness conditions' assurance, described by Algorithm 4.3. It uses three structures to store the various elements encountered during the execution: two sets hold the nodes and the arcs (N and A) that compose the constraint graph, while a queue Q collects the nodes that need to be expanded. First of all, it initializes the previous structures by adding the initial node n_0 to the set of nodes and the queue, imposing the starting point of the search (lines 1-6); the first node is composed of the DPN's initial marking, and the corresponding difference constraint set formed by the initialization of each variable.

At line 7, a while loop starts until no more expandable nodes exist. At each iteration, a node is extracted from the queue, and all the enabled transitions starting from it are considered in a for loop (lines 8-9). Each enabled transition produces a new marking and a new system of difference constraints, computed by adding the transition's guard to the previous one and verifying that it is still consistent (lines 10-11). If so, the algorithm performs a further check. If the new node's marking M' is greater than the marking M^* of a node with the same

difference constraint set found previously, then "*it means that at least one of the places of the DPN is unbounded, therefore that the DPN is certainly unsound*" [30]. In this case, *false* is returned immediately, as proceeding would not make sense (lines 12-13). On the other hand, if the condition is not met, the new node can be expanded in successive iterations (lines 14-16). After the algorithm treats the "*normal*" transition, it proceeds to handle the *silent* one. First, it verifies that the guard doesn't write any variable, and then it computes the new constraints set, obtained by adding the negated guard to the current transition (lines 17-18). The new node is considered only if the resulting set is consistent.

From line 23, the verification of the soundness conditions starts. It starts by checking the presence of any dead node that breaks the condition $\boldsymbol{P_1}$. In a constraint graph, the only nodes with no outgoing arcs are the ones with the final marking. Thus, if there is a node without a final marking and any outgoing arcs, it means that for some transition, the system will reach a situation from which it will not progress further. If such node exists, it returns *false*; otherwise, it proceeds to check the condition $\boldsymbol{P_3}$ by finding a transition not used by any arc previously computed. Instead, condition $\boldsymbol{P_2}$ is checked during the construction of the constraint graph (lines 12- 13).

Theorem 2 in [30] states that "$RG_{\mathcal{N}}$ *is data-aware sound iff* $CG_{\mathcal{N}}$ is data-aware sound". Consequently, it is sufficient to assert that the constraint graph satisfies the conditions of Definition 4.3.2 to affirm that the Data Petri Net is sound. Nevertheless, a false-positive case exists, showing a Data Petri Net with a constraint graph that satisfies all three conditions but is unsound. The following sections cover this peculiar case and the new procedure for its detection.

4.4 Satisfiability Modulo Theories (SMT)

The procedure proposed in this work as an addendum to the soundness verification, discussed in the next section, leverages the Satisfiability Modulo Theories (SMT). This section gives the reader a brief overview of SMT.

Satisfiability is an old and well-known problem in the theoretical computer science field, which aims to determine whether a formula expressing a constraint has a solution. *Boolean satisfiability*, also known as SAT, is one of the most familiar satisfiability problems, where the goal is to understand whether a formula using logical connectors over a set of Boolean variables can become *true* by assigning a *true/false* value to all the variables involved. The scientific community has dedicated extensive work to the SAT problem, its optimization, and its application to different areas [33][34]. To cite some of these, *Combinational Equivalence Checking*, whose focus is to check the equivalence of two circuits, *Automatic Test-Pattern Generation*, for detecting defects in fabricated integrated circuits, *Job Scheduling*, for deciding whether there exists a schedule such that the end-time of every task is less than or equal to a given maximum time, and finally *Haplotyping Inference in Bioinformatics*, consisting in deriving haplotype data from genotype data.

However, it is common that applications in some fields require defining the satisfiability problem using formulas composed of more expressive logic, such as first-order logic. Nevertheless, despite the advancement in developing general-purpose first-order theorem solvers, some formulas remain directly unsolvable. Barret et al. in [35] give the following justification:"*many applications require not general first-order satisfiability, but rather satisfiability with respect to some background theory, which fixes the interpretations of certain predicate and function symbols*". This situation led to the introduction of a new research field named *Satisfiability Modulo Theories*, or SMT, for short, interested in the satisfiability of formulas concerning a background theory.

Algorithm 4.3 Procedure for verifying the soundness of a DPN

Input: A DPN $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$ with final marking M_F

Output: *True* if $\mathcal{N}$ is sound, *false* otherwise

1: Let N and A be empty sets
2: Let Q be an empty queue
3: $C_0 :=$ `CanonicalForm`$(\bigcup_{v \in V}\{v = \alpha_I(v)\})$
4: $n_0 := (m_0, C_0)$ ▷ Initial node
5: `Push`(N, n_0)
6: `Enqueue`(Q, n_0)
7: **while** $Q \neq \emptyset$ **do**
8: $\quad (M, C) :=$ `Dequeue`(Q)
9: $\quad$ **for** $t \in T$ s.t. $M \xrightarrow{t} M'$ **do**
10: $\quad\quad C' := C \oplus guard(t)$
11: $\quad\quad$ **if** $C' \neq null$ **then**
12: $\quad\quad\quad$ **if** $\exists (M^*, C^*) \in N$ s.t. $M' > M^* \wedge C' = C^*$ **then**
13: $\quad\quad\quad\quad$ **return** *false* ▷ The net is unbounded
14: $\quad\quad\quad$ **if** $(M', C') \notin N$ **then** ▷ Add node if not present
15: $\quad\quad\quad\quad$ `Push`$(N, (M', C'))$
16: $\quad\quad\quad\quad$ `Enqueue`$(Q, (M', C'))$
17: $\quad\quad\quad$ `Push`$(A, ((M, C), t, (M', C')))$
18: $\quad\quad$ **if** $write(t) = \emptyset$ **then**
19: $\quad\quad\quad C'' := C \oplus \neg guard(t)$ ▷ Silent transition case
20: $\quad\quad\quad$ **if** $C'' \neq null$ **then**
21: $\quad\quad\quad\quad$ **if** $(M, C'') \notin N$ **then**
22: $\quad\quad\quad\quad\quad$ `Push`$(N, (M, C''))$
23: $\quad\quad\quad\quad\quad$ `Enqueue`$(Q, (M, C''))$
24: $\quad\quad\quad\quad$ `Push`$(A, ((M, C), \tau_t, (M, C')))$
25: **if** $\exists (M, C) \in N$ s.t. $(M, C) \xrightarrow{t} \wedge M \neq M_F, \forall t \in T$ **then**
26: $\quad$ **return** *false* ▷ A dead node exists
27: **if** $\exists t \in T$ s.t. $(M, C) \xrightarrow{t}, \forall (M, C) \in N$ **then**
28: $\quad$ **return** *false* ▷ Missing transitions
29: **return** *true*

It is widely known that the SAT problem is NP-complete and that first-order logic is undecidable. This high computational complexity makes constructing a procedure that can solve arbitrary SMT solvers unfeasible. Nevertheless, in recent years, the innovation of core algorithms and data structures, with the exploration of new heuristics and focus on implementation details, has brought tremendous progress in the scale of solvable problems. Modern SAT and SMT solvers can check formulas with hundreds of thousands of variables and millions of clauses.[36]. Many SMT solvers have been developed through the years like Z3 [37], Yices [38], MathSAT 5 [39], and SMTInterpol [40]. Moreover, an international initiative named SMT-LIB [41] was born around 2003 to facilitate the research and development of Satisfiability Modulo Theories by providing rigorous descriptions of background theories used in SMT systems and promoting standard input and output languages across the various SMT solvers.

SMT is a tool well suited to support constraint graphs, specifically for the difference constraints sets. One of the many theories available in SMT that relates more to this work is the *Difference arithmetic* theory. It is a fragment of the *Linear arithmetic* theory, where predicates are restricted to the form $x - y \leq c$, for some variables x, y, and a constant c. A system of difference constraints can be considered as a formula where each inequality is a predicate, and all link together with conjunctions. In a DPN, though, the constraints admit also strict ($<$) inequalities. Figure 4.6 shows how the system of difference constraints corresponds to a logical formula composed as a conjunction of predicates. Furthermore, the formula (and the constraints set as a consequence) admits a graphical representation (4.6c), showing the correlation between variables, values, and level of strictness ($<$ or $\leq$). Even if not used in this work, de Moura and Bjørner states in [36] that "*Conjunctions of difference arithmetic inequalities can be checked very efficiently for satisfiability by searching for negative cycles in weighted directed graphs.*"

4.5 Co-reachability Analysis

4.5.1 False-positive Sound DPN

The soundness verification procedure described by Algorithm 4.3 should return *true* if and only if the input DPN is data-aware sound, as proved by Theorem 1 in [30]. Nevertheless, there exists a case identified by the procedure as a sound DPN but which is actually unsound. Figure 4.7 illustrates the DPN, with the corresponding constraint graph, of such a case. It comprises three places and three transitions, with p_1 as the initial marking and p_3 as the final marking. The first transition assigns a new value to variable x, followed by a choice primitive. Transition t_2 is enabled and can fire, staying in the same place but writing a new value in y, greater than or equal to x. Transition t_3, instead, can be only enabled if the value of y is strictly lower than 10. The peculiarity of this Data Petri Net is the presence of a cycle in $p2$, where transition t_2 can indefinitely keep writing new values for y. The case that makes this DPN unsound is when variable y is initialized with a value greater than or equal to 10 or the first transition writes in x such value. For this last case, if transition t_3 fires, the execution reaches the final marking, but if t_2 fires, the cycle starts updating variable y indefinitely, creating a never-ending loop. Since y is given a value greater than or equal to x, transition t_3 cannot be enabled.

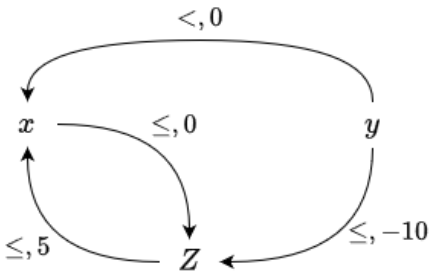
The constraint graph presented in Figure 4.7b, computed using the soundness verification procedure, respects all the conditions defined by Definition 4.3.2. Every non-final node has at least one outgoing arc using a non-

p_n
$Z - y \leq -10$
$x - y < 0$
$Z - x \leq 0$
$x - Z \leq 5$

(a) Constraint graph node

$$Z - y \leq -10 \wedge x - y < 0 \wedge Z - x \leq 0 \wedge x - Z \leq 5$$

(b) Logical formula of the difference constraints set of the node



(c) Graph representing the system of difference constraints

Figure 4.6: Correlation between a node of a constraint graph and Satisfiability Modulo Theories. The node depicted in Figure (a) has the corresponding formula composed as a conjunction of inequalities (b), which is further represented as graph (c).

silent transition, indicating the absence of dead nodes (condition $\boldsymbol{P_1}$). Condition $\boldsymbol{P_2}$ is satisfied because when the execution reaches the final marking, it does so without leaving tokens in other places of the net. Finally, for each transition t of the model, there exists an arc using t for connecting two nodes, thus satisfying $\boldsymbol{P_3}$. The constraint graph respects all the conditions that define a sound DPN. Still, as just described, the model shows a situation with a loop preventing the reach of the final marking (deadlock), creating, as a consequence, a false-positive case.

4.5.2 Co-reachability

The main problem with the previous model is that the execution, when firing transition t_2, is unaware of the condition on t_3, which imposes an upper bound on variable y. Transition t_2 breaks such bound because its guard writes a value in y that depends on variable x with no upper limit. An additional procedure is introduced to overcome this issue and correctly identify the DPN as an unsound model, described by Algorithm 4.4.

The core idea behind this new verification technique is to determine the valid variables' values that allow the execution to reach the final marking(s). To obtain such a goal, the concept of ***co-reachability*** played a central role in the development of the algorithm. It is usually used in the context of Finite State Machines (FSM) and automata, but it can be easily extended to constraint graphs, given the similarity with the directed graph used for automata representation. A node n is said to be *reachable* when there exists a path that connects the initial node n_0 to n; instead, it is said *co-reachable* when there exists a path that connects node n to a final node [42]. With these definitions in mind, instead of working in a forward direction from the starting node to the finals, there was a change of approach in favor of a backward analysis, starting from the final nodes down to the initial one. The values that respect the constraints of the final nodes are those defined by themselves. Consequently, the algorithm starts by considering only the difference constraints sets of the final nodes and iteratively builds the ones of the other nodes. During the computation, a node may contain multiple difference constraints sets.

The iterative procedure starts by marking as co-reachable only the nodes with a final marking:

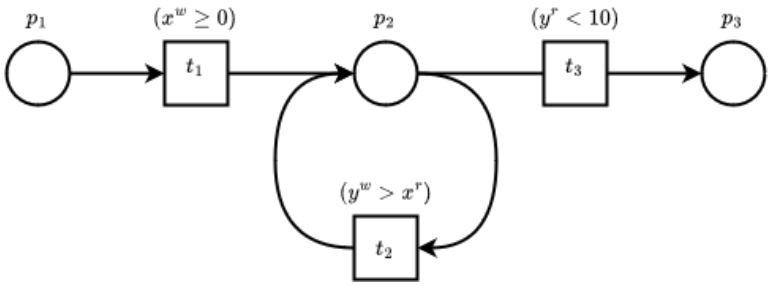
$$CoReach_0(M, C) = \begin{cases} \{C\} & \text{if } M = M_F \\ \emptyset & \text{otherwise} \end{cases} \tag{4.1}$$

Then, two cases are identified based on the type of transitions. The first one manages the nodes reached by a transition whose guard doesn't write any value or is silent.

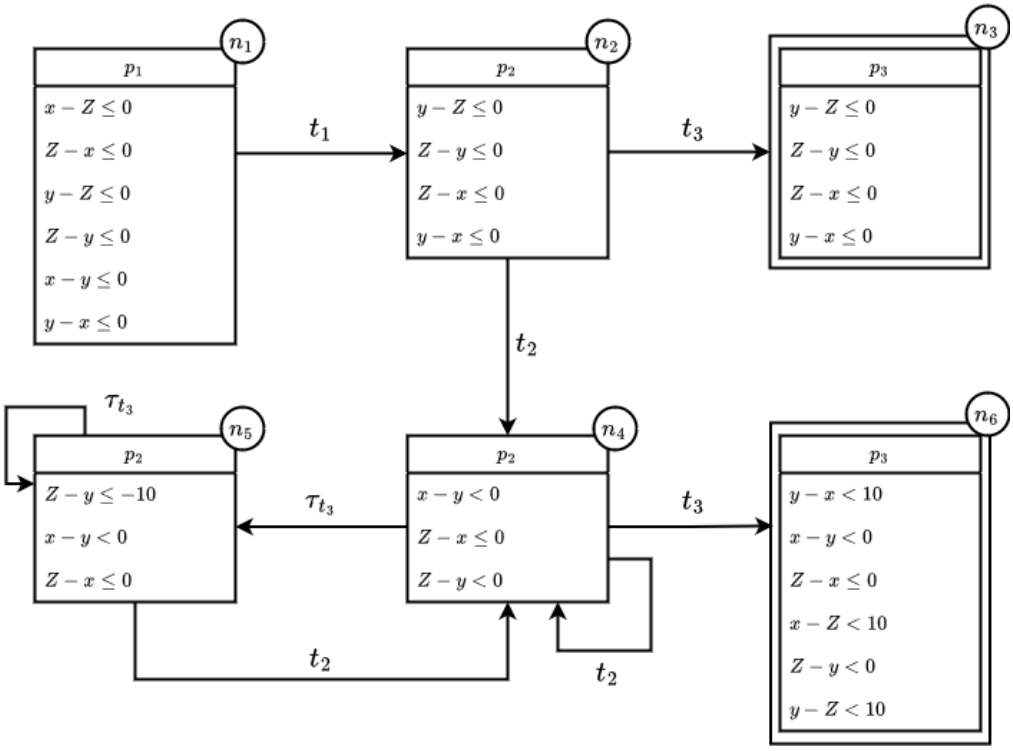
$$CoReach^r_{i+1}(M, C) = \bigcup_{(M,C)\xrightarrow{t}(M',C')} CoReach_i(M', C'), \text{ where } write(t) = \emptyset \text{ or } t \text{ silent} \tag{4.2}$$

Since the transition doesn't change any value and it is in place a backward analysis, the constraints found to be valid for the successive nodes, which allow reaching the final nodes, are also valid for the current one. Hence, a node (M, C)'s set at iteration $i + 1$ is the union between the node's set and all the sets of the successive nodes (M', C') computed at the previous iteration i. The second case considers the transitions whose guard contains at least a written variable.

$$CoReach^w_{i+1}(M, C) = \bigcup_{(M,C)\xrightarrow{t}(M',C')} (\exists write(t).CoReach_i(M', C') \cap C), \text{ where } write(t) \neq \emptyset \tag{4.3}$$



(a) Data Petri Net characterized by a cycle in p_2, with initial marking p_1 and final marking p_3



(b) Constraing graph of 4.7a

Figure 4.7: False-positive sound DPN verified using Algorithm 4.3

Since now the transition's guard writes at least a variable and modifies the values, it cannot be stated that the constraints of the subsequent nodes are also valid for this one. For this reason, the written variables are removed through quantifier elimination and re-introduced by intersecting the result with the original node's constraint set. In this case, the intersection may produce an inconsistent system, considered only if consistent.

The iterative computation produces for each node a set of systems of difference constraints defining the values that reach the final nodes, defined as follows:

$$CoReach_{i+1}(M, C) = CoReach_i(M, C) \cup CoReach'_{i+1}(M, C) \cup CoReach^{w}_{i+1}(M, C) \tag{4.4}$$

$$CoReach(M, C) = \bigcup_{i \geq 0} CoReach_i(M, C) \tag{4.5}$$

The following Lemma holds:

Lemma 1. *Let $\mathcal{N}$ be a DPN with final marking M_F, $\mathcal{CG}_\mathcal{N}$ the constraint graph of $\mathcal{N}$ and $\mathcal{RG}_\mathcal{N}$ the reachability graph of $\mathcal{N}$. Given a node (M, C) of $\mathcal{CG}_\mathcal{N}$, it holds that:*

$$\forall \alpha \in CoReach(M, C), \exists \alpha'.(M, \alpha) \xrightarrow{*} (M_F, \alpha')$$

Proof. Let $CoReach(M, C) = \{C_1, ..., C_n\}, \alpha \in C_i$ an assignment of values satisfying C and k the iteration when C_i is inserted in $CoReach(M, C)$ for the first time. Proof by induction on the iteration k.

Case $k = 0$: consider a difference constraints set C_i. By construction, C_i is inserted in $CoReach(M, C)$ at iteration 0 only if $M = M_F$. Since $M = M_F$, all the values α satisfying C make the property true. Thus, it holds that $\forall \alpha \in CoReach(M_F, C), \exists \alpha'.(M_F, \alpha) \xrightarrow{*} (M_F, \alpha')$ with $\alpha = \alpha'$.

Case $k + 1$: suppose that the property holds for all the difference constraints sets inserted at iteration k, that is, it holds that $\forall \alpha \in CoReach_k(M, C), \exists \alpha'.(M, \alpha) \xrightarrow{*} (M_F, \alpha')$. The goal is to prove that the property holds for all the sets inserted at iteration $k + 1$, that is, $\forall \alpha \in CoReach_{k+1}(M, C), \exists \alpha'.(M, \alpha) \xrightarrow{*} (M_F, \alpha')$. Consider a difference constraints set C_i. The set C_i is inserted at iteration $k + 1$ by applying either Rule 4.2 or Rule 4.3 on some $C'_i \in CoReach_k(M', C')$ such that $(M, C) \xrightarrow{t} (M', C')$ for some t. By inductive hypothesis, it holds that $\forall \alpha' \in CoReach_k(M', C'), \exists \alpha''.(M', \alpha') \xrightarrow{*} (M_F, \alpha'')$. There are two cases:

- $write(t) = \emptyset$: transition t is either a normal transition with read-only variables or silent (τ_t). Rule 4.2 applies. Since transition t does not change the value of variables, it holds that $(M, \alpha) \xrightarrow{t} (M', \alpha)$ with $\alpha \in CoReach_k(M', C')$. By inductive hypothesis, the property holds for $CoReach_k(M', C')$. Hence, there exists α' such that $(M, \alpha) \xrightarrow{t} (M', \alpha) \xrightarrow{*} (M_F, \alpha')$, proving that the property also holds for $CoReach_{k+1}(M, C)$.
- $write(t) \neq \emptyset$: Rule 4.3 applies. By construction, the constraint set C'_i obtained from $\exists write(t).CoReach_k(M', C') \cap C$ is considered only when consistent. Hence, there exists $\alpha' \in CoReach_k(M', C')$ such that $(M, \alpha) \xrightarrow{t} (M', \alpha')$. By inductive hypothesis, the property holds for $CoReach_k(M', C')$. Hence there exists α'' such that $(M, \alpha) \xrightarrow{t} (M', \alpha') \xrightarrow{*} (M_F, \alpha'')$, proving that the property also holds for $CoReach_{k+1}(M, C)$.

□

Once the procedure terminates, each node (M, C) has its *CoReach* computed. The last step is to verify that the sets in $CoReach(M, C)$ are equivalent to the original difference constraints set C. If they are not equivalent, it means that there exists an assignment of values satisfying C but for which it is not possible to reach the final marking, that is, $\exists \alpha \in C, \nexists \alpha'.(M, \alpha) \xrightarrow{*} (M_F, \alpha')$. It is the case of the example described in Section 4.5.1.

With the procedure explained above, the following theorem holds:

Theorem 2. *A DPN $\mathcal{N}$ is data-aware sound if and only if its constraint graph $\mathcal{GC}_\mathcal{N}$ is data-aware sound and for every node (M, C), $CoReach(M, C) \equiv C$.*

Proof. ($\Longrightarrow$) Assume that a DPN $\mathcal{N}$ is *data-aware sound*. Then, its constraint graph $\mathcal{GC}_\mathcal{N}$ is *data-aware sound* and for every node (M, C), $CoReach(M, C) = C$. This direction is proved by Felli et al. in [30]. Consequently, since the DPN $\mathcal{N}$ and its constraint graph are *data-aware sound*, it means that it does not exist an assignment of values such that from a node it is not possible to reach a final marking, thus proving also that for every node (M, C), $CoReach(M, C) \equiv C$.

($\Longleftarrow$) Assume that the constraint graph $\mathcal{GC}_\mathcal{N}$ of a DPN $\mathcal{N}$ is *data-aware sound* and that for every node (M, C) it holds $CoReach(M, C) \equiv C$. Then DPN $\mathcal{N}$ is *data-aware sound*. Consider property $\boldsymbol{P_1}$ of the data-aware soundness property. Since the constraint graph is *data-aware sound*, it means that for every reachable node (M, C), $\exists C'.(M, C) \xrightarrow{*} (M_F, C')$. Moreover, given that $CoReach(M, C) \equiv C$, by Lemma 1, it holds that $\forall \alpha \in C, \exists \alpha'$ such that $(M, \alpha) \xrightarrow{*} (M_F, \alpha')$. Consequently, it holds that for every reachable state (M, α), $\exists \alpha'.(M, \alpha) \xrightarrow{*} (M_F, \alpha')$, proving that the property $\boldsymbol{P_1}$ of Definition 4.3.1 also holds. With the same approach followed for $\boldsymbol{P_1}$, it is possible to prove also that the properties $\boldsymbol{P_2}$ and $\boldsymbol{P_3}$ holds, proving that the DPN is thus *data-aware sound*. □

4.5.3 Co-reachability procedure

The procedure for the co-reachability analysis described by Algorithm 4.4 works as follows. It takes in input a DPN and the corresponding constraint graph and returns *true* if, for all the nodes of the constraint graph, the sets of difference constraints computed are equivalent to the original one, *false* otherwise. From line 2 to line 8, it initializes the key-value map that will keep track of the sets of each node computed through the execution. It starts by considering only the difference constraints set of the final nodes, putting all the others empty. It continues in a while-loop until it reaches a fixed point when the sets found at the last iteration do not change from the previous one (line 10). At every iteration, it replaces the sets of each node with the output of the `ComputeCoReach` function (lines 13-14), which computes the constraints that allow reaching the final nodes, as described by the above Equations 4.2 and 4.3. Finally, it verifies that the set of systems of difference constraints obtained through backward analysis is equivalent to the original one for each node (lines 16-17). The equivalent operation between systems of difference constraints is performed using the Satisfiability Modulo Theories introduced in Section 4.4. Each set is converted into a logical formula, and thanks to an SMT solver, it tries to find an assignment of values satisfying one set and unsatisfying the other. If such an assignment does not exist, then the systems are equivalent, and if all the nodes respect this condition, the procedure terminates, returning *true*.

With this new procedure, it is possible to identify the DPN of Figure 4.7 as unsound. Table 4.1 gives a visual representation of the co-reachability analysis by showing for each node the constraint sets found through the different iterations of the while-loop. The rows are the iterations, and the columns are the nodes of the DPN's constraint graph (illustrated by Figure 4.7b). The original difference constraints are set as a reference near the

Algorithm 4.4 Procedure for computing $CoReach(M, C)$ and equivalence verification

Input: A DPN $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$ with final marking M_F and its Constraint Graph $\mathcal{CG}_\mathcal{N} = (N, n_0, A)$

Output: *true* if $CoReach(M, C) \equiv C$ for all the nodes, *false* otherwise

```
1:  procedure VerifyCoReach(𝒩, 𝒞𝒢_𝒩)
2:      Let S be an empty key-value map
3:      Let Finals be the set of nodes (M, C) ∈ N such that M = M_F
4:      for node := (M, C) ∈ N do                          ▷ Map initialization
5:          if node ∈ Finals then
6:              S[node] := {C}
7:          else
8:              S[node] := ∅
9:      Let S_prev be an empty key-value map
10:     while S ≠ S_prev do                  ▷ Continue until fixed point is reached
11:         S_prev := S
12:         S := ∅
13:         for node := (M, C) ∈ N do
14:             S[node] := ComputeCoReach(S_prev, node)
15:     for node := (M, C) ∈ N do
16:         if S[node] ≢ C then              ▷ Equivalence checked via SMT solver
17:             return false                                ▷ The net is unsound
18:     return true
19: procedure ComputeCoReach(S, node = (M, C))
20:     Res := S[node]
21:     RT := {(M', C') | ∃((M, C), t, (M', C')) ∈ A s.t. write(t) = ∅ or t silent}
22:     for node' ∈ RT do
23:         Res := Res ∪ S[node']
24:     WT := {(M', C') | ∃((M, C), t, (M', C')) ∈ A s.t. write(t) ≠ ∅}
25:     for node' ∈ WT do
26:         canonical := CanonicalForm(Exists(write(t), S[node']) ∩ C)
27:         if canonical ≠ null then
28:             Res := Res ∪ {canonical}
29:     return Res
```

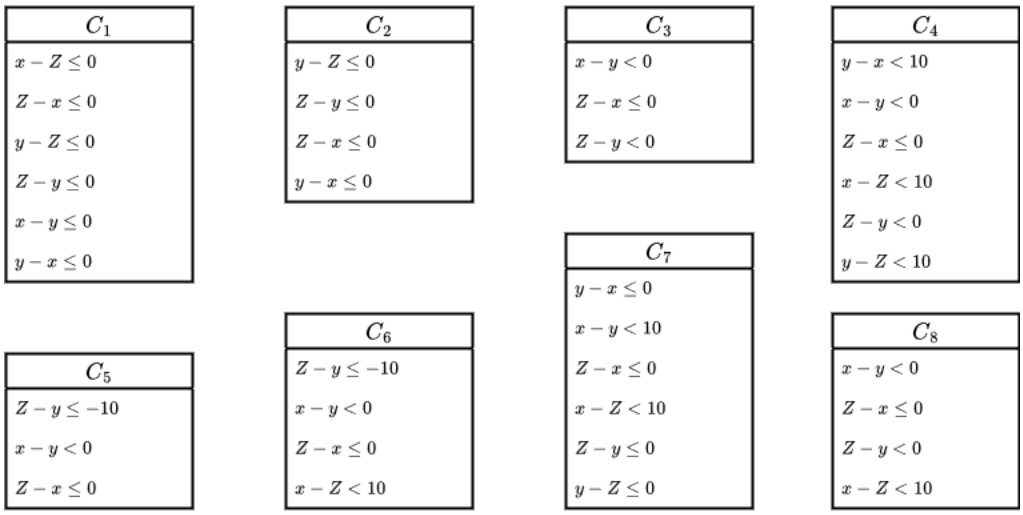


Figure 4.8: The systems of difference constraints found during the co-reachability analysis of the DPN model at Figure 4.7

	$n_1(C_1)$	$n_2(C_2)$	$n_3(C_2)$	$n_4(C_3)$	$n_5(C_5)$	$n_6(C_4)$
I_0	$\emptyset$	$\emptyset$	$\{C_2\}$	$\emptyset$	$\emptyset$	$\{C_4\}$
I_1	$\emptyset$	$\{C_2\}$	$\{C_2\}$	$\{C_4\}$	$\emptyset$	$\{C_4\}$
I_2	$\{C_1\}$	$\{C_2, C_7\}$	$\{C_2\}$	$\{C_4, C_8\}$	$\{C_6\}$	$\{C_4\}$
I_3	$\{C_1\}$	$\{C_2, C_7\}$	$\{C_2\}$	$\{C_4, C_8, C_6\}$	$\{C_6\}$	$\{C_4\}$
I_4	$\{C_1\}$	$\{C_2, C_7\}$	$\{C_2\}$	$\{C_4, C_8, C_6\}$	$\{C_6\}$	$\{C_4\}$

Table 4.1: Matrix representation of the co-reachability analysis of the model at Figure 4.7. The rows are the iterations performed by the while-loop, and the columns are the nodes of the constraint graph. Near the node, there is the original difference constraint set. Each cell shows the difference constraint sets computed through the execution.

node's name. Moreover, Figure 4.8 shows all the sets found during the execution so the reader can see and compare them. The initialized first nodes are n_3 and n_6 because they are final nodes. The next iteration is the turn of $n2$ since it is connected to n_3 by transition t_3, and n_4, linked to n_6 by the same. The execution keeps cycling until the fixed point is reached at iteration I_4. In the last step, for the nodes n_4 and n_5, it is detected that the sets computed by the procedure are not equivalent to the original ones. It can be easily seen by looking at the node n_5 since constraints C_5 and C_6 are almost the same, except that in C_6, there is the additional constraint $x - Z \leq 10$. Going backward from the final nodes allowed capturing an additional constraint missing from the constraint graph, adding an upper bound to variable x. Using Algorithm 4.3 and Algorithm 4.4, it is possible to identify such cases as unsound Data Petri Nets.

5

Data-aware Process Repair

Applying soundness verification techniques to Data Petri Nets helps to understand whether the underlying business process is well-formed. An additional tool for supporting designers in analyzing and optimizing workflows is the ability to automatically repair the conditions that prevent an unsound DPN from behaving correctly.

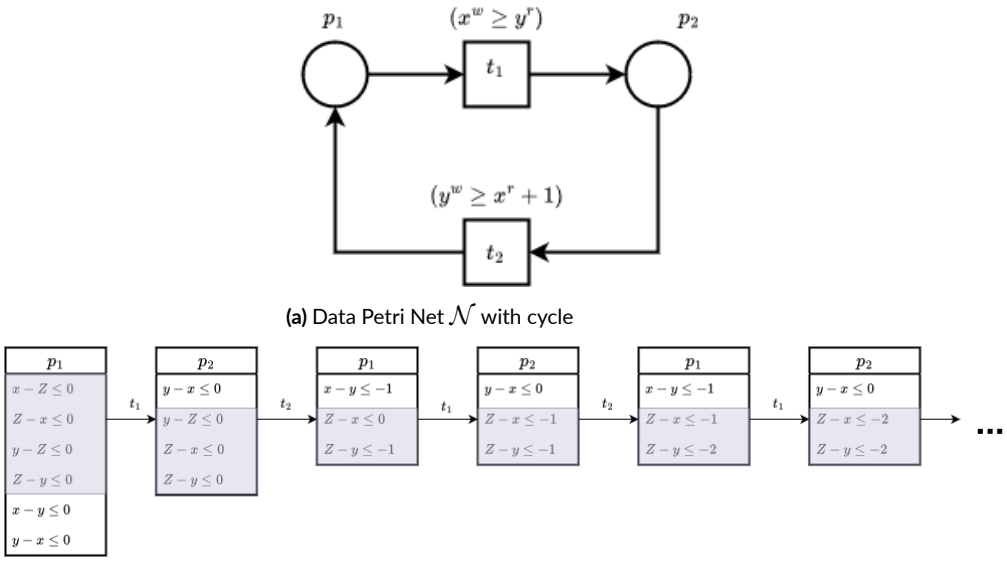
This chapter introduces a general algorithm for the automatic repair of data-aware processes. The first part defines what data-aware process repair means and the prerequisites a DPN must satisfy to apply such a procedure correctly. Then, two sections describe the cases of acyclic and cycle DPNs, respectively, and why two different procedures are needed to handle such cases. Finally, the last section covers the implementation details, the tests, and their evaluation.

5.1 Repair definition and prerequisites

Before the reparation procedure, it is crucial to define what data-aware process repair means and on what factors the algorithm works to achieve a sound Data Petri Net.

Definition 5.1.1 (Data-aware process repair [25]). Let $\mathcal{N} = \{P, T, I, O, m_0, V, \alpha_I, guard\}$ be a Data Petri Net. A *data-aware process repair* of $\mathcal{N}$ is a DPN $\mathcal{N}' = \{P', T', I', O', m_0', V', \alpha_I', guard'\}$ meeting the following conditions:

1. $\{P, T, I, O, m_0, V, \alpha_I\} = \{P', T', I', O', m_0', V', \alpha_I'\}$;
2. for each transition $t \in T$ such that $guard(t) = y - x \bowtie k$ for some $\bowtie, k$, either $guard'(t) = y - x \bowtie' k'$, or $guard'(t) = y - Z \bowtie' k'$ or $guard'(t) = Z - x \bowtie' k'$. Moreover, if $\mathcal{N}$ contains cycles and $guard'(t) = y - x \bowtie' k'$, then $k' = 0$;
3. $\mathcal{N}'$ is *data-aware sound*.



(a) Data Petri Net $\mathcal{N}$ with cycle

(b) Infinite constraint graph of $\mathcal{N}$. The upper bound of x and y keeps indefinitely growing

Figure 5.1: Example of a cyclic DPN whose constraint graph is infinite, where $m_0 = \{p_1\}, M_F = \{p_2\}, \alpha_I(x) = 0, \alpha_I(y) = 0,$

The cost of the repair of $\mathcal{N}'$ is the number of guards in $\mathcal{N}'$ that differs from $\mathcal{N}$

From the above definition, it is understood that the repair process works only on the data level by changing the guards of the transitions. The first condition, in fact, says that all the components of the DPN but the *guard* remain unchanged. This condition strictly correlates with one of the prerequisites the DPN must satisfy to use the repair procedure. The algorithm assumes that the DPN's unsoundness is caused only at the data level, with the underlying Petri Net's structure being sound. By taking the DPN without the guards, the Petri Net should be able to reach the final markings cleanly. The workflow should be fixed using existing state-of-the-art solutions if it is unsound.

The second condition defines the form the guards of the repaired net have, which can either be as difference constraints (for example, $x < y + k$) or unary constraints (for example, $x < k$). An additional constraint is set when the DPN has cycles. For such cases, general difference constraints $y - x \bowtie k$ are allowed only if $k = 0$. If a DPN has cycles, the constraint graph may be infinite. By imposing such a condition on the constraint's constant, the constraint graph of the DPN is finite [25]. Figure 5.1 illustrates a Data Petri Net with a cycle and a guard of the form $y - x \leq k$ where $k \neq 0$ (Fig. 5.1a) and the corresponding constraint graph (Fig. 5.1b). For such DPN, the constraint graph is infinite. Both variables are written and depend on each other's values. Nevertheless, what causes the constraint graph to expand indefinitely is transition t_2 because the writing in y of the value of x incremented by one keeps creating new constraint sets that generate new expandable nodes, making the algorithm unable to finish. Figure 5.2, instead, shows the constraint graph of the DPN in Figure 5.1a with transition t_2's guard modified to $y^w - x^r \geq 0$. The constraint graph is finite and does not expand indefinitely, with a loop between places p_1 and p_2.

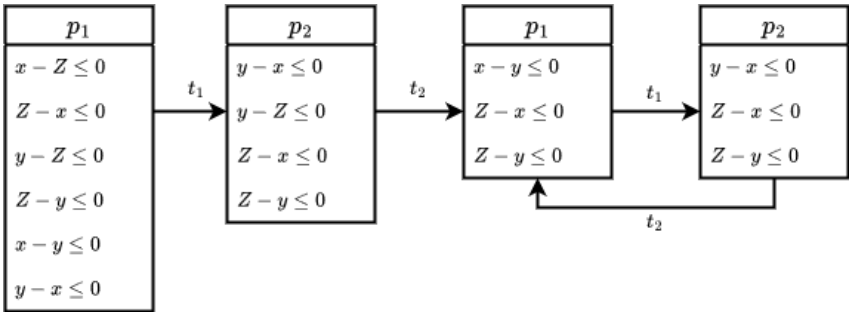


Figure 5.2: Constraint graph of the DPN in Figure 5.1a, with transition t_2's guard change to $y^w \geq x^r$

Finally, the third condition imposes that the repaired DPN has to be *data-aware sound*.

5.2 Acyclic DPNs

This section covers the repair of acyclic Data Petri Nets split into two parts in the Algorithms 5.1 and 5.2. Starting from the original DPN, the core idea is to try to modify the transition guards and verify whether the resulting DPN is sound. The main procedure is `DPNRepair`, taking in input a DPN $\mathcal{N}$ and returning a new one respecting the definition of soundness described in 4.3.1. It uses a priority queue $\mathcal{Q}$ to store the DPNs obtained through the execution. The number of guards different from the original one defines the priority of a DPN. The algorithm aims to make the lowest number of changes to obtain a sound DPN.

It initializes the priority queue with the original DPN, which has no change and thus has a priority equal to zero and starts the iteration process (lines 2-5). It extracts the DPN with the lowest priority at each step and verifies the soundness. If the DPN is sound, it exits the loop and returns the DPN (lines 8-9); otherwise, it executes the repair procedures `FixDead` and `FixMissing` (lines 10-11). The first one, `FixDead`, aims to repair the cases where the violation of the condition $\boldsymbol{P_1}$ of Definition 4.3.1 causes the unsoundness, that is, the presence of one or more dead nodes. The second one, `FixMissing`, instead, tries to repair DPNs that do not satisfy $\boldsymbol{P_3}$, namely the ones with one or more transitions missing from the constraint graph. Both procedures identify a set of transitions to work on, and for each, they take the guard and modify it according to the type of condition to solve. For each modified guard, the procedures push a new version of the DPN in the priority queue, increasing the pool of potential solutions.

5.2.1 Dead nodes

The presence of a dead node in the constraint graph means that after a sequence of transitions, the system reaches a situation from which it is unable to proceed forward. `FixDead` handles such cases. There are two possible approaches to repair such cases: enable a transition that is not firing from the dead node or prevent the execution from reaching it. The procedure `ForwardRepair`, described in the Algorithm 5.1, relates to the first approach. In contrast, the function `BackwardRepair`, exposed in the second part of the algorithm (Alg. 5.2), handles the second approach.

Algorithm 5.1 Algorithm for the repair of an acyclic DPN (part 1)

Input: A DPN $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$
Output: A sound DPN $\mathcal{N}' = (P, T, I, O, m_0, V, \alpha_I, guard')$ according to Definition 5.1.1

```
1:  procedure DPNRepair(𝒩)
2:      Let 𝒬 be a priority queue
3:      Enqueue(𝒬, 𝒩, 0)                                  ▷ Add 𝒩 with priority 0
4:      Let 𝒩′ be an empty DPN
5:      while true do
6:          𝒩′ := Dequeue(𝒬)                  ▷ Remove DPN with minimum priority
7:          Let 𝒞𝒢_𝒩′ be the constraint graph of 𝒩′
8:          if 𝒞𝒢_𝒩′ is data-aware sound then
9:              break
10:         FixDead(𝒩′, 𝒞𝒢_𝒩′)
11:         FixMissing(𝒩′, 𝒞𝒢_𝒩′)
12:     return 𝒩′                                          ▷ Sound DPN
13: procedure UpdateQ(𝒩′)
14:     if 𝒩′ has not been visited yet then
15:         Let p be the number of guards of 𝒩′ that differ from their original ones in 𝒩
16:         Enqueue(𝒬, 𝒩′, p)
17: procedure FixDead(𝒩′, 𝒞𝒢_𝒩′)
18:     Let (m_0, C_0) be the initial node of 𝒞𝒢_𝒩′
19:     for dead node (M, C) in 𝒞𝒢_𝒩′ do
20:         Let FW be the set of all non-silent transitions that can fire from M in the Petri Net
21:         for t ∈ FW do
22:             ForwardRepair(𝒩′, t, C)
23:         Let BW be the set of transitions in all paths (m_0, C_0) →* (M, C)
24:         for t ∈ BW do
25:             BackwardRepair(𝒩′, t, C)
26: procedure ForwardRepair(𝒩′, t, C)        ▷ "Replace with the same constraint in C"
27:     Let 𝒩″ := (P, T, I, O, m_0, V, α_I, guard″) be a copy of 𝒩′
28:     Let y − x ⋈ k be the guard of t
29:     Let y − x ⋈′ k′ be the corresponding constraint in C
30:     guard″(t) = y − x ⋈′ k′
31:     UpdateQ(𝒩″)
```

Algorithm 5.2 Algorithm for the repair of an acyclic DPN (part 2)

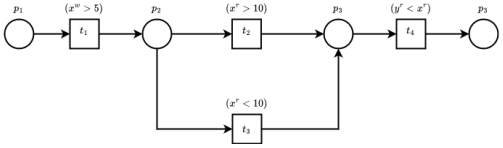
32: **procedure** BACKWARDREPAIR($\mathcal{N}', t, C$) ▷ "Replace with the opposite constraint in C"
33: Let $\mathcal{N}'' := (P, T, I, O, m_0, V, \alpha_I, guard'')$ be a copy of $\mathcal{N}'$
34: Let $y - x \bowtie k$ be the guard of t
35: **if** $x - y \bowtie' k' \in C \wedge k' \neq \infty$ **then**
36: **if** $\bowtie'$ *is* $\leq$ **then**
37: $guard''(t) = y - x < -k'$
38: **else**
39: $guard''(t) = y - x \leq -k'$
40: `UpdateQ`($\mathcal{N}''$)
41: **procedure** FIXMISSING($\mathcal{N}', \mathcal{CG}_{\mathcal{N}'}$)
42: Let (m_0, C_0) be the initial node of $\mathcal{CG}_{\mathcal{N}'}$
43: Let *Missing* be the set of missing transitions in $\mathcal{CG}_{\mathcal{N}'}$
44: **for** $t \in$ *Missing* **do**
45: Let *Nodes* be the set of nodes (M, C) of $\mathcal{CG}_{\mathcal{N}'}$ s.t. t can fire from M in the PN
46: **for** $(M, C) \in$ *Nodes* **do**
47: `ForwardRepair`($\mathcal{N}', t, C$)
48: Let BW be the set of non-silent transitions in all paths $(m_0, C_0) \xrightarrow{*} (M, C)$
49: **for** $t \in BW$ **do**
50: `BackForwardRepair`($\mathcal{N}', t, C$)
51: **procedure** BACKFORWARDREPAIR($\mathcal{N}', t, C$) ▷ "Make the guard *true*"
52: Let $\mathcal{N}'' := (P, T, I, O, m_0, V, \alpha_I, guard'')$ be a copy of $\mathcal{N}'$
53: Let $y - x \bowtie k$ be the guard of t
54: $guard''(t) = y - x \leq \infty$
55: `UpdateQ`($\mathcal{N}''$)

`ForwardRepair` procedure takes as input a DPN $\mathcal{N}$, a transition t and a difference constraint set C. It first duplicates the DPN and then considers the guard of the given transition (lines 27-28). It is crucial to recall the assumption that the underlying Petri Net is sound, and the unsoundness is present at the data level. Thus, if a transition cannot fire, it means that the firing of a transition creates a new inconsistent difference constraint set. The inconsistency happens because the guard of the transition is in contrast with a similar one already in the difference constraint set. Thus, a way to enable the transition is to take the constraint from the set C and put it as the guard of the transition (lines 29-30). This way, when the transition fires, it just adds the same constraint to the system, keeping it consistent and reaching a new node. Finally, the procedure `UpdateQ` takes the new DPN $\mathcal{N}'$ and adds it to the priority queue if it has not been visited previously (lines 14-16).

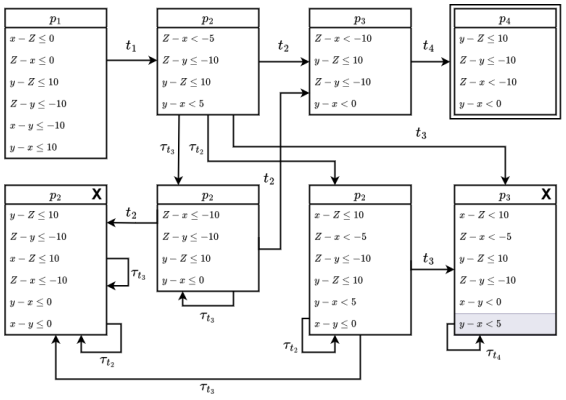
Figure 5.3 illustrates the repair process using only the `ForwardRepair` procedure. The DPN $\mathcal{N}'$ depicted in Figure 5.3a is unsound due to two dead nodes, as identified with a cross in the corresponding constraint graph on the right. The first iteration of `ForwardRepair` works on the rightmost dead node with the marking p_3, where transition t_4 cannot fire because it creates an inconsistent system. Transition t_4's guard is $y - x < 0$, replaced with the corresponding one in the constraint set, that is, $y - x < 5$. With this operation, transition t_4 can fire without producing inconsistencies, allowing the reaching of the final node, as shown in the constraint graph of Figure 5.3d. Nevertheless, the constraint graph is still unsound because of another dead node. Using the same approach, `ForwardRepair` replaces transition t_3's guard with the one in the constraint set, obtaining a sound DPN with a cost of 2.

`BackwardRepair` procedure takes the same arguments as `ForwardRepair`, duplicating the DPN and considering the transition's guard. The procedure aims to prevent the execution from reaching the dead node; thus, it takes all the transitions from the initial node to the dead one and replaces the guards with the opposite ones in the constraints sets. The constraint graph of the resulting DPN cannot reach the previous dead node because, by adding the opposite constraint, the node has an inconsistent set and is not considered. `ForwardRepair` alone is enough to repair a data-aware unsound DPNs, but the joint application of ForwardRepair and BackwardRepair in some cases allows finding solutions with smaller costs [25].

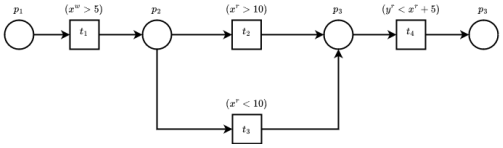
Figure 5.4 gives an example of such a case. The first modification is done by the execution of `ForwardRepair`($\mathcal{N}$, t_2, C), which changes the transition's guard to $x' \geq 5$ to remove the dead node. The resulting DPN $\mathcal{N}'$ is still unsound and thus requires further changes (note the dead node in the constraint graph of Figure 5.4d). On one side, if the algorithm uses only `ForwardRepair` procedure, it also changes transitions t_3 and t_4 since the guards need to allow the execution to proceed forward, reaching a cost of 3. Conversely, the execution of `BackwardRepair` allows changing lesser transitions. Take the constraint graph $\mathcal{CG}_{\mathcal{N}'}$ of DPN $\mathcal{N}'$ in Figure 5.4d and observe that dead node in place p_3. `BackwardRepair` works on all the transitions that reach the node currently considered; hence, the algorithm executes the call `BackwardRepair`($\mathcal{N}'$, t_2, C). The procedure takes t_2 guard's opposite constraint in the set, namely $x - Z < 10$, and sets the new guard as $Z - x \leq -10$. In the resulting DPN's constraint graph, $\mathcal{CG}_{\mathcal{N}''}$ in Figure 5.4f, the previous dead node is no longer present, as t_2's new guard made the set of constraints inconsistent. It is crucial to note that the transition being modified is still t_2, so the cost of the partial solution is only 1. Finally, the algorithm identifies the last dead node. It removes it by calling `ForwardRepair`($\mathcal{N}''$, t_5, C), obtaining a data-aware sound DPN $\mathcal{N}'''$ with a cost of 2 (Figure 5.4g).



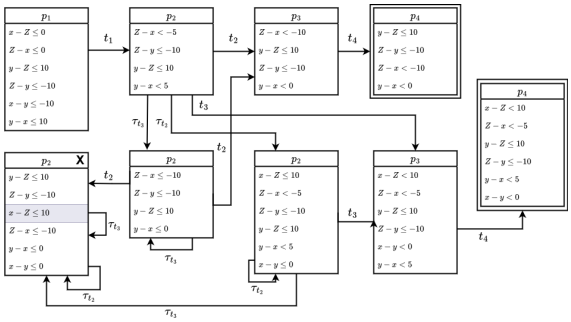
(a) Unsound DPN $\mathcal{N}$ with $m_0 = \{p_1\}, M_F = \{p_4\}$, $\alpha_I(x) = 0$, and $\alpha_I(y) = 10$,



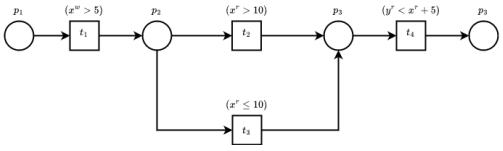
(b) Constraint graph $\mathcal{CG}_{\mathcal{N}}$ showing the unsoundness of DPN $\mathcal{N}$. There are two dead nodes, identified by the **X** symbol



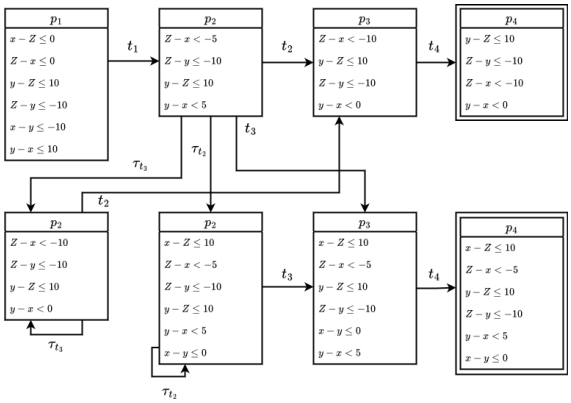
(c) ForwardRepair generates a new DPN $\mathcal{N}'$ from $\mathcal{N}$ with the guard of transition t_4 modified as $y^r < x^r + 5$



(d) Constraint graph $\mathcal{CG}_{\mathcal{N}'}$ of DPN $\mathcal{N}'$. The rightmost dead node is repaired, allowing the final marking p_4 to be reached. The DPN $\mathcal{N}'$ is still unsound due to the presence of another dead node

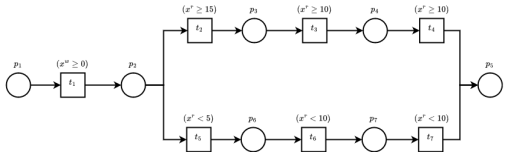


(e) ForwardRepair generates a new DPN $\mathcal{N}''$ from $\mathcal{N}'$ with the guard of transition t_3 modified as $x^r \leq 10$

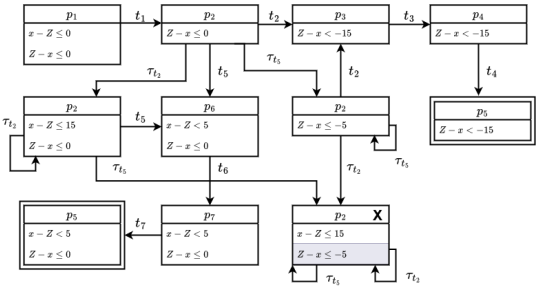


(f) Constraint graph $\mathcal{CG}_{\mathcal{N}''}$ of DPN $\mathcal{N}''$. All dead nodes are repaired, obtaining a sound DPN with two modifications

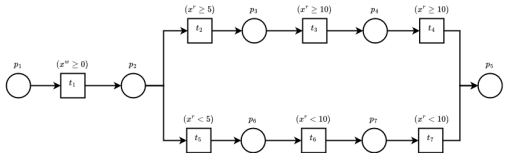
Figure 5.3: Example of ForwardRepair procedure application to an unsound DPN. A sound DPN is obtained by modifying two guards, thus with a cost of 2.



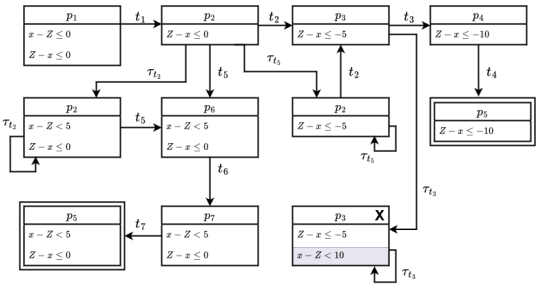
(a) Unsound DPN $\mathcal{N}$ with $m_0 = \{p_1\}, M_F = \{p_5\}$, and $\alpha_I(x) = 0$



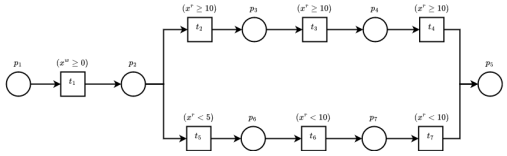
(b) Constraint graph $\mathcal{CG}_{\mathcal{N}}$ showing the unsoundness of DPN $\mathcal{N}$. There is a dead node, identified by the **X** symbol



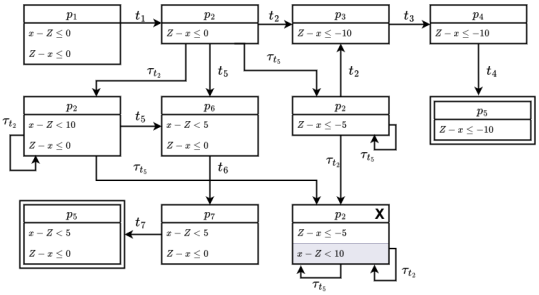
(c) `ForwardRepair` generates a new DPN $\mathcal{N}'$ from $\mathcal{N}$ with the guard of transition t_2 modified as $x^r \geq 5$



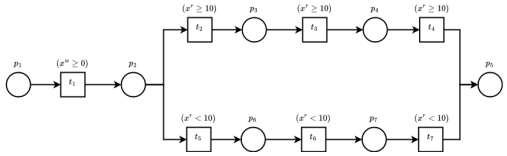
(d) Constraint graph $\mathcal{CG}_{\mathcal{N}'}$ of DPN $\mathcal{N}'$. The DPN is still unsound due to the presence of another dead node



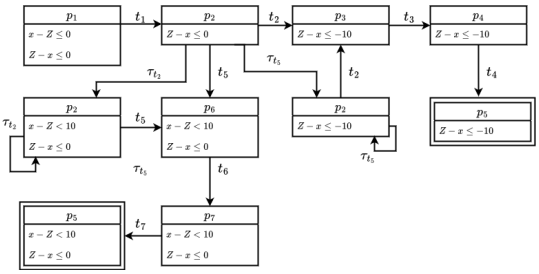
(e) `BackwardRepair` generates a new DPN $\mathcal{N}''$ by working again on transition t_2



(f) Constraint graph $\mathcal{CG}_{\mathcal{N}''}$ of DPN $\mathcal{N}''$. The DPN is still unsound due to the presence of another dead node

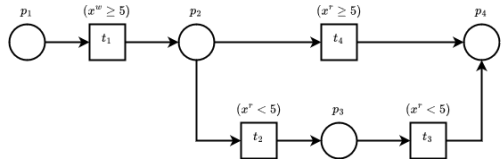


(g) `ForwardRepair` generates the last DPN $\mathcal{N}'''$ from $\mathcal{N}''$ with the guard of transition t_5 modified as $x^r < 10$

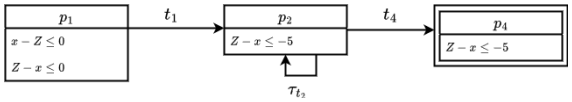


(h) Constraint graph $\mathcal{CG}_{\mathcal{N}''}$ of DPN $\mathcal{N}''$. All dead nodes are repaired, obtaining a sound DPN with three modifications but a cost of 2

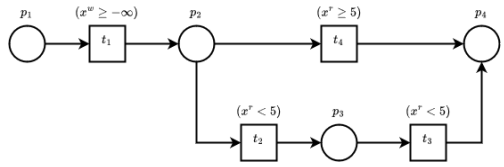
Figure 5.4: Example showing how the joint application of ForwardRepair and BackwardRepair can find a solution with a smaller cost than applying only ForwardRepair. If ForwardRepair is the only one applied, transitions t_3 and t_4 would be modified, obtaining a solution with cost 3. The joint application, instead, gives a solution by changing only two guards.



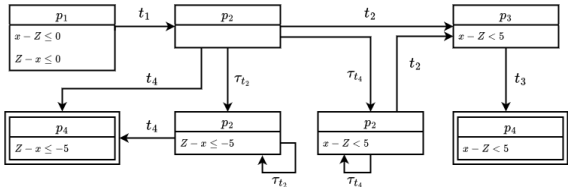
(a) Unsound DPN $\mathcal{N}$ with $m_0 = \{p_1\}, M_F = \{p_4\}$, and $\alpha_I(x) = 0$



(b) Constraint graph $\mathcal{CG}_{\mathcal{N}}$ showing the unsoundness of DPN $\mathcal{N}$. Transitions t_2 and t_3 are missing



(c) `BackForwardRepair` generates a new, sound DPN $\mathcal{N}'$ with transition t_1 modified



(d) Constraint graph $\mathcal{CG}_{\mathcal{N}'}$ showing previously missing transitions t_2 and t_3.

Figure 5.5: Example showing the repair process using the BackForwardRepair procedure

5.2.2 Missing transitions

The missing transition of a DPN from the corresponding constraint graph violates condition $\boldsymbol{P_3}$ of *data-aware soundness* definition. The repair utilizes the `FixMissing` procedure, described in the Algorithm 5.2. It first detects all the transitions that do not appear in the constraint graph and then, for each one, considers all the nodes from which the transition may fire in the underlying Petri Net (the net without data). The first repair approach is to enable transitions by `ForwardRepair` application. The second approach, instead, mixes the ones implemented by `ForwardRepair` and `BackwardRepair` in a new procedure named `BackForwardRepair`, which sets the guards as $y - x \leq \infty$. With this operation, the guards always evaluate to true, enabling the transitions.

Figure 5.5 illustrates an example where the repair of an unsound DPN returns a solution with a single change by applying `BackForwardRepair` procedure. The constraint graph of the net in Figure 5.5b misses transition t_2, and consequently, t_3. The node distinguished by place p_2 is one from which transition t_2 may fire in the underlying Petri Net. With the same fashion of `BackwardRepair`, all non-silent transitions reaching the node are considered, with t_1 being one of them. The replacement of the guard with $x^W \geq -\infty$ enables transition t_2 and t_3, as a consequence. The repair process terminates with a data-aware sound DPN $\mathcal{N}'$ with a cost of 1, as shown in Figures 5.5c and 5.5d.

In the above example, `BackForwardRepair` repairs the net by making the guard constantly evaluate to *true*. It is fundamental to keep in mind, though, that even if `BackForwardRepair` implements a naive solution, further applications of *ForwardRepair* and *BackwardRepair* on the resulting DPN may still process the guard of the transitions, thus possibly tighten the constraints in subsequent operations.

Theorem 3. *Let $\mathcal{N}$ be an acyclic DPN where the underlying dataless Petri Net is sound. Algorithm 5.1 terminates on $\mathcal{N}$ by returning a data-aware sound DPN.*

Proof. Let $\mathcal{N}$ be an acyclic DPN with the underlying Petri Net sound. Without self-loop silent transitions, the constraint graph of a DPN is a Directed Acyclic Graph. Since the set of transitions T is finite, and the underlying

Petri Net is bounded, the branching factor of each node in the constraint graph is bounded by $2 \cdot |T|$, composed of all the transitions plus the corresponding silent ones. Moreover, the length of a path starting from the initial node and terminating in a final one cannot exceed $|T|$. Consequently, the number of nodes in a constraint graph is at most $(2 \cdot |T|)^{|T|}$. Among the possible sequences of modifications executed by Algorithm 5.1, there is always one that uses `ForwardRepair` only. Such sequences can always be explored because all the constraint graphs built are finite, the possible modifications applied to a constraint graph are limited, and a breadth-first search strategy explores such changes. Consider a sequence that calls only `ForwardRepair`. Every time a new DPN is generated, the guard of a transition t is replaced with some constraint in the system of difference constraints of some node. Such a node can be either a dead node if the procedure is called inside `FixDead` or a node from which is missing a transition if the function is called inside `FixMissing`. In the first case, some paths starting from the initial node and ending with the silent transition τ_t are removed. Moreover, such paths are not introduced again by successive modifications because if t is processed again, the current guard $y - x \bowtie k$ is replaced with a weaker one $y - x \bowtie' k'$. In the second case, some paths are extended with transition t instead. By the same monotonicity argument on subsequent modifications of t, such extended paths are not removed by later applications of `ForwardRepair`. Since the number of paths in a constraint graph is finite, a sequence of `ForwardRepair` reaches a sound DPN in a finite number of steps. □

5.3 Cyclic DPNs

As already stated in Section 5.1, the constraint graph of a cyclic DPN may be infinite; thus, the repair process requires that the guards are either in the form $y - x \bowtie 0$ or at least one variable being Z (for example, $Z - x \leq 7$). The repair process seen previously is modified to consider this last condition. Algorithms 5.3 and 5.4 expose only the procedures modified for handling cyclic cases. The introduction of the *CoReach* verification and repair is particularly relevant. Due to cases like the one explained in Section 4.5.1, for cyclic DPNs, it is not enough to check only the conditions of Definition 4.3.2.

As seen from line 8 in Algorithm 5.3, `VerifyCoReach` procedure checks the additional property, as defined by Algorithm 4.4, after verifying the absence of dead nodes and missing transitions. `ForwarwdRepair` and `BackwardRepair` extend the previous approaches and consider the guards based on the form required for cyclic DPNs. For example, the *if* condition at line 18 of Algorithm 5.3 checks the form of the constraint. If the guard does not respect the form, for example, $y - x < 6$, the procedure does not consider it. Instead, it pushes in the priority queue two separate DPNs, using the corresponding unary constraints in the set (lines 22-28). `BackwardRepair` implements the same technique.

When `VerifyCoReach` fails, it means that some nodes have values for which the net cannot reach the final nodes. The repair process uses an additional procedure named `FixNotCoReach` (Algorithm 5.4) to fix such nodes. `FixNotCoReach` works by using an approach similar to `FixDead`, but with some adaptations. Instead of dead nodes, it considers all the nodes for which `VerifyCoReach` fails, that is, the nodes that have the difference constraints sets computed via co-reachability analysis not equivalent to the original one (line 48). Then, for each node, the procedure computes the transitions necessary for `ForwardRepair` and `BackwardRepair` executions (lines 51-52). The peculiarity relies on the constraints used for computing the repair operations. Instead of the set of constraints of the node, the procedure computes the subtraction between the original one and the ones calcu-

lated via co-reachability analysis (lines 54-55), using the operation described in Section 4.1. The reason to apply the subtraction is to try to tighten the original constraints using the ones allowing the reaching of the final nodes. Then, for each resulting constraints set, it executes `ForwardRepair` and `BackwardRepair` on the corresponding transitions FW and BW (lines 56-60). The process continues until it finds a sound solution.

Algorithm 5.3 Algorithm for the repair of a cyclic DPN (part 1)

Input: A DPN $\mathcal{N} = (P, T, I, O, m_0, V, \alpha_I, guard)$
Output: A sound DPN $\mathcal{N}' = (P, T, I, O, m_0, V, \alpha_I, guard')$ according to Definition 5.1.1

```
1:  procedure DPNRepair(𝒩)
2:      Let 𝒬 be a priority queue
3:      Enqueue(𝒬, 𝒩, 0)                                  ▷ Add 𝒩 with priority 0
4:      Let 𝒩′ be an empty DPN
5:      while true do
6:          𝒩′ := Dequeue(𝒬)                              ▷ Remove DPN with minimum priority
7:          Let 𝒞𝒢_𝒩′ be the constraint graph of 𝒩′
8:          if 𝒞𝒢_𝒩′ is data-aware sound AND VerifyCoReach(𝒩′, 𝒞𝒢_𝒩′) = true then
9:              break
10:         FixDead(𝒩′, 𝒞𝒢_𝒩′)
11:         FixMissing(𝒩′, 𝒞𝒢_𝒩′)
12:         FixNotCoReach(𝒩′, 𝒞𝒢_𝒩′)
13:     return 𝒩′                                         ▷ Sound DPN
14: procedure ForwardRepair(𝒩′, t, C)        ▷ "Replace with the same constraint in C"
15:     Let 𝒩″ := (P, T, I, O, m_0, V, α_I, guard″) be a copy of 𝒩′
16:     Let y − x ⋈ k be the guard of t
17:     if y − x ≤ k′ ∈ C s.t k′ = 0 or y is Z or x is Z then
18:         guard″(t) = y − x ≤ k′
19:         UpdateQ(𝒩″)
20:     else                                  ▷ k′ ≠ 0 and y is not Z and x is not Z
21:         Let y − Z ⋈′ k′ be the constraint in C
22:         guard″(t) = y − Z ⋈′ k′
23:         UpdateQ(𝒩″)
24:         Let 𝒩″ be a new copy of 𝒩′
25:         Let Z − x ⋈′ k′ be the constraint in C
26:         guard″(t) = Z − x ⋈′ k′
27:         UpdateQ(𝒩″)
```

The algorithm proposed repairs the unsound example described in Figure 4.7, finding a sound DPN with a cost of 1. The solution has the guard of transition t_2 changed to $x' \leq \infty$. By avoiding the writing of variable y, the DPN reaches correctly the final node p_3. It is interesting to notice how the solutions may change depending on the variables' initial values. Take the same example, but with $\alpha_I(x) = 0$ and $\alpha_I(y) = 10$, as shown in Figure

Algorithm 5.4 Algorithm for the repair of a cyclic DPN (part 2)

28: **procedure** BACKWARDREPAIR($\mathcal{N}', t, C$) ▷ "Replace with the opposite constraint in C"
29: Let $\mathcal{N}'' := (P, T, I, O, m_0, V, \alpha_I, guard'')$ be a copy of $\mathcal{N}'$
30: Let $y - x \bowtie k$ be the guard of t
31: **if** $x - y \bowtie' k' \in C$ s.t $k' = 0$ or y is Z or x is Z **then**
32: **if** $\bowtie'$ *is* $\leq$ **then** $guard''(t) = y - x < 0$
33: **else** $guard''(t) = y - x \leq 0$
34: `UpdateQ`($\mathcal{N}''$)
35: **else** ▷ $k' \neq 0$ and y is not Z and x is not Z
36: **if** $x - Z \bowtie' k' \in C$ s.t $k' \neq \infty$ **then**
37: **if** $\bowtie'$ *is* $\leq$ **then** $guard''(t) = Z - x < k'$
38: **else** $guard''(t) = Z - x \leq k'$
39: `UpdateQ`($\mathcal{N}''$)
40: Let $\mathcal{N}''$ be a new copy of $\mathcal{N}'$
41: **if** $Z - y \bowtie' k' \in C$ s.t $k' \neq \infty$ **then**
42: **if** $\bowtie'$ *is* $\leq$ **then** $guard''(t) = y - Z < k'$
43: **else** $guard''(t) = y - Z \leq k'$
44: `UpdateQ`($\mathcal{N}''$)
45: **procedure** FIXNOTCOREACH($\mathcal{N}', \mathcal{CG}_{\mathcal{N}'}$)
46: Let (m_0, C_0) be the initial node of $\mathcal{CG}_{\mathcal{N}'}$
47: Let *Nodes* be the set of nodes failing the co-reachability analysis
48: **for** $n := (M, C) \in Nodes$ **do**
49: Let S_n be the the set of systems of difference constraints of n computed via co-reachability analysis
50: Let FW be the set of all non-silent transitions that can fire from M in the Petri Net
51: Let BW be the set of non-silent transitions in all paths $(m_0, C_0) \xrightarrow{*} (M, C)$
52: Let Diff be an empty set
53: **for** $C' \in S_n$ **do** ▷ Difference Bounded Matrix Subtraction
54: Let $Sub_{C'}$ be the set of consistent systems resulting from $C - C'$
55: Diff := Diff $\cup$ $Sub_{C'}$
56: **for** $C' \in$ Diff **do**
57: **for** $t \in FW$ **do**
58: `ForwardRepair`($\mathcal{N}', t, C'$)
59: **for** $t \in BW$ **do**
60: `BackwardRepair`($\mathcal{N}', t, C'$)
61:

5.6. In this case, the previous solution is still unsound, due to the initial value of y not enabling transition t_3. The repair process detects the unsoundness of the last DPN and proceeds with other modifications. The algorithm terminates with a solution consisting of the guard of transition t_2 updated as $y^w \geq 0$.

Theorem 4. *Let $\mathcal{N}$ be a DPN with the underlying dataless Petri Net sound. Algorithm 5.1 with the modifications of Algorithm 5.3 terminates on $\mathcal{N}$ by returning a data-aware sound DPN.*

Proof. The proof follows the style of Theorem 3. Consider the path that applies `ForwardRepair` calls only. At every step of the path, the guard of a transition is replaced with some constraint in the difference constraints set of some node, either a dead node if the procedure is called inside `FixDead`, a node from which is missing a transition if it is called from `FixMissing`, or a node whose constraints systems computed via co-reachability analysis are different from the original one if it is called inside `FixNotCoReach`. Since every node in the constraint graph is the canonical representation of a set of constraints taken from a finite universe, it implies that the number of possible modifications is finite, too. Moreover, if a transition is processed again, the guard is replaced with a weaker one. In the worst case, all the guards are made true (for example, $y - x \leq \infty$) and soundness of the DPN follows from the assumption that the underlying dataless Petri Net is sound. In every case, the algorithm terminates in a finite number of steps.

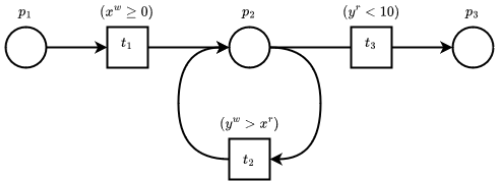
□

5.4 Tests and evaluation

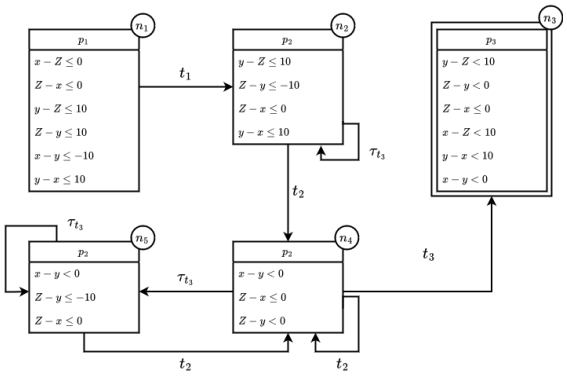
This work comprise a concrete implementation of the algorithms proposed previously to verify their behavior from a practical point of view. The following section covers the details of such implementation, followed by the tests on some selected case studies taken from the literature. All the material is publicly accessible on Github at [43].

The implementation is written in Java. The choice of this language is to make the code available as a potential plug-in for "*ProM*", an extensible framework that supports a wide variety of process mining techniques[44]. It is divided mainly into two parts, the parsing part and the repair part. The parsing part covers all the classes and structures utilized for correctly representing a Data Petri Net. A Data Petri Net is modeled using the Petri Net Markup Language (PNML), an XML-based format thanks to which it is possible to define the different elements forming a Petri Net with Data (places, transitions, guards, arcs, etc...). The repair part, instead, covers the structures and methods utilized for the implementation of the algorithms seen through this work, that is, the *difference constraints set*, the *constraint graph*, the *acyclic* and *cyclic repair*, and so on. All the code is written from scratch, except for the SMT solver, for which an external package was used [45].

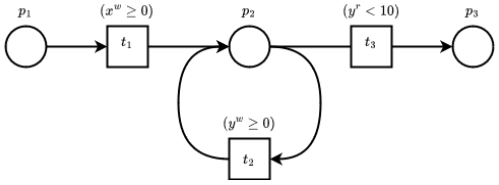
The tests consist of taking in input a Data Petri Net, verifying its unsoundness, and repairing it if unsound. The tests' goal is to show the correct repair of an unsound DPN, plus some statistical information about the repair process. Table 5.1 shows a summary of the execution details for each test case. For each DPN, besides the number of places, transitions, and nodes of the constraint graph, there is the amount of time required for the repair, the distance from the original DPN, that is, the number of guards changed, and finally, the number of DPNs analyzed through the process. The lettering A/C indicates whether the DPN is acyclic (A) or cyclic (C)



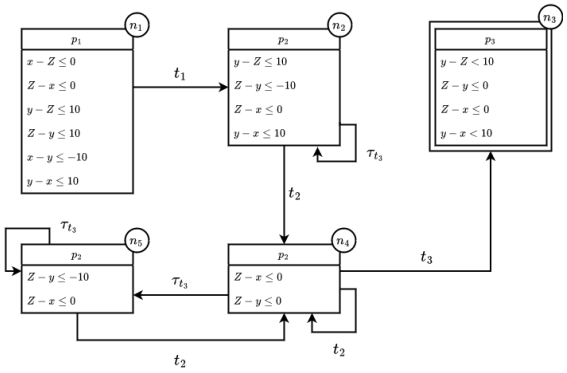
(a) Unsound DPN $\mathcal{N}$ with $m_0 = \{p_1\}, M_F = \{p_3\}$, $\alpha_I(x) = 0$, and $\alpha_I(y) = 10$



(b) Constraint graph $\mathcal{CG}_{\mathcal{N}}$. It is *data-aware sound* but `VerifyCoReach` returns *false*



(c) DPN $\mathcal{N}'$ generated from $\mathcal{N}$, with transition t_2 updated



(d) Constraint graph $\mathcal{CG}_{\mathcal{N}'}$. Now the net is *data-aware* sound and `VerifyCoReach` returns *true*

Figure 5.6: Example showing the repair of the DPN of Figure 4.7, but with different initial values

DPN (A/C)	# Places	# Trans.	# CG Nodes	Exec.Time (s)	Distance	Iterations
1-Figure 5.3 (A)	4	4	16	0.07s	2	21
2-Figure 5.4 (A)	7	7	18	0.07s	2	24
3-Figure 5.5 (A)	4	4	3	0.04s	1	3
4-Figure 4.7 (C)	4	4	3	0.35s	1	3
5-Exp. Growth 2 (A)	9	11	16	0.11s	4	42
6-Exp. Growth 3 (A)	17	23	32	35s	8	69282
7-Exp. Growth 4 (A)*	33	47	64	28s	16	13258
8-Credit Loan (C)*	12	12	62	8s	4	2180
9-Package Handling-Half (A)*	25	32	295	-	-	-
10-Package Handling (A)*	25	38	11229	-	-	-

Table 5.1: Evaluation summary of the repair process on DPNs of different sizes

since the repair process slightly changes between the two cases. Finally, the asterisk symbol (*) near the DPN's name specifies an additional constraint on the size of the priority queue.

DPNs from 1 to 3 are taken from [25], the ones from 5 to 7 and 9-10 are taken from [46], a list of Data Petri Nets used as a supporting material in [47] for a similar goal of this work, while the number 8 is taken from [30]. Beside the first four examples, the other ones were not directly usable from the code, mainly because the current implementation does not support guards composed of conjunction or disjunction of multiple conditions. Thus, there were applied some changes on the guards. The tests are executed on a PC with Intel(R) Core(TM) i7, 6 cores at 2.60GHz and 16 GB of RAM.

The results show that the first five examples, characterized by fewer places and transitions, are solved almost immediately with a few changes. Starting from the number 6 and above, the repair process requires much more time, and the number of iterations increases drastically. The cases marked with the asterisk have a limitation on the priority queue, that is, after reaching a fixed limit (1000, for these experiments) no more DPNs were added, thus forcing the process to consume a DPN before exploring further ones. In fact, without this limit, the repair of DPNs from number 7 to 10 was not able to finish within 5 minutes, due to unsound DPNs being explored and filling the queue. On the other side, preventing the priority queue from indefinitely growing allowed repairing DPNs 7 and 8 within 30 seconds. Nevertheless, the repair of the last two DPNs did not terminate within the time interval. The two refer to the same DPN, but the "half" one is modified to reduce the number of constraint graph nodes (295 compared to the 11229). It is important to highlight that, for the last case, the construction of the constraint graph requires much time ($>$ 20 seconds), which consequently affects the overall repair process.

The results show that the proposed solutions work correctly in repairing unsound DPNs, but the exponential complexity heavily affects the performance when the DPN's size increases. Moreover, as previously stated throughout this work, some algorithms are implemented using a naive approach, since this work goal was focused more on the correctness of the repair process rather than the performance, consequently leaving room for improvements.

6

Concluding Remarks

In the first part, this work provided an overview of existing solutions in the literature concerning Data-aware soundness verification of Data Petri Nets, for which false-positive cases exist. Thanks to a detailed investigation of the causes, it was possible to elaborate a patch of the previous approaches to identify these unsound Data Petri Nets correctly. Applying a co-reachability analysis allowed the detection of the nodes of the constraint graph for which the constraint set was consistent but it was impossible to reach the final nodes for some assignment of values. In the second part, the work focused on repairing unsound DPNs, where a sound solution was obtained by keeping the states and the transitions of the underlying Petri Net intact and modifying only the constraints of the guards. Two different algorithms were shown to handle acyclic and cyclic DPNs. For both algorithms, though, the basic assumption is that the underlying data-less Petri Net must be sound (that is, on the workflow level). Due to the presence of cycles and possibly an infinite constraint graph, special conditions were imposed on the guards of cyclic DPNs to restrict them only to variable-to-constant and variable-to-variable constraints. In this way, the resulting constraint graph is finite. Multiple examples were described to show how the combination of a forward and backward analysis could optimize the repair process and provide a solution with fewer changes.

The following work provides multiple points for future developments. On the soundness verification part, the procedure responsible for the co-reachability computation can be optimized by computing at each step a single system of difference constraints instead of a set of them, thus reducing the pool of sets used by the repair algorithms. On the repair process part, instead, there are currently two separate algorithms to handle cyclic and acyclic DPNs. An improvement is developing a generalized version of the algorithms to tackle both cases so that the restrictions on cyclic DPN constraints are applied only to the guards involved in cycles. The subtraction between sets of difference constraints in this work follows a naive implementation, where different sets may overlap each other; thus, it can be optimized by introducing a *disjoint subtraction* algorithm, aiming to obtain a union of disjoint sets, without redundant points between them. Furthermore, since the algorithms assume that the underlying Petri Net is sound, the repair process can also be extended to cover those DPNs with the unsoundness present at the workflow level. Finally, the repair algorithms can be tested on more complex cases involving a higher number of

nodes in the constraint graph to verify how the execution time and performance behave.

References

[1] J. Clempner, "Verifying soundness of business processes: A decision process petri nets approach," *Expert Systems with Applications*, vol. 41, no. 11, pp. 5030–5040, 2014.

[2] Merriam-Webster, "Process," 10 2023. [Online]. Available: https://www.merriam-webster.com/

[3] A. Abijith, S. Fosso Wamba, and D. Gnanzou, "A literature review on business process management, business process reengineering, and business process innovation," vol. 153, 06 2013.

[4] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers *et al.*, *Fundamentals of business process management*. Springer, 2018, vol. 2.

[5] A. ter Hofstede, W. M. van der Aalst, A. H. ter Hofstede, and M. Weske, "Business process management: A survey," in *Business Process Management: International Conference, BPM 2003 Eindhoven, The Netherlands, June 26–27, 2003 Proceedings 1*. Springer, 2003, pp. 1–12.

[6] B. Curtis, M. I. Kellner, and J. Over, "Process modeling," *Commun. ACM*, vol. 35, no. 9, p. 75–90, sep 1992. [Online]. Available: https://doi.org/10.1145/130994.130998

[7] J. Recker, M. Rosemann, M. Indulska, and P. Green, "Business process modeling-a comparative analysis," *Journal of the association for information systems*, vol. 10, no. 4, p. 1, 2009.

[8] M. Mohammadi, "Combination of modeling techniques for supporting business process architecture layers," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 7, p. 1038, 06 2017.

[9] K. T. Phalp, "The cap framework for business process modelling," *Information and Software Technology*, vol. 40, no. 13, pp. 731–744, 1998.

[10] W. M. Van der Aalst, "Formalization and verification of event-driven process chains," *Information and Software technology*, vol. 41, no. 10, pp. 639–650, 1999.

[11] G. Engels, A. Förster, R. Heckel, and S. Thöne, "Process modeling using uml," in *Process-Aware Information Systems*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:5465981

[12] G. Booch, I. Jacobson, J. Rumbaugh *et al.*, "The unified modeling language," *Unix Review*, vol. 14, no. 13, p. 5, 1996.

[13] G. Decker, R. Dijkman, M. Dumas, and L. García-Bañuelos, "The business process modeling notation," in *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009, pp. 347–368.

[14] "Business process model and notation," https://www.omg.org/spec/BPMN/2.0/, publication date: December 2010.

[15] M. Chinosi and A. Trombetta, "Bpmn: An introduction to the standard," *Computer Standards and Interfaces*, vol. 34, no. 1, pp. 124–134, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548911000766

[16] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in bpmn," *Information and Software technology*, vol. 50, no. 12, pp. 1281–1294, 2008.

[17] J. Wang, "Petri nets for dynamic event-driven system modeling," *Handbook of Dynamic System Modeling*, 01 2007.

[18] C. A. Petri, "Kommunikation mit automaten," 1962.

[19] E. Best and J. Desel, "Partial order behaviour and structure of petri nets," *Form. Asp. Comput.*, vol. 2, no. 1, p. 123–138, mar 1990. [Online]. Available: https://doi.org/10.1007/BF01888220

[20] J. L. Peterson, *Petri Net Theory And The Modeling Of Systems*. Prentice-Hall, 1981.

[21] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[22] C. Zhong, W. He, Z. Li, N. Wu, and T. Qu, "Deadlock analysis and control using petri net decomposition techniques," *Information Sciences*, vol. 482, pp. 440–456, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025519300416

[23] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer, 2008.

[24] B. Choi, "Petri net approaches for modeling, controlling, and validating flexible manufacturing systems," 1994.

[25] M. Zavatteri, D. Bresolin, and M. de Leoni, "Repair of unsound data-aware process models," *Proceedings of First International Workshop on Formal Methods for Business Process Management (FM-BPM 2023)*, 2023.

[26] J. P. Fishburn, "Solving a system of difference constraints with variables restricted to a finite set," *Information Processing Letters*, vol. 82, no. 3, pp. 143–144, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019001002678

[27] K. Quaas, M. Shirmohammadi, and J. Worrell, "Revisiting reachability in timed automata," in *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017, pp. 1–12.

[28] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, "Model checking. cyber physical systems series," 2018.

[29] A. David, J. Håkansson, K. G. Larsen, and P. Pettersson, "Model checking timed automata with priorities using dbm subtraction," in *Formal Modeling and Analysis of Timed Systems: 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006. Proceedings 4*. Springer, 2006, pp. 128–142.

[30] P. Felli, M. de Leoni, and M. Montali, "Soundness verification of decision-aware process models with variable-to-variable conditions," in *2019 19th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2019, pp. 82–91.

[31] W. M. Van der Aalst, "The application of petri nets to workflow management," *Journal of circuits, systems, and computers*, vol. 8, no. 01, pp. 21–66, 1998.

[32] M. De Leoni, P. Felli, and M. Montali, "A holistic approach for soundness verification of decision-aware process models," in *Conceptual Modeling: 37th International Conference, ER 2018, Xi'an, China, October 22–25, 2018, Proceedings 37*. Springer, 2018, pp. 219–235.

[33] J. Marques-Silva, "Practical applications of boolean satisfiability," in *2008 9th International Workshop on Discrete Event Systems*, 2008, pp. 74–80.

[34] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, p. 69–77, sep 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[35] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability: Second Edition*, 2021. [Online]. Available: https://escholarship.org/uc/item/11n7z852

[36] L. De Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Brazilian Symposium on Formal Methods*. Springer, 2009, pp. 23–36.

[37] ——, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[38] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[39] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 93–107.

[40] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: An interpolating smt solver," in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 248–254.

[41] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[42] M. Goorden, J. van de Mortel-Fronczak, M. Reniers, M. Fabian, W. Fokkink, and J. Rooda, "Model properties for efficient synthesis of nonblocking modular supervisors," *Control Engineering Practice*, vol. 112, p. 104830, 2021.

[43] "dpn-repair," https://github.com/aureliomakaj/dpn-repair.

[44] "Prom," https://promtools.org/.

[45] "java-smt," https://github.com/sosy-lab/java-smt.

[46] "Data petri nets," https://soundness.adatool.dev/repair.html.

[47] P. Felli, M. Montali, and S. Winkler, "Repairing soundness properties in data-aware processes," in *2023 5th International Conference on Process Mining (ICPM)*. IEEE, 2023, pp. 41–48.

Acknowledgments

I would like to say thank you firstly to my supervisor Prof. Davide Bresolin and co-supervisor Dr. Matteo Zavatteri for guiding and supporting me in the writing of this thesis. I want to thank them for all the time they have dedicated to revising the work, the suggestions, and the improvements both on the technical and non-technical side. Special and heartfelt thanks are dedicated to my girlfriend Arianna for always believing in my capacities and for staying by my side during all the weekends dedicated to studying in the last two years. Without her support, I do not think I would have been able to conclude this path in a reasonable time. Finally, I would like to also thank my family, friends, co-workers, and every person who helped me reach this important goal, marking the end of a journey and the beginning of an adventure.