Verification of Inter-Dependent Interfaces in Component-Based Architectures

Bart-Jan Hilbrands
bj.hilbrands@gmail.com

Academic supervisor:	Benny Åkesson, k.b.akesson@uva.nl
Daily supervisor:	Debjyoti Bera, debjyoti.bera@tno.nl
Host organisation/Research group:	ESI (TNO), https://esi.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering

Acknowledgements

I first of all would like to thank my supervisors Benny and Deb. It was a long half year, during which I received a lot of support. I really appreciated the weekly meetings as well as your overall involvement, which made sure that I was never stuck and could keep making progress.

I would also like to thank Tobias Bachmann and Elvin Alberts, with whom I worked together for most of the projects throughout the master. From the record breaking 11 hour teams call during SSVT, to spending an evening writing reports, we eventually always got the job done. It became a tradition to have drinks at the Polder on the Monday afternoon after a long day of Requirements Engineering, and this helped us all overcome this last hurdle.

Abstract

To manage the complexity of systems, they can be developed as component-based systems. Here, the system is divided into modular, reusable components that abstract externally visible behaviour using interfaces. Components can then act as servers, and offer services through these interfaces. These services can in turn be used by clients, or even other components part of the same system. Component-based systems can be modeled in the Component Modeling and Analysis (ComMA) framework. ComMA interfaces describe both structure and behaviour, where the behaviour of an interface is specified using protocol state machines. The big challenge with component-based systems, is that their concurrent nature makes it hard to verify their behaviour. When changing the behaviour of an interface, how does this affect any clients using the interface? It is entirely possible that the update caused the behaviour of the interface to become incompatible with its client interfaces, causing deadlock and unbounded behaviour.

Given a ComMA component specification, the verification of interface compliance through trace analysis is already possible. This is not formal verification however, and to formally verify properties like deadlock-freedom and boundedness, ComMA models are translated to Petri nets. Using Petri nets, a mathematical modeling language, two interfaces can be verified to be compatible, implying the absence of deadlock and unbounded behaviour. However, this translation was not complete, as constraints defined on interfaces could not be modeled as a Petri net yet. Such constraints could be that an interface can only transition to another state if another interface is in a certain state.

In this thesis, three of these constraints are categorized. Then, for each of these constraints, a formalization is provided describing how the constraint should limit the behaviour of a Petri net. A series of assumptions is furthermore given, describing properties that help ComMA users avoid creating inherently deadlocking specifications. For each of the constraints, a method was then proposed to represent and integrate the constraint into existing Petri net interface representations. The methods were lastly applied on a case, and the results suggest that the proposed methods do not lead to an exponential increase of the state space, indicating that they are scalable.

Contents

Chapter 1

Introduction

Systems are becoming increasingly more complex [1], making it more challenging to develop, maintain, and evolve them. One way to manage a complex system, both inside and outside the world of software development, is to make the system modular [2][3]. To then achieve modularity in a system, it can be developed as a *component-based system*.

A Component-based system (CBS) tries to achieve modularity by breaking the system down into different asynchronously communicating pieces, referred to as *components*. The goal is to then make these components easy to reuse and maintain. This is achieved by letting each component provide a set of services through a set of *interfaces*. An interface can specify types of messages and data, but also behaviour, thereby defining how a certain resource can be accessed. Having such a broad definition, interfaces can be described using many things ranging form a simple word document, and Interface Description Language (IDL) such as OpenAPI, or a Domain Specific Language (DSL) such as Component Modelling and Analysis (ComMA) [4]. As long as the definition of the interface of a service does not change, any change to the service behind the interface is not noticeable by anything using the service.

This arrangement potentially allows for a wide variety of clients to access a multitude of services provided by one or more different servers . The big challenge with CBS's is that they are hard to verify because of their concurrent nature. Changing the definition of an interface is easy, but what is difficult is to determine how this affects any clients using the service that the interface provides, or how this affects other interfaces that may depend on the changed interface in some way.

By creating a component specification using a modeling framework like ComMA, some analysis and verification can already be done. For example, traces can be generated from the interaction of specified interfaces, and monitors can be added to analyze compliance with the specified interface. However, this does not formally verify whether two interfaces are compatible. Compatibility between interfaces in this case implying the absence of deadlocks, livelocks, and any unbounded behaviour.

To do formal verification and analysis, the component specification could be translated to be represented in formal frameworks, such as Petri nets [5] or communicating state machines [6], that have existing and proven verification methods. With this representation, the compatibility of interfaces can be verified, and adapters that solve compatibility issues can even be generated [7][8]. Petri nets in particular are well suited to model component-based systems, as they provide a graphical, intuitive way of modeling systems in which events happen concurrently, furthermore allowing constraints on the occurrence, precedence and frequency of these events [9].

Verifying the compatibility of a single interface with a client is already possible. In ComMA specifically, an interface specification can already be represented as a Petri net, and then be verified to be compatible with a client interface [8]. However as mentioned previously, interfaces can have interdependencies. An example of such a dependency is that an action in one interface may only happen if another interface is in a certain state. Without a method to encode these constraints, interfaces that have such dependencies can currently not be verified formally.

1.1 Problem Statement

Manual verification for a set of interfaces is simply not feasible if the interfaces have any degree of complexity, so an automated method is required. It is already possible to take a ComMA interface specification, generate a Petri net representing its behaviour, and then formally verify its compatibility

with a single client. However, there exists no method to encode the interface constraints. To be able to do verification for specifications that contain these constraints, there would have to exist a way to encode those constraints as a Petri net, and no such method currently exists currently. By developing such a method, the range of models that can be formally verified is therefore increased.

Several approaches, using different types of Petri nets to model these constraints can be considered. Each of these types of Petri nets have different degrees of expressive power, but also come with a different range of verification methods. Here, the Petri nets with the least amount of expressive powers, have the widest range of verification methods, while this range is more limited for the more expressive Petri nets.

Encoding the constraints using a Petri net with a lower expressive power is desirable, as it would mean the resulting net is compatibly with existing verification and adapter generation methods [8].

It possible that a less expressive net will not suffice though. In this case, a more expressive Petri net type have to be used, making the encoding of constraints easier. However, as existing verification methods would no longer work in this case, more work would have to be done to make verification possible.

1.2 Contributions

This thesis has four main contributions:

1. A formalization for each interface constraint, describing how each constraint should limit the behaviour of a P/T net.
2. A set of assumptions, describing properties that help ComMA users avoid creating specifications with termination issues.
3. Methods for encoding the behaviour of these constraints into existing Petri net representations of interfaces.
4. Methods for validating whether a set of given constraints is encoded correctly into a given Petri net.

The goal is to allow for the verification of inter-dependent interfaces part of a component. Before introducing the methods that achieve this, a formalization of each of these types of constraints is given. This clearly defines the expected behaviour of each constraint. This then leads to the second contribution, which is the definition of a set of assumptions about how the constraints are expected to be defined in a specification. These assumptions act as guidelines, and can help people to specify interfaces that can be formally verified and do not deadlock.

The third contribution will be the methods that encode the constraints as a Petri net. The proposed methods are presented in the form of four pseudo-code algorithms, and each of the algorithms have a corresponding implementation in ComMA. Two of them encoding enabling and disabling constraints, and the other two encoding causal sequence constraints. Alongside each algorithm, proof sketches are given, showing that the properties defined in the formalization of each constraint are respected.

Lastly, methods for validating whether a set of given constraints is encoded correctly into a Petri net is provided. This validation is done using Neo4j, and its query language Cypher. These methods are given in the form of Cypher templates, one for each of the three constraints. As the algorithms are accompanied by proof sketches, the validation methods are meant to build additional confidence in the solution, and can be seen as a way to provide feedback to a user's given specification.

1.3 Outline

The rest of the thesis is organized as follows. Chapter 2 covers other works in the area of component-based model verification. The differences in the angles of approach and solutions are highlighted. Chapter 3 gives a brief introduction to ComMA, provides some necessary basics of Petri nets, and shows an example of the existing ComMA Petri net representation. In Chapter 4, for each type of constraints, a formalization alongside some terminology is presented. This is followed by series of assumptions introduced in Chapter 5. These assumptions are about the specification of interfaces and act as guidelines, and some of them will be used for proof sketches in Chapter 6.

Chapter 6 continues by presenting for each type of constraint, an algorithm encoding the constraint, as well as proof sketches showing that the algorithm produces a Petri net that complies with the properties defined in Chapter 4. Causal sequences are divided into smaller subsections covering different cases, starting with a simple case, and ending with the most complex case. Chapter 7 then covers a case study.

A ComMA specification of the case is given, and the results of applying the algorithms of Chapter 6 on these specifications are shown. This is followed by an explanation on how the results can be verified using Neo4j and Cypher. Lastly, the effects of the different constraint specifications on the state space of the resulting Petri net are covered. This is followed by a short chapter that covers the implementations that were made, both inside and outside of ComMA. Lastly, Chapter 9 gives a summary of the results is given, alongside some potential improvements and optimizations listed in the future work section.

Chapter 2

Related Work

There is a variety of works covering the verification of component-based systems. In general, two types of verification methods can be distinguished: Static and dynamic verification. Static verification is done based purely on specification, and does not involve executing an implementation of the system, while dynamic verification involves executing a program and performing runtime analysis. While this work uses a static verification method, there exist works that propose dynamic verification of component-based systems [10]. In [10], runtime verification is integrated into the component-based BIP framework [11]. Runtime verification approaches like the one used in [10], sacrifice completeness for applicability [12]. In this sense, it can be compared to trace analysis that can be done in ComMA to ensure compliance with an interface specification [13]. So while these types of approaches can be valuable in certain contexts, they do not align with the goal of this thesis to formally verify the compatibility of interfaces in component-based systems.

Another approach is *compositional verification*, in which a divide and conquer approach to verification is taken. Both [14] and [15] use such an approach, and in both cases a verifiable decomposition of the system must be found. In [15] specifically, a *guarantee-reasoning* approach is used. Using this technique, to verify a component that is part of larger network of components, a set of assumptions is calculated. The potential limitations of this technique lie in the calculations of these assumptions, whereas the limitation of the approach in this thesis lies in having to analyze the statespace of an entire system. Another example involving a static verification method, is the work done in [16]. Like in this work, a translation is made from a modeling framework specification, Dezyne, to a formal framework mCRL2 [17]. The first difference in this approach is that the work in [16] focuses on translating the full language of Dezyne to a formal framework, while this work focuses purely on interface constraints in the ComMA framework. In [16], this translation is being done to mCRL2, a framework based on process algebra, rather than Petri nets. These things make the approach in [16] fundamentally different and less applicable to this work.

There are several works that do propose Petri net based methods for component-based system verification [18][19][20]. In that sense, these works have an approach that is the closest to what is being done in this thesis. How they fundamentally differ, is that they do verification on the component level, focusing on the behaviour and dependencies of components. This work takes an approach that is a level of abstraction lower, focusing on the internal behaviour of the interfaces part of a single component. The methods proposed in [18],[19] and [20] all consider multiple components and the dependencies between them.

Some form of interface dependencies are considered implicitly in [18],[19] and [20]. One interface providing a service may expect a certain order of operations, and if another interface requiring that service does not adhere to that order, deadlock may occur. In that sense, more implicit interface dependencies that cause deadlock can be captured using these methods. However, as mentioned previously, this work considers explicitly defined interface dependencies, such as the requirement that an action in one interface must be followed by a sequence of actions in other interfaces. And no method exists yet to encode such dependencies into a Petri net, meaning that interfaces containing these constraints could not be verified feasibly.

Chapter 3

Background

This chapter provides the necessary background information for the concepts used in this thesis. Section 3.1 covers ComMA, which is the modeling framework in which the component and interface specifications are given. Afterwards, in Section 3.2, some basic mathematical notation is introduced. In Section 3.3 Petri nets are covered, which is what the ComMA specifications will be translated to by the methods proposed in chapter 5. Finally, Section 3.4 will go over Pnet, which is the Petri net representation that already exists in ComMA.

3.1 ComMA

ComMA is a modeling framework that supports component-based development of systems[1]. ComMA supports model-based engineering, by allowing for the specification of components and interfaces, the generation of monitors that can verify compliance, and the generation of documentation among other things. For this thesis, we are primarily interested in the specification of components and interfaces in ComMA.

3.1.1 ComMA Interface Specification

A ComMA interface definition describes both structure and behaviour. The internal behaviour of an interface is specified in the form of a protocol state machine. The structure of an interface is defined by its *signature*. An interface's *signature* specifies a set of events, and these events can come in three forms: signals, notifications, and commands.

- A *signal* is an asynchronous message sent from a client to a server.
- A *notification* is an asynchronous message sent from a server to a client.
- A *command* is a synchronous message sent from a client to a server. This is a blocking call, as the client waits for a reply after the command has been executed.

Figure 3.1 shows an example of an interface signature, containing five signals and one notification, but no commands.

```
1  signature Imaging
2
3  signals
4  turnOn
5  turnOff
6  image
7  image2
8  initialize
9
10 notifications
11 done
```

Figure 3.1: ComMA interface signature

[1] https://esi.nl/research/output/tools/comma

An interface's behaviour is described as a protocol state machine, so its definition will therefore contain a set of *states*, one of which is the initial state. For each state, a set of outgoing *transitions* can then be defined. A transition can be defined to be triggered or non-triggered. A triggered transition happens as a result of a signal or a command, meaning it was initiated by the client. A triggerless transition happens as a result of a notification, meaning that the server executed a transition autonomously. Figure 3.2 shows the specification of an interface with three states: Off, On and processing. For the transitions, the events of Figure 3.1 are used. Note that each transition has a definition of a tag. These tags allow for specific transitions to be referenced in a component specification, such that they can be used to define constraints. This will be shown later in Section 3.1.2 Tags are always in the format < source state > _ < event > _ < target state >, so the tag on Line 10 in Figure 3.2 refers to the transition turnOn coming from the state Off, going to the state On.

Transitions can furthermore contain guards, and allow the specification and manipulation of data. However, these concepts are outside the scope of this thesis, and will therefore not be explained further.

```
1  import "Imaging.signature"
2
3  interface Imaging version "1.0"
4
5  machine StateMachine {
6    initial state Off {
7
8      transition trigger: turnOn
9      next state: On
10     (tag Off_turnOn_On)
11   }
12
13   state On {
14
15     transition trigger: turnOff
16     next state: Off
17     (tag On_turnOff_Off)
18
19     transition trigger: image
20     next state: Processing
21     (tag On_image_Processing)
22
23     transition trigger: image2
24     next state: Processing
25     (tag On_image2_Processing)
26
27   }
28
29   state Processing {
30     transition do:
31     done
32     next state: On
33     (tag Processing_done_On)
34   }
35 }
```

Figure 3.2: ComMA interface specification

3.1.2 CommA Component Specification

The specification of a component is done in component file. The first thing that gets specified are the interfaces that are part of the component. When adding interfaces to a component, an interface can either be a provided interface or a required interface. The provided interfaces of a component are the interfaces that the client will interact with directly. Required interfaces then provide services in the background that the client does not interact with directly, but are required for a system to function. So in that way, components can be users of other components, while they are technically not clients. Dependencies between interfaces can only be defined between interfaces belonging to the same class, i.e. constraints can be defined between two provided interfaces, but not a required and a provided interface. To scope the problem, only provided interfaces are considered in this thesis.

The second part of a component specification that we consider are the functional constraints, which

are the constraints defined on the interfaces. In this thesis these constraints are classified into the previously mentioned three different classes of *enabling*, *disabling* and *causal sequence* constraints. It needs to be noted however, that in ComMA, an enabling constraint can only be defined as part of a causal sequence, while a disabling constraint is defined on its own.

Figure 3.3 shows an example of a component specification with four interfaces, and two causal sequence constraints containing two enabling constraints. Here, the first causal sequence constraint *init_sequence* specifies that a turnOn action in the Imaging interface must be followed by a turnOn in the Vacuum interface, followed by a turnOn in the Temperature interface. The enabling constraint of the sequence then specifices that the turnOn action in the imaging interface can only happen if both the temperature and the vacuum interface are in the Off state.

A disabling constraint is essentially the inverse of the enabling constraint, specifying that a transition cannot be taken if another interface is in a certain state.

```
1  import "Temperature/Temperature.interface"
2  import "Vacuum/Vacuum.interface"
3  import "Imaging/Imaging.interface"
4  import "Monitor/Monitor.interface"
5  component imagingComponent
6
7  provided port Temperature iTemperaturePort
8  provided port Imaging iImagingPort
9  provided port Vacuum iVacuumPort
10 provided port Monitor iMonitorPort
11
12 functional constraints
13
14 causal-seq init_sequence {
15   iImagingPort::Off_turnOn_On
16   where iTemperaturePort in Off and iVacuumPort in Off
17   leads-to
18   iVacuumPort::Off_turnOn_On
19   iTemperaturePort::Off_turnOn_On
20
21 }
22
23 causal-seq off_sequence {
24   iImagingPort::On_turnOff_Off
25   where iTemperaturePort in On and iVacuumPort in On
26   leads-to
27   iVacuumPort::On_turnOff_Off
28   iTemperaturePort::On_turnOff_Off
29
30 }
```

Figure 3.3: ComMA component specification

3.2 Basic Mathematical Notations

A set S is an unordered potentially infinite collections of elements, and is denoted using curly brackets. The set $S = \{1, 2, 3\}$, denotes a set containing elements 1, 2 and 3. $|S|$ denotes the number of elements in the set S, and $\emptyset$ denotes the empty set. We use $s \in S$, to denote that the element s is part of the set S. $S' \subseteq S$ denotes that S' is a subset of S, meaning that every element of S' is also an element of S, implying that $|S'| \leq |S|$. We use $\mathcal{P}$ to denote the powerset of a set S, which is the set of all subsets of S. The set union set operation $\cup$ is defined in the standard way. $A \times B$ denotes the Cartesian product, which is the set of all ordered pairs $\{(a, b) \mid a \in A \wedge b \in B\}$. We say that a function $m : S \to \mathbb{N}$ over some set S is a bag over S. For some $s \in S$, $m(s)$ denotes the number of occurrences of s in m. The set of all bags over S is denoted by $B(S)$.

We use $\mathbb{N}$ to denote the set of natural numbers, where $0 \in \mathbb{N}$. A finite sequence over a set S with length $n \in \mathbb{N}$ is denoted by σ, where σ is a function $\sigma : \{1, .., n\} \to S$. We denote the set of all finite sequences over S by S^*. $|\sigma|$ denotes the length of a sequence. A sequence of length n is represented by $\sigma = \langle s_1, ..., s_n \rangle$, where $s_1, ..., s_n \in S$ and $\sigma(i) = s_i$ for $1 \leq i \leq n$. We use $x \in \sigma$ to denote that the element x is part of the sequence σ. If $|\sigma| = 0$, the sequence is empty, which is denoted by ϵ. For a sequence $\sigma = \langle a_1, ..., a_n \rangle$ with $n \in \mathbb{N}$, a subsequence σ' of σ is defined as $\sigma'_k = \sigma_{n_k}$ with $k \in \mathbb{N}$, where

$n_1 < n_2 < ...n_k < ...$ is an increasing sequence of indices. For a sequence σ, σ^* denotes the set of all possible subsequences.

A directed graph is an ordered pair $G = (V, E)$, where V is a set of vertices, and $E \subseteq V \times V$ a set of edges. Edges have directions represented by a head and a tail, and for an edge (x, y), x is the head and y is the tail. We say that a graph is *strongly connected*, if for each $e_1, e_2 \in V$, there exists a directed path from e_1 to e_2. A *bipartite graph*, is a graph whose vertices can be divided into two disjoint, independent sets. This means that no edge exists whose head and tail is in the same set.

3.3 Petri Nets

A Petri net [21], is a modeling language that is often used to describe distributed systems. A big advantage of Petri nets, is that it provides an intuitive, graphical representation, while still having a rigorous mathematical foundation. In this thesis we focus on one type of Petri net, called a place/transition (P/T) net.

A P/T net is a bipartite graph, and consists of three different things: a set of places that can hold one or more tokens, a set of transitions that can consume or produce tokens to these places, and a set of arcs that determine which places transitions can consume and produce tokens to. A formal definition of a P/T net is given in Definition 3.1.

Definition 3.1 (P/T net)
A P/T net N is defined as $N = (P, T, F)$, where

- *P is a set of places*
- *T a set of transitions*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs.*

where P and T are pairwise disjoint. We refer to $P \cup T$ as the set of nodes.

Figure 3.4 shows an example of a P/T net defined as follows:

$$N = (P = \{S1, S2, S3, S4, S5\}, T = \{T1, T2, T3, T4\}, F = \{(S1, T1), (T1, S2), (S2, T2), (S2, T3), (T2, S3), (T3, S3), (S3, T4), (T4, S1)\}$$

Here, S1 is an example of a place, T1 a transitions, and the arrows between the places and transitions are the arcs.

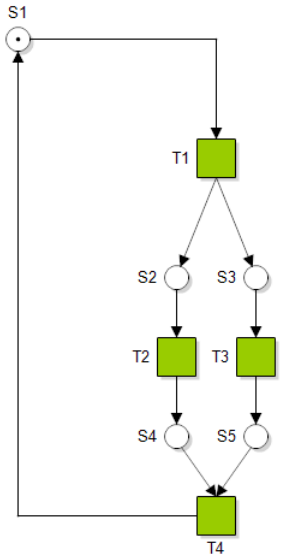

Figure 3.4: An example P/T net

The *preset* of a node n, consists of all nodes that have an outgoing arc to n. The *postset* of a node n, consists of all nodes that have an incoming arc from n. This is formalized in Definition 3.2. In Figure 3.4, the preset of the transition T1 consists only of S1, and its postset consists of both S2 and S3.

Definition 3.2 (Presets and postsets)
The preset of a node is denoted by ${}^\bullet n = \{m \mid (m,n) \in F\}$*, and the post set as* $n^\bullet = \{m \mid (n,m) \in F\}$*. We use the preset and postset as functions, where* ${}^\bullet u(v) = 1$ *and* $u^\bullet(v) = 1$ *if* $v \in {}^\bullet u$ *and* $v \in u^\bullet$ *respectively, and* ${}^\bullet u(v) = 0$ *and* $u^\bullet(v) = 0$ *otherwise.*

The state of a P/T net is the distribution of tokens in the net, called a *marking*, and is a bag over a nets places. Definition 3.3 describes this formally. The current marking of the net shown in Figure 3.4, would indicate that only S1 has one token.

Definition 3.3 (Marking)
A marking m *is defined as* $m : P \mapsto \mathbb{N}$*, indicating the number of tokens that a given place has in the marking. The initial marking of a net is denoted by* m_0

We say that a transition t is *enabled* in a marking, if at least one token is present in all places in the preset of t. Definition 3.4 formalizes what it means for a transition to be enabled.

Definition 3.4 (Enabledness of transitions)
A transition t *is enabled in some marking* m *in some P/T net* N *iff* $\forall p \in P_N : {}^\bullet t(p) \leq m(p)$*.*

When a transition is enabled, it is allowed to *fire*. This notion is formalized in Definition 3.5. When a transition fires, it removes a token from each place in its preset, and places a token in each of the places in its postset.

Definition 3.5 (Transition firings)
In a P/T net N*, when an enabled transition* t *fires in a marking* m*, this results in a new marking* m' *where:*

$$\forall p \in P_N : m'(p) = m(p) - {}^\bullet t(p) + t^\bullet(p)$$

.

We use the term *firing sequence* to refer to a sequence of transition firings. This is formally defined in Definition 3.6.

Definition 3.6 (Firing sequences)
A firing sequence of length $n \in \mathbb{N}$ *is denoted by* $\sigma = \langle t_0, t_1, ..., t_n \rangle$*. A firing sequence* $\sigma = \langle t_0, t_1, ..., t_n \rangle$ *is enabled in a marking* m *iff:* $\exists m_1, ..., m_n \in B(P) : m \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 ... \xrightarrow{t_n} m_n$*, and we denote this as* $m \xrightarrow{\sigma}$*.*

An example of a firing sequence would be the sequence of $T1$ followed by $T2$ in Figure 3.4.

The concept of reachability is important when we need to determine the possible behaviour of a Petri net. The reachability of a Petri net defines the nets reachable markings, or its *state space*. Definition 3.7 formalizes this.

Definition 3.7 (Reachability)
We say that a marking m' *is reachable from another marking* m *in a net* N *if* $\exists \sigma T*$*, such that* $m \xrightarrow{\sigma} m'$*. We use* $R(N,m) = \{m' \mid m \xrightarrow{*} m'\}$ *to denote all reachable markings in a net* N *from the marking* m*.*

The reachability of a net is often described in a reachability graph, an example of which is shown in Figure 3.5. Here, each node represents a marking, and the connection between each node a transition firing. In Figure 3.5, we can see that from m0, which is the initial marking also displayed in Figure 3.4, only T1 can fire. After T1 fires, either T2 can fire followed by T3, or T3 followed by T2.

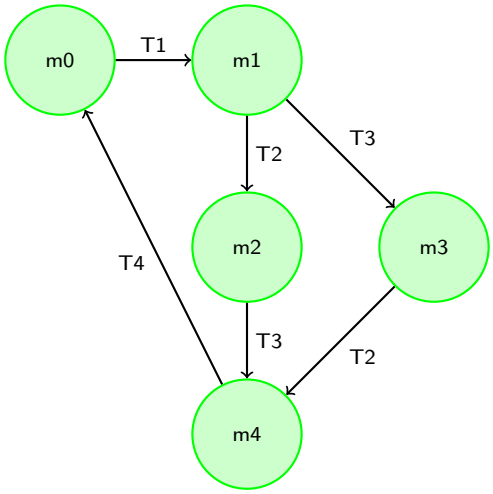

Figure 3.5: Reachability graph of the example P/T net

In Figure 3.5, we can see that it is possible to end up back in the initial marking m0. In fact, in any given node, we always end up in back in the initial marking at some point. In this case we say that the net is *weakly terminating*.

Definition 3.8 (Weak termination)
A P/T net N is weakly terminating iff:

$$\forall m \in R(N, m0) : \exists m \xrightarrow{*} m0$$

where $m \xrightarrow{*} m0$ *indicates that* $m0$ *is reachable from* m *through one or more transition firings.*

Because the behaviour of ComMA interfaces is described by protocol state machines, we introduce the notion of a state machine P/T net, formalized in Definition 3.9. Being a state machine net restricts the behaviour of P/T net, in that it disallows concurrent behaviour.

Definition 3.9 (S-net)
A P/T net N is an S-net iff: $\forall t \in T_N : |{}^\bullet t| \leq 1 \wedge |t^\bullet| \leq 1$ *and all markings have exactly one token.*

An open net, formalized Definition 3.10, is the standard way of modeling an interface as a Petri net, as it has dedicated places representing the inputs and outputs of an interface. Given two open nets, their interface places can be fused. This allows for the modeling of, for example, a client and a server. The concept of interface places is out of scope for this thesis, so a more detailed explaination can be found in [22].

Definition 3.10 (Open nets)
OPNs have two separate sets of input and output interface places. All sets of places and transitions in the OPN are pairwise disjoint. The net in which an OPN's interface places are omitted, is called the OPN's skeleton. If the skeleton of an OPN is an S-net, the OPN is called an state machine open net (S-OPN).

All interfaces are defined as S-OPN's. However, during the rest of this thesis, only the skeleton of interfaces will be considered. This is because we only care about the internal behaviour of the interfaces when trying to encode interface constraints.

In this context, we use the term *component* to refer to a collection of interfaces. This is formally defined in Definition 3.11.

Definition 3.11 (Component)
Components are defined as a set of S-OPN's representing its interfaces. For a component $\mathcal{O}$*, all* $N \in \mathcal{O}$ *are pairwise disjoint. For a component* $\mathcal{O}$*,* $P_\mathcal{O}$ *denotes the set of all places* $\bigcup_{N \in \mathcal{O}}(P_N)$*, and* $T_\mathcal{O}$ *the set of all transitions* $\bigcup_{N \in \mathcal{O}}(T_N)$*.*

3.4 Pnet

As mentioned previously, there already exist methods to translate a ComMA interface to a Petri net. This currently done by translating a ComMA interface to the *Pnet* format, which is a ComMA DSL representation of a Petri net. When not considering variable definitions and guards, a Pnet representation is semantically equivalent to a formal P/T net definition. Therefore, this representation can be directly mapped to a more standardized P/T net format such as pnml, which is used by most Petri net verification tools [23]. This means that the interface skeletons are a given, and that this thesis focuses only on connecting these skeletons through methods encoding interface constraints.

Figure 3.6 shows an example Pnet representation of an interface with two states, On and Off, and two transitions, turnOn and turnOff. The representation then simply consists of two places, Off and On. On then has turnOff in its postset and turnOn in its preset, and Off has turnOn in its postset and turnOff in its preset.

```
1  interface-net: StateMachine
2  {
3    init-token
4    internal-place Off {
5      post-transitions:
6                Off_Imaging_turnOn_1_1
7    }
8    internal-place On {
9      post-transitions:
10               On_Imaging_turnOff_1_1
11   }
12
13
14   transition-trig: On_Imaging_turnOff_1_1 {
15       ( tag  On_turnOff_Off  )
16     trigger: turnOff
17     post-place: Off
18   }
19
20   transition-trig: Off_Imaging_turnOn_1_1 {
21       ( tag  Off_turnOn_On  )
22     trigger: turnOn
23     post-place: On
24   }
25 }
```

Figure 3.6: Example Pnet representation

Chapter 4

Formalizing The Constraints

We can categorize three different types of constraints that can model interface dependencies:

1. Enabling constraint: An action in one interface can only occur if one or more other interfaces are in a certain state.
2. Disabling: An action in one interface *cannot* occur if one or more other interfaces are in a certain state.
3. Causal sequence constraint: An action must be followed by an uninterrupted sequence of actions across different interfaces.

This chapter introduces formalizations for each of these constraints. Each constraint will have one definition, and this definition includes a property that the Petri net encoding introduced in Chapter 6 should satisfy.

4.1 Enabling Constraint

Because interfaces are S-nets, there is a one to one mapping between states in an interface in ComMA, and the places in the Petri net representation of this interface. That is, with S-net markings having only one token, a place p having a token, represents the interface being in a state s. An enabling constraint for a transition is defined as an additional enabling condition based on the states of other interfaces. That is, an enabling constraint on a transition requires places in other interfaces to contain a token in order to be enabled.

Definition 4.1 (Enabling constraint)
Given a Component $\mathcal{O}$ and a set of transitions $T_c \subseteq T_\mathcal{O}$, an enabling constraint is defined as

$$C_e : T_c \to P_\mathcal{O}$$

For a transition $t \in T_c$, the enabling constraint $C_e(t) = \{p1, p2\}$ indicates that t may only fire if both $p1$ and $p2$ have at least one token.

A component $\mathcal{O}$ satisfies a set of constraints C_e iff for any given transition $t \in T_c$ and marking $m \in B(P_\mathcal{O})$:

$$\text{if } m \xrightarrow{t} \text{ then } \forall p \in C_e(t) : m(p) = 1$$

4.2 Disabling Constraint

The disabling constraint requires a transition to be *disabled* if other interfaces are in a certain state. The same reasoning used for the enabling constraint can be applied here. But instead, places must now act as a disabling condition, rather than an additional enabling condition. That is, a disabling constraint on a transition requires that this transition is disabled if a set of places in other interfaces contain at least one token.

Definition 4.2 (Disabling constraint)
Given a Component $\mathcal{O}$ and a set of transitions $T_c \subseteq T_\mathcal{O}$, a disabling constraint is defined as

$$C_d : T_c \to P_\mathcal{O}$$

For a transition $t \in T_c$, the enabling constraint $C_d(t) = \{p1, p2\}$ indicates that t cannot be enabled if both $p1$ and $p2$ have at least one token.

A component $\mathcal{O}$ satisfies a set of constraints C_d iff for any given transition $t \in T_c$ and marking $m \in B(P_\mathcal{O})$:

$$\text{if } m \xrightarrow{t} \text{ then } \forall p \in C_d(t) : m(p) = 0$$

4.3 Causal Sequence

A causal sequence constraint requires a certain transition to be followed by an uninterrupted firing of a specified sequence of transitions.

Definition 4.3 (Causal sequence constraint)
Given a Component $\mathcal{O}$ and a set of transitions $T_c \subseteq T_\mathcal{O}$, a causal sequence constraint is defined as follows:

$$C_s : T_c \to T^*$$

A component satisfies a causal sequence constraint C_s iff for any given transition $t \in T_c$ and marking $m \in B(P_\mathcal{O})$:

$$\text{if } m \xrightarrow{t} m' \text{ then:}$$

$$\exists m' \xrightarrow{\sigma} \text{ such that } \sigma = C_s(t) \wedge \neg \exists m' \xrightarrow{\sigma'} \text{ such that } \sigma' \neq \sigma$$

For a component $\mathcal{O}$ and a transition $t \in T_\mathcal{O}$, if $C_s(t) \neq \epsilon$, we say that t is an activation transition of a sequence. Any transition $t_n \in C_s(t)$ where $0 \leq n \leq |C_s(t)|$, is a consequence transition. An activation transition cannot belong to more than one causal sequence constraint of a component $\mathcal{O}$. Furthermore, a transition cannot be an activation transition if it is also a consequence transition. If a transition is neither a consequence nor an activation transition of any defined causal sequence constraint of component O, we refer to it as free. For two consequence transitions t_n, $t_{n+1} \in C_s(t)$ where $0 \leq n < |C_s(t)|$, we say that t_{n+1} is a sequence successor of t_n, and that t_n is a sequence predecessor of t_{n+1}. Furthermore, only one sequence may be active at the same time, meaning that after some activation transition t fires, any other activation transition cannot be enabled.

4.3.1 Overlapping and Diverging Sequences

The concepts of *overlapping* and *diverging* sequences are now introduced. As will become clear in Chapter 6, these concepts influence the complexity of the algorithm required to model the causal sequence constraint.

It is possible that the sequence of consequence transitions of two or more causal sequence constraints overlap. That is, for two activation transitions t and t', and some sequence of consequence transitions σ, σ may be a subsequence of both $C_s(t)$ and $C_s(t')$. Whenever a set of sequences share a subsequence of consequence transitions, we say that the sequences are *overlapping*. If a set of sequences has σ as a common subsequence, we say that the sequences overlap on σ. For two sequences $C_s(t) = \langle t_0, t_1, t_2, s_0, s_1 \rangle$ and $C_s(t') = \langle t'_0, t'_1, s_0, s_1 \rangle$, this is illustrated in Figure 4.1.

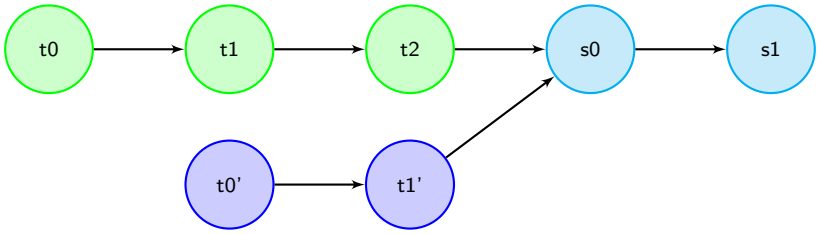

Figure 4.1: Overlapping sequences

The behaviour of the sequences shown in Figure 4.1 is fully deterministic. In every step, we know exactly which transition comes next. However, what if we have two overlapping sequences in which the two common subsequences are followed by different transitions? Consider two sequences $C_s(t) = \langle t_0, t_1, t_2, s_0, s_1, t_3 \rangle$ and $C_s(t') = \langle t'_0, t'_1, s_0, s_1, t'_2 \rangle$, where the sequences overlap on $\langle s_0, s_1 \rangle$. Figure 4.2 illustrates these sequences, and we can see that for transition $s1$ it is no longer clear which transition should follow afterwards. It could be $t3$ or $t2'$, depending on which sequence is active. In this case, we say that the set of sequences are *diverging*, and that $s1$ is a *divergence point*.

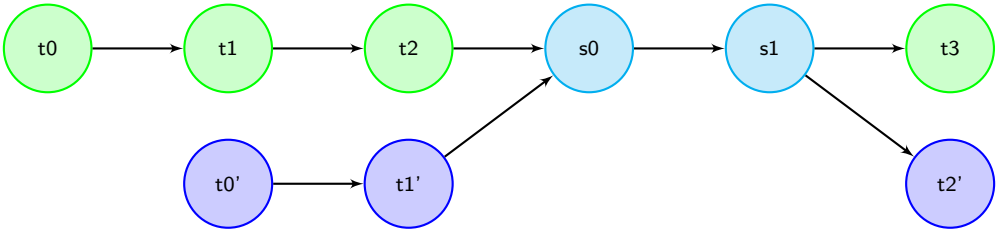

Figure 4.2: Overlapping and diverging sequences

Suppose now that $t2'$ would not be part of the causal sequence constraint $C_s(t')$, as illustrated in Figure 4.3.

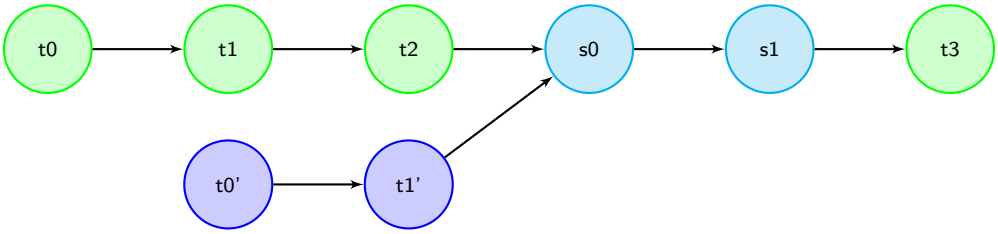

Figure 4.3: Same sequences where the transition $t2'$ is left out

In this case, $s1$ may not have to be followed by another transition at all, depending on which sequence is active. In this case, $s1$ is also a divergence point. As will become clear in Chapter 6, these are both problems that need to be dealt with when creating the Petri net representation of causal sequence constraints.

4.3.2 Multi-Stage Sequences

The concept of divergence can also occur in the context of a single sequence. This is the case when a consequence transition occurs more than once in a single causal sequence constraint. Consider the following causal sequence constraint on an activation transition t, where the index of each element in the sequence is denoted as a superscript: $C_s(t) = \langle t_0^0, t_1^1, t_2^2, t_0^3, t_3^4 \rangle$. This is illustrated in Figure 4.4.

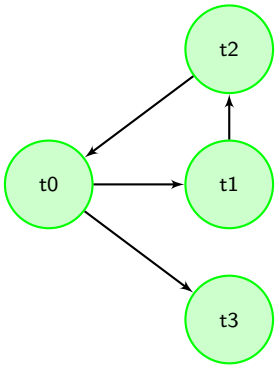

Figure 4.4: Multi-stage sequence

As can be seen in Figure 4.4, the causal sequence contains a loop, making it possible for transitions to have multiple successors within the same sequence. In this case, how do we know whether $t0$ should be followed by $t1$ or $t3$? To be able to make this distinction, we divide the sequence up in *stages*. We then call the sequence shown in Figure 4.4 a *multi-stage* sequence, as transitions may have to be followed up by different transitions depending on which stage of execution the sequence is in. Because the sequences shown in Figures 4.1 and 4.2 do not have this problem, we call them *single-stage* sequences. In the example illustrated in Figure 4.4, the only point of ambiguity is in $t0$, where we have to decide between $t1$ or $t3$. We can solve this ambiguity by dividing the sequence up into two stages. The first stage is active as soon as the sequence is activated, and we can then say that if the sequence is in the first stage, $t0$ should be followed by $t1$. After $t0$ fires the first time, the second stage becomes active. We can then say that if the sequence is the second stage, $t0$ should be followed by $t3$.

We now want to generalize this, and to do this we want to identify divergence points like with the diverging sequences covered in Section 4.3.1. We first define the *sequence successors* of a transition. For a transition t part of some causal sequence S, the *sequence successors* of t are all of the successors of t in the order in which they appear in the S. So for $t0$, this is the sequence $\langle t_1^1, t_3^4 \rangle$, and for $t1$ it is $\langle t_2^2 \rangle$. The sequence successors are defined in the context of a single sequence, and as such, transitions part of multiple sequences will have multiple different sequences of sequence successors.

To identify the divergence points, we want to look at the sequence successors of each transition t of a sequence. Every transition t is considered a divergence point, if t has more than one sequence successors when excluding recurring transitions, but including any transitions that are the final transition of the sequence. We want to include the final transitions of a sequence for the same reason that we consider $s1$ in Figure 4.3 to still be a divergence point. The final transition of a sequence is a special case that needs separate consideration when creating the representation in Chapter 6. For example, $t0$ would be considered a divergence point, as its sequence successors are $\langle t_1^1, t_3^4 \rangle$, which are two unique transitions. A transition $t0'$ part of some causal sequence S with $\langle t_1^1, t_1^3 \rangle$ as its sequence successors is not considered a divergence point, unless t_1^3 is the final transition of S.

Once all divergence points within a sequence are identified, we want to combine their sequence successors into one large sequence of *stage transitions*. When adding the sequences of sequence successors, we omit the final transition, as the final transition of a sequence successor sequence does not mark the start of a new stage. For the example in Figure 4.4, only t_0 is a divergence point. Therefore, the sequence of stage transitions for $C_s(t)$ is simply $\langle t_1^1 \rangle$, as we take $\langle t_1^1, t_1^3 \rangle$ and omit t_1^3.

In the above example, only one transition was actually a divergence point, making the construction of the stage transition sequence trivial.

Now consider the following causal sequence constraint on an activation transition t:

$$C(t) = \langle t_0^0, t_3^1, t_1^2, t_2^3, t_0^4, t_1^5, t_2^6, t_0^7, t_3^8, t_4^9 \rangle$$

illustrated in Figure 4.5, where the index of each element is superscripted. We can divide this sequence

up into stages consisting of the following transitions:

$$\begin{aligned} \textit{Stage } 0 &= \langle t_0^0 \rangle \\ \textit{Stage } 1 &= \langle t_3^1 \rangle \\ \textit{Stage } 2 &= \langle t_1^2, t_2^3, t_0^4 \rangle \\ \textit{Stage } 3 &= \langle t_1^5, t_2^6, t_0^7, t_3^8, t_4^9 \rangle \end{aligned}$$

So the stage transitions are t_3^1, t_1^2 and t_1^5.

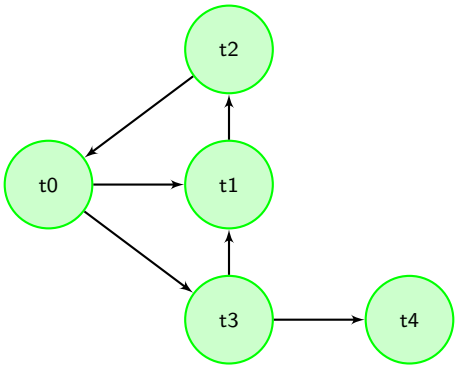

Figure 4.5: Multi-stage sequence

We now construct the sequence of stage transitions for the causal sequence illustrated in Figure 4.5. We first identify the divergence points, which are t_0 and t_3, as their sequence successors are $\langle t_3^1, t_1^5, t_3^8 \rangle$ and $\langle t_1^2, t_4^9 \rangle$ respectively. We now want to combine these sequences to get the sequence of stage transitions. We do this by interleaving them, preserving the order based on each element's index in $C_s(t)$. A simple algorithm to do for a causal sequence $C_s(t)$ is shown in Figure 4.6. The algorithm iterates over each transition of $C_s(t)$, and for each transition its sequence successors are obtained. If the transition is a divergence point, the first element of the sequence of sequence successors is added to the sequence of stage transitions. This element is then removed afterwards. By then also checking that the length of the sequence of sequence successors is greater than one, we ensure that the last element is always omitted.

```
1  sequence_successors = mapping of a transition to its sequence successors.
2  stage_transitions = ⟨⟩
3  for each transition t in C_s(t):
4      successors = stage_successors[t]
5      if t is a divergence point and |successors| > 1:
6          successor = successors[0]
7          stage_transitions.add(successor)
8          delete successors[0]
```

Figure 4.6: Stage transition sequence algorithm

This gives us the following stage transition sequence: $\langle t_3^1, t_1^2, t_1^5 \rangle$. Once again the last elements of each sequence of sequence successors are omitted, and as the superscripted indices are in ascending order, the order in which they appear in $C_s(t)$ is preserved. Having three stage transitions, this sequence then has four stages. In this sequence, we call t_1^2 a *stage transition successor* of t_3^1. We say that t_2^3 is part of the third stage, as it occurs after the second stage transition, but before the third.

Chapter 5

Specification Guidelines

When creating component and interface specifications, it is possible that the specification has inherent problems. For example, it might be that the initial state is not reachable from all other states, or that the interface constraints contain cyclical dependencies. Such errors could lead to a deadlocking Petri net representation, regardless of the representation method. This chapter covers several problematic specification patterns in the form of assumptions, some of which are used in the proof sketches of Chapter 6. It is important to note that these assumptions are a necessary, but not sufficient, condition for deadlock-freedom.

5.1 Assumptions

Aside from constraint specifications, interface specifications can also be bad. That is, given a component $\mathcal{O}$, one of its interfaces $N \in \mathcal{O}$ might already be deadlocking without any constraints being encoded yet. This is if the protocol state machine describing the behaviour of an interface is not strongly connected, meaning that the any state of the interface is not reachable from all other possible states. Naturally, this would mean that the Petri net representation is then also not strongly connected, which means that it not deadlock-free.

Assumption 5.1
Given a component $\mathcal{O}$, the following is assumed to be true:

$$\forall N \in \mathcal{O} : N \textit{ is strongly connected}$$

We must furthermore make an assumption about the state of each interface as soon as the activation transition of a causal sequence fires. Let t be an activation transition for the causal sequence $C_s(t)$. It is possible that after t fires, for any $t_n \in C_s(t)$ belonging to an interface N, that N is not in a state in which t_n is enabled. If this is not always the case, the Petri net representation of the of the specification will not be deadlock free. Consider Figure 5.1, where two interfaces named interface one and interface two are shown on the left side and right side respectively. Suppose that there is a causal sequence constraint $C_s(T0) = \langle T2, T1 \rangle$. This means that after $T0$ fires, $T2$ must fire next. In the current marking however, $T2$ cannot fire, and the firing of $T3$ would be required to make that possible. However, after $T0$ fires in this situation, any transition not part of $C_s(T0)$ must be disabled, which includes the transition $T3$. In this case the net deadlocks after the firing of $T0$, because $s2$ is not guaranteed to have a token after the firing of $T0$. In terms of interface specifications, this means that constraints defined on interface two do not guarantee that interface two is in the state $s2$ as soon as the transition $T0$ is taken.

We furthermore need to consider that the firing of consequence transitions also affects the states of the interfaces. This means that making an assumption about the state of each interface before the firing of an activation transition is not enough. To satisfy this constraint, any causal sequence constraint must comply with the behaviours defined by the protocol state machine of each interface part of the sequence. For example, the sequence of transitions $\langle T0, T1, T0, T1 \rangle$ would be valid for interface one in Figure 5.1, but the sequence $\langle T0, T0, T1 \rangle$ is not. This is because a $T0$ in interface one can only be followed by a $T1$ in interface one. When looking at causal sequence constraints specifically, $C_s(T0) = \langle T2, T1, T3 \rangle$ respect the behaviour of both interface one and two, but $C_s(T0) = \langle T2, T1, T2 \rangle$ does not. This is because a $T2$ in interface two can only be followed by a $T3$ in interface two.

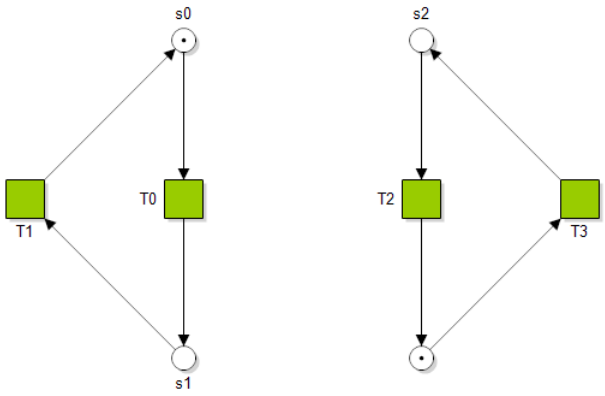

Figure 5.1: Illustration of the described situation

Assumption 5.2 formalizes this by requiring that after a consequence or activation transition fires, the interface that its sequence successor t_{succ} is part will have a token in the correct place.

Assumption 5.2
Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, ..., t_n \rangle$ where $n \in \mathbb{N}$. For any marking m that is the result of the firing of t:

$$t_0 \text{ is enabled in } m$$

For any marking m' that is the result of the firing of $t_i \in C_s(t)$ where $0 \leq i \leq |C_s(t)| - 1$:

$$t_{i+1} \text{ is enabled in } m'$$

Because consequence transitions cannot fire while no sequence is active, there may be situations in which an interface deadlocks. This could happen if a place only has consequence transitions in its postset, while having at least one free transition in its preset. This situation is illustrated in Figure 5.2. In this case, if p receives a token through the free transition, it has to be possible to reach a marking in which an activation transition in another interface is enabled. This activation transition must then lead to a sequence containing one of the consequence transitions in the postset of p.

Assumption 5.3
For a component $\mathcal{O}$ and a set of causal sequence constraints defined by C_s the following must hold:

$$\forall p \in P_{\mathcal{O}}, \text{ such that } \exists t \in {}^\bullet p: \text{ where } t \text{ is free, and } \forall t' \in p^\bullet: \text{ } t' \text{ is a consequence transition}:$$

$$\text{Then } \forall m \in R(m_0, \mathcal{O}) \text{ such that } m(p) = 1:$$

$$\exists m' \in R(m, \mathcal{O}) \text{ such that } m' \xrightarrow{t_{act}} \wedge\, t' \in C_s(t_{act}), \text{ where } t' \in p^\bullet$$

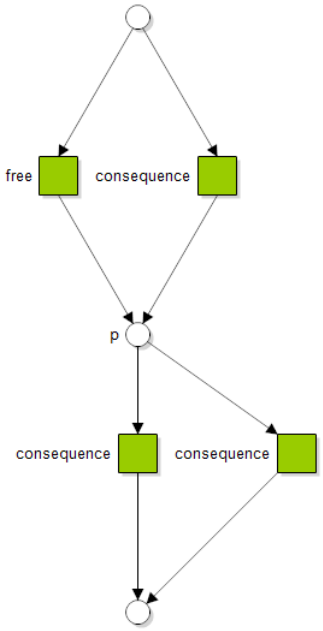

Figure 5.2: Illustration of the described situation

An enabling constraint may also interfere with a causal sequence constraint. Let $\mathcal{O}$ be a component, where t is an activation transition for the consequence sequence $C_s(t) = \langle T_m, T_n \rangle$, where T_m and T_n are two consequence transitions part of different interfaces. It is possible to define an enabling constraint $C_e(t_n) = \{p\}$, where p is a place part of $N \in \mathcal{O}$. t_m may now also be part of N, and could furthermore only be enabled in any marking m where $m(p) = 0$. If we then assume that Assumption 5.2 holds, then p cannot have a token in the current marking, as in all markings in which t_m was enabled, p could not have a token. This is because the behaviour of all interfaces is described by protocol state machines, which cannot be in two states at the same time. Because p does not have a token now, t_n cannot be enabled because of the constraint $C_e(t_n)$, resulting in deadlock as soon the sequence gets activated by the firing of t. Such a situation is shown in Figure 5.3, where both the interfaces are in the correct state, but the enabling constraint $C_e(t_n) = \{p\}$ cannot possibly be satisfied to enable t_n. This is because after T_m fires, the only transition that is allowed to fire is T_n. Because T_2 is not allowed to fire, there is no way for there to be a token in p, which means that T_n cannot be enabled.

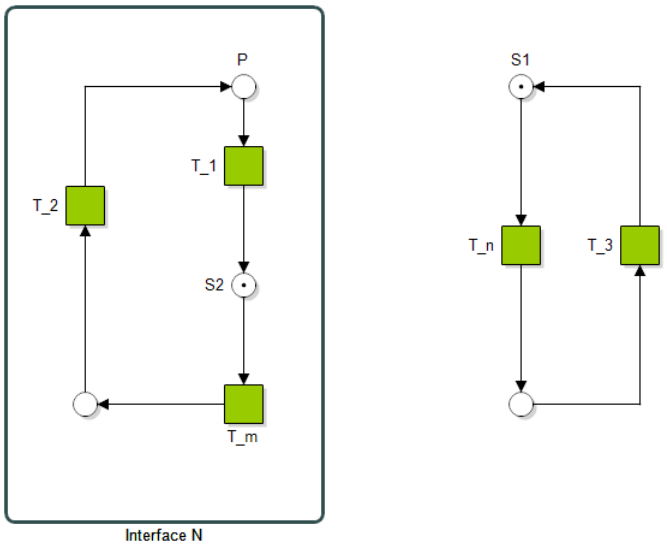

Figure 5.3: Enabling constraint interference

Assumption 5.4
Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, ..., t_n \rangle$ where $n \in \mathbb{N}$. For any marking m that is the result of the firing of t:

$$\forall c \in C_e(t_0) : m(c) = 1$$

For any marking m' that is the result of the firing of $t_i \in C_s(t)$ where $0 \leq i \leq |C_s(t)| - 1$, and the transition $t_{i+1} \in C_s(t)$ part of some interface N:

$$\forall c \in C_e(t_{i+1}) : m'(c) = 1$$

Interference may also arise between disabling and causal constraints. Let $\mathcal{O}$ be a component, where t is an activation transition for the consequence sequence $C_s(t) = \langle T_n, T_m \rangle$, where T_n and T_m are two consequence transitions part of different interfaces. There may now be a disabling constraint $C_d(t_n) = \{p\}$, while t_m is only enabled in markings in which p does have a token. Assuming that Assumption 5.2 holds, p must have a token as t_m needs to be enabled. But because of $C_d(t_n) = \{p\}$, T_n is simultaneously not allowed to fire, resulting in deadlock. Such a situation is shown in Figure 5.3, where both the interfaces are in the correct state according to Assumption 5.2, but the disabling constraint $C_e(t_n) = \{p\}$ cannot possibly be satisfied to enable T_n.

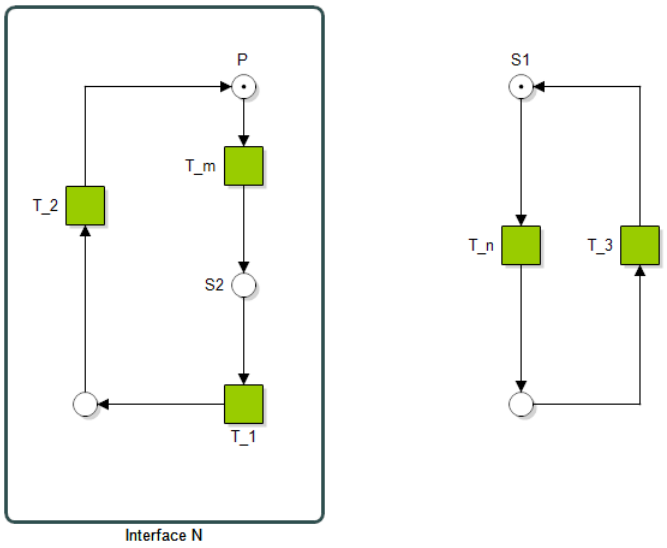

Figure 5.4: Disabling constraint interference

Assumption 5.5
Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, ..., t_n \rangle$ where $n \in \mathbb{N}$. For any marking m that is the result of the firing of t:

$$\forall c \in C_d(t_0) : m(c) = 0$$

For any marking m' that is the result of the firing of $t_i \in C_s(t)$ where $0 \leq i \leq |C_s(t)| - 1$, and the transition $t_{i+1} \in C_s(t)$ part of some interface N:

$$\forall c \in C_d(t_{i+1}) : m'(c) = 0$$

5.2 Cyclical Dependencies

It is possible that a specified set of enabling or disabling constraints have cyclical dependencies, leading to a deadlocking net. Consider two transitions $t \in N$ and $t' \in N'$ that have cyclical dependencies in terms of enabling constraints. A cyclical dependency in this case means that t relies on interface N' being in a state only reachable through a firing sequence containing t', while t' relies on the interface N being in a state only reachable through a firing sequence containing t.

Consider the example shown in Figure 5.5, containing two interfaces. Suppose now that there are two enabling constraints $C_e(T_1) = \{S4\}$ and $C_e(T_4) = \{S2\}$. For T_1 to be enabled, the firing of T_4 is required to satisfy the enabling condition on T_1. However, for T_4 to be enabled, the firing of T_1 is required to satisfy the enabling condition on T_4. This is a cyclical dependency.

For disabling constraints, again consider Figure 5.5. Suppose now that there are two disabling constraints $C_d(T_1) = \{S3\}$ and $C_d(T_4) = \{S1\}$. For T_1 to be enabled, the firing of T_4 is required to satisfy the disabling condition on T_1. However, for T_4 to be enabled, the firing of T_1 is required to satisfy the disabling condition on T_4. This is again a cyclical dependency.

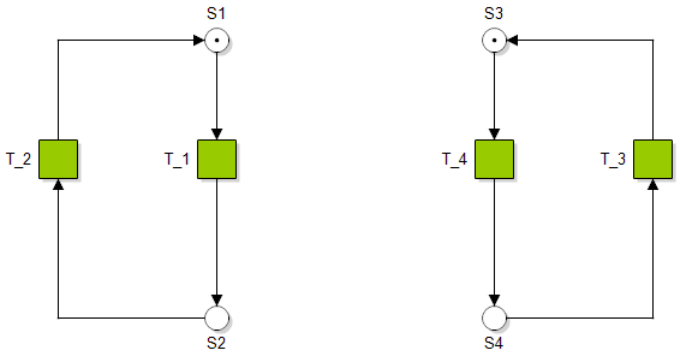

Figure 5.5: Example to showcase cyclical dependencies

5.2.1 Detecting cyclical dependencies

The recursively defined function $findCycles$ in Figure 5.6 walks through all possible dependency chains to try and find cyclical dependencies. The algorithm is applied on the interface skeletons of a component, before any of the structures introduced in Chapter 6 get added. The algorithm starts by going over all places, and looking at the enabling constraints of the transitions in the preset of each place. This is then repeated for each of the places found in the enabling constraints of these transitions. Cyclical dependencies can then be found by keeping track of which places have visited while following these chains. However, just considering visited places is not enough. This has to do with the fact that interfaces are S-nets, and can therefore only have one token at a time. If we follow a dependency chain, and end up at a place which we have not yet visited, but is part of an interface we have already visited, this also problematic. It would imply that a transition depends on an interface being in two places at the same time. This means that we have to keep track of interfaces that were already visited.

For there to be a deadlock, a cyclical dependency must exist along every single chain of dependencies starting from a transition. That why when visiting a place, a list that stores Boolean values is kept. For every enabling or disabling constraint on the transition, either True or False is then added to this list depending on whether a cycle is found directly, or somewhere further along a dependency chain. The function findCycles then only returns True if every single value in the results list is True.

The algorithm of Figure 5.6 specifically is for cycles in a set of enabling constraints. Given a place $c \in C_e(t)$, a list of visited places, and a list of visited interfaces, the algorithm considers three different cases:

1. c is in the list of visited places. In this case, a cycle has been found, and True will be added to the list of results. (Line 9)
2. c is not in the list of visited places, but the interface i that c is part of is in the list of visited interfaces (Lines 10-19). If i has already been visited, it simply means that a place part of i has already been visited. If this is the case, the specification may be bad or simply contain redundancies, depending on the situations covered by the two cases below. In this case, the first visited place v of i is considered, and one of the following cases applies:

    - If c is in the preset of all transitions t' in the preset of v, then it means that all t', through a chain of dependencies, depend c having a token (Lines 12-17). However, this is naturally the case as c is in the preset of all t' by the design of the interface i. In this case, there is a redundancy in the specification, but this does not lead to deadlock. In this case, the recursion will continue with either True or False being added to the list of results depending on the result of each findCycles call.
    - If c is not in the preset of all transitions t' in the preset of v, then all transitions t', through a chain of dependencies, depend on c having a token (Lines 18-19). However, considering that i is an S-net, and that therefore only one place of i can have a token in any marking, this does not make sense. This is because by the design of the interface i, all transitions t' depend on some place c' that is not c to fire. Because i is an S-net, there cannot simultaneously be a token in both c and c'. In this case True is added to the list of results, as we have encountered a situation that is impossible to satisfy.

3. c is not in the list of visited places, and the interface that c is part of has not been visited yet (Lines 20-25). In this case, the recursion will continue with either True or False being added to the list of results depending on the result of each findCycles call.

The check is done for each interface $i \in \mathcal{O}$, for each place in $p \in P_i$, where $findCycles$ is called with each transition t in the preset of p. In the end, a specification will only be guaranteed to lead to deadlock if a cycle is found amongst all followed dependency chains starting in each t.

While the only purpose of this algorithm is to check if a specification will not lead to a deadlocking net, it is important to keep in mind that any cycles that are found are a sign of a bad specification. This is because the existence of cycles guarantees that there will be transitions that are always disabled.

```
1  function findCycles(C_e, t, visited_places, visited_interfaces):
2      results = ∅
3      new_visited_places = visited_places.add(C_e(t))
4      new_visited_interfaces = visited_interfaces
5      for each c in C_e(t):
6          new_visited_interfaces.add(interface of c)
7      for each c in C_e(t):
8          if(c in visited_places):
9              results.add(True)
10         else if(interface of c in visited_interfaces):
11             v = first visited place part of the interface of c:
12             if c in •t' for each t' in •v:
13                 for each t in •c:
14                     if(findCycles(C_e, t, new_visited_places, new_visited_interfaces)
                           :
15                         results.add(True)
16                     else:
17                         results.add(False)
18             else:
19                 results.add(True)
20         else:
21             for each t in •c:
22                 if(findCycles(C_e, t, new_visited_places, new_visited_interfaces):
23                     results.add(True)
24                 else:
25                     results.add(False)
26         return (all(results, True) && results != ∅)
27
28
29
30 function validateComponent(O, C_e)
31     for each i in O:
32         for each p in P_i
33             results = ∅
34             for each t in •p:
35                 if(findCycles(C_e, t, {p}, {i})):
36                     results.add(True)
37                 else:
38                     results.add(False)
39
40             if all(results, True):
41                 print(Deadlock. For all transitions in •p, a cyclical depenency was
                       found)
42                 return False
43
44     return True
```

Figure 5.6: Dependency cycles algorithm for enabling constraints

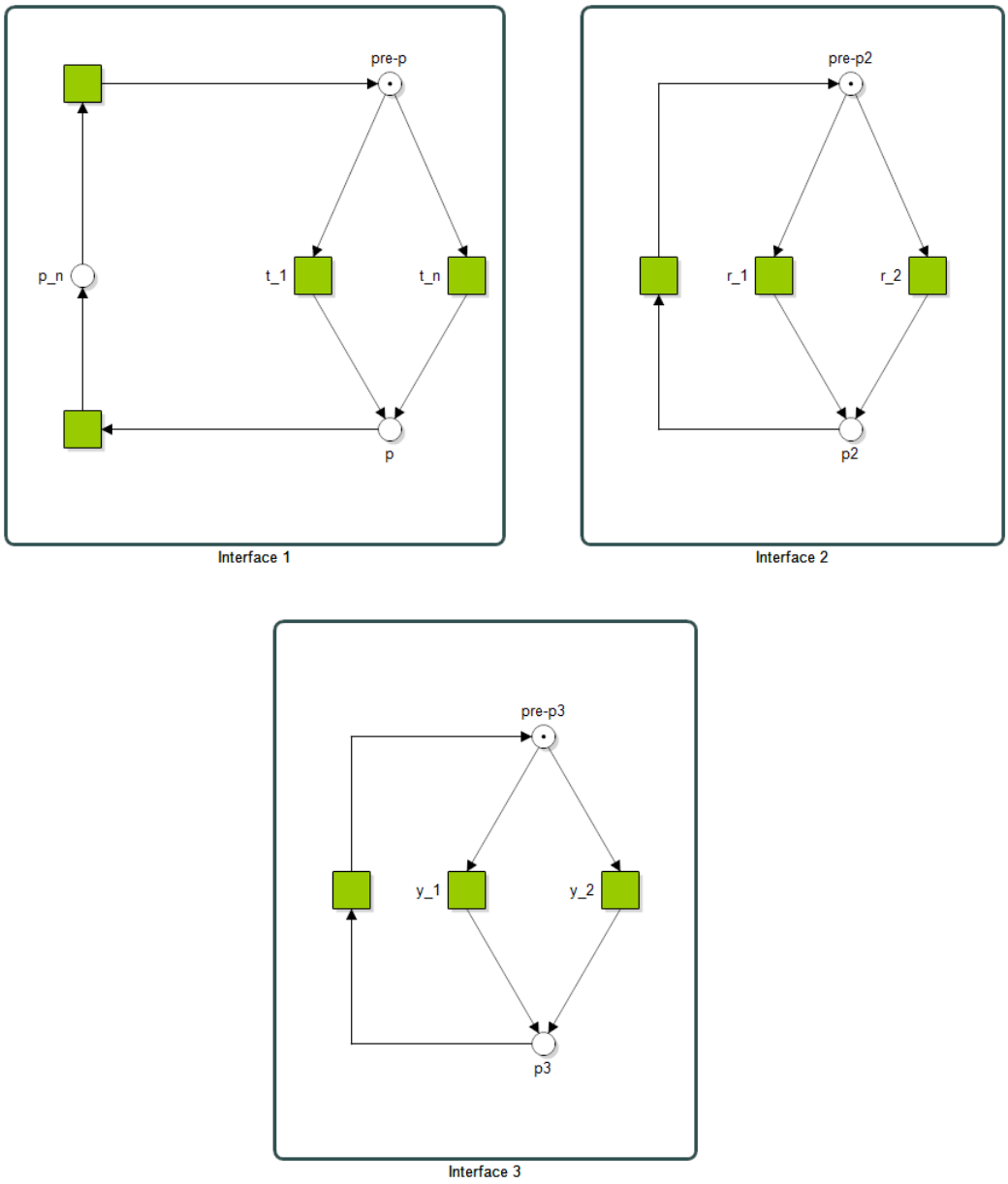

Figure 5.7: Example Component on which we apply enabling constraints that lead to circular dependencies

Example 5.6
This example shows a situation in which case 1 applies, and a cyclical dependency is thus found. Figure 5.7 shows parts of the skeletons of three different interfaces. Let the place p of interface one be the place referenced on line 32 in the Algorithm of Figure 5.6. For each transition t_1, t_2 in the preset of p, findCycles is now called. Suppose that $C_e(t\_1) = \{p2\}$, $C_e(r\_1) = \{p3\}$ and $C_e(y\_1) = \{p\}$. When findCycles is now called with t_1, the third case will apply first as neither p2 nor interface 2 has been visited yet. findCycles will thus be called for r_1 and r_2. When findCycles get called with r_1, case 3 will again apply for p3, meaning that findCycles will now get called for y_1 and y_2. After calling findCycles on y_1, case one will now apply, as p has already been visited. A cycle has now been found for the following chain of transitions: t_1, r_1, y_1. This does not imply deadlock, however, as findCycles will return False for the chain starting with t_2. This is because this transition does not have constraints, meaning that the results variable will be empty. For t_1 to be able to fire, any r_2 could simply fire as this transition does not have dependencies either. This showcases how a cycle must be found following all chains starting a place's preset in order to guarantee deadlock.

Example 5.7
This example shows in which situations case 2 applies. Figure 5.7 shows parts of the skeletons of three different interfaces. Let the place p of interface one be the place referenced on line 32 in the Algorithm of Figure 5.6. For each transition t_1 and t_2 in the preset of p, findCycles is now called. Suppose that $C_e(t\_1) = \{p2\}$, $C_e(r\_1) = \{p3\}$, $C_e(y\_1) = \{p\_n\}$, $C_e(t\_2) = \{p2\}$, $C_e(r\_2) = \{p3\}$ *and* $C_e(y\_2) = \{p\_n\}$. *When findCycles is now called with both t_1, the third case will apply in both cases as neither p2 nor interface 2 have been visited yet. In both instances, findCycles will thus be called for r_1 and r_2. For both r_1 and r_2, case 3 will again apply for p3, meaning that findCycles will now get called for y_1 and y_2.*

After calling findCycles on y_1 and y_2, case two will now apply in both instances. This is because p_n *has not been visited, but interface 1 has. The second sub-case of case two will apply specifically, as* p_n *is not in the presets of the transition in the presets of p, which is the first visited place of interface one. In this case, we have found that the specified enabling constraints lead to deadlock. This is because there in all dependency chains starting from p lead to a cycle. More specifically, we found the following dependency chains starting from* $_1$*:* $t_1 \to r_1 \to y_1 \to p_n$, $t_1 \to r_2 \to y_1 \to p_n$, $t_1 \to r_2 \to y_2 \to p_n$ *etc. Furthermore, starting from* t_2*:* $t_2 \to r_2 \to y_2 \to p_n$, $t_2 \to r_1 \to y_1 \to p_n$, $t_2 \to r_1 \to y_2 \to p_n$ *etc.*

Being able to detect cycles given a set of enabling constraints, this leads us to Assumption 5.8

Assumption 5.8
Given a component $\mathcal{O}$ *and a set of enabling constraints* C_e *defined on* $\mathcal{O}$, $\mathcal{O}$ *is only considered valid if the function validateComponent in Figure 5.6 returns true when called with* $\mathcal{O}$ *and* C_e.

5.2.2 Disabling Constraints

Like for enabling constraints, it is possible that a specified set of disabling constraints necessarily lead to a deadlocking net. Similarly to enabling constraints, there may exist cyclical dependencies. This is detected in almost the same way as for enabling constraints, except the postsets of places are now considered, rather than the presets. The algorithm can be found in Appendix A. This is showcased in Example 5.9. Enabling constraints may also interfere with disabling constraints and vice versa. This is currently not considered, but this could be covered by combining the algorithms of Figures 5.6 and A.1.

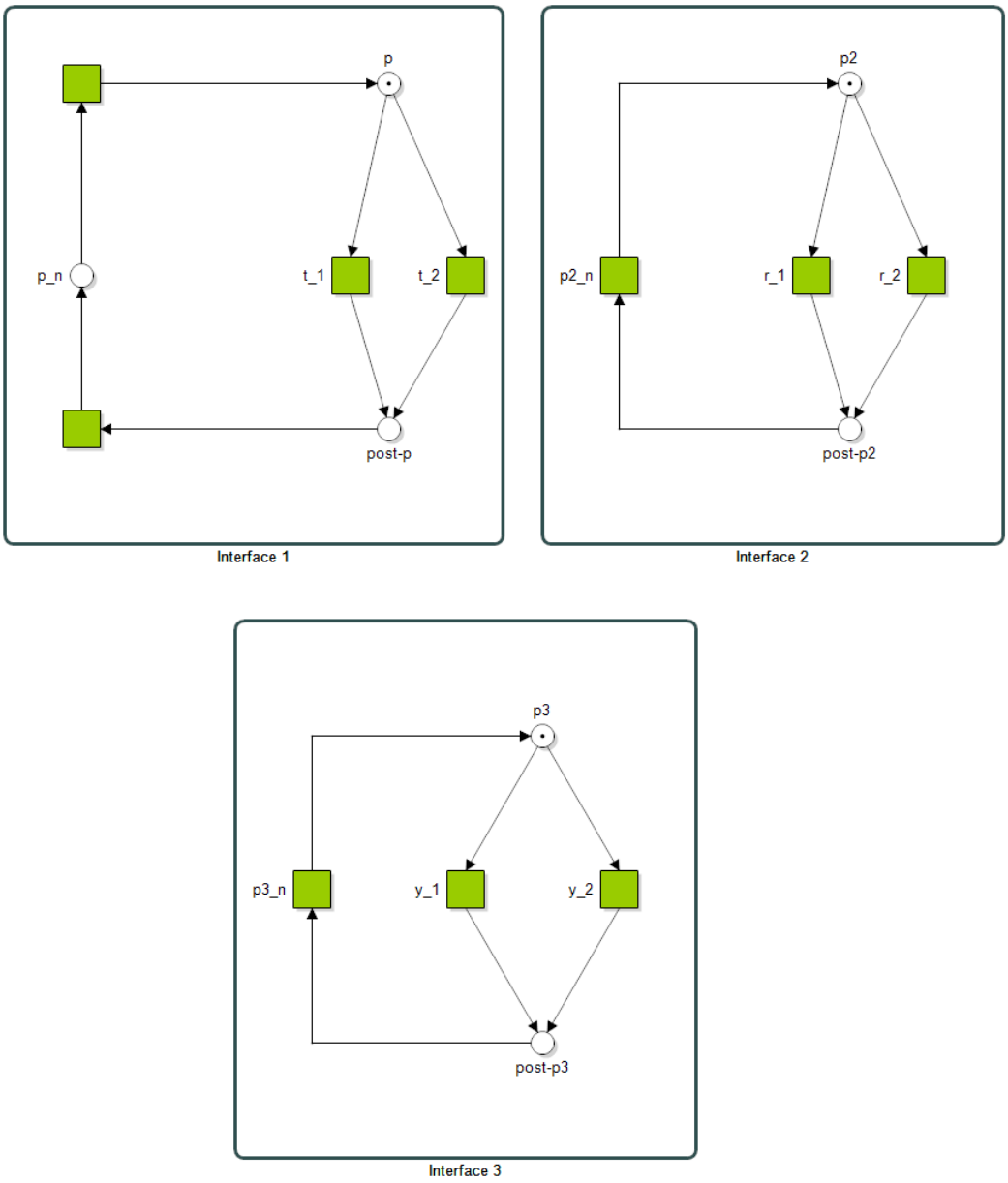

Figure 5.8: Example component on which the following disabling constraints are defined: $C_d(t\_1) = \{p2\}$, $C_d(t\_2) = \{p2\}$, $C_d(r\_1) = \{p3\}$, $C_d(r\_2) = \{p3\}$, $C_d(y\_1) = \{p\}$, $C_d(y\_2) = \{p\}$

Example 5.9

Figure 5.8 shows a net that will deadlock when the following disabling constraints are defined: $C_d(t\_1) = \{p2\}$, $C_d(t\_2) = \{p2\}$, $C_d(r\_1) = \{p3\}$, $C_d(r\_2) = \{p3\}$, $C_d(y\_1) = \{p\}$, $C_d(y\_2) = \{p\}$. *Let the place p of interface one be the place of line 28 in the algorithm. For each transition t_1 and t_2 in the preset of p, findCycles is now called. When findCycles is now called with t_1, the third case will apply first as neither $p2$ nor interface 2 has been visited yet. findCycles will thus be called for r_1 and r_2. When findCycles get called with r_1, case 3 will again apply for $p3$, meaning that findCycles will now get called for y_1 and y_2.*

After calling findCycles on y_1, case 1 will now apply, as p has already been visited. A cycle has now been found for the following chain of transitions: t_1, r_1, y_1. The same now goes for the chain t_2, r_2, y_2. This means that findCycles will return True for both calls with t_1 and t_2, correctly determining that the net in Figure 5.8 would deadlock if we were to apply the disabling constraints as described in this example.

Being able to detect cycles given a set of disabling constraints, this leads us to Assumption 5.10.

Assumption 5.10
Given a component $\mathcal{O}$ and a set of disabling constraints C_d defined on $\mathcal{O}$, $\mathcal{O}$ is only considered valid if the function validateComponent in Figure A.1 returns true when called with $\mathcal{O}$ and C_d.

Chapter 6

Encoding Constraints in Existing Interface Representations

Given a component that is described by both interface and constraint specifications, a Petri net must now be generated that satisfies these constraints. Having a set of interface specifications, an intermediate Petri net representation called Pnet can already be generated by ComMA. With this Pnet representation, the interface skeleton nets of a component are a given. What remains is to model the component constraints that add restrictions on enabling of transitions of the interface skeleton nets. In this chapter, we will present algorithms to generate constraint nets for all three kinds of component constraints. These nets are added on top of existing interface nets.

6.1 Enabling and Disabling Constraint

6.1.1 Enabling Constraints

For a component $\mathcal{O}$, and a set of enabling constraints defined by C_e, Figure 6.3 shows an algorithm that generates a Petri net that correctly encodes the constraints defined in C_e. The algorithm is fairly straightforward: Line 3 creates bidirectional arcs between t and all $p \in C_e(t)$ in order to enforce the constraints of $C_e(t)$. This is to ensure that when t fires, the token in p is not consumed.

Example 6.1
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.1. Let C_e denote the enabling constraints on $\mathcal{O}$, where $C_e(T1) = \{S4\}$ and $C_e(T2) = \{S3\}$. Figure 6.2 shows the result of applying the enabling constraint algorithm on $\mathcal{O}$.

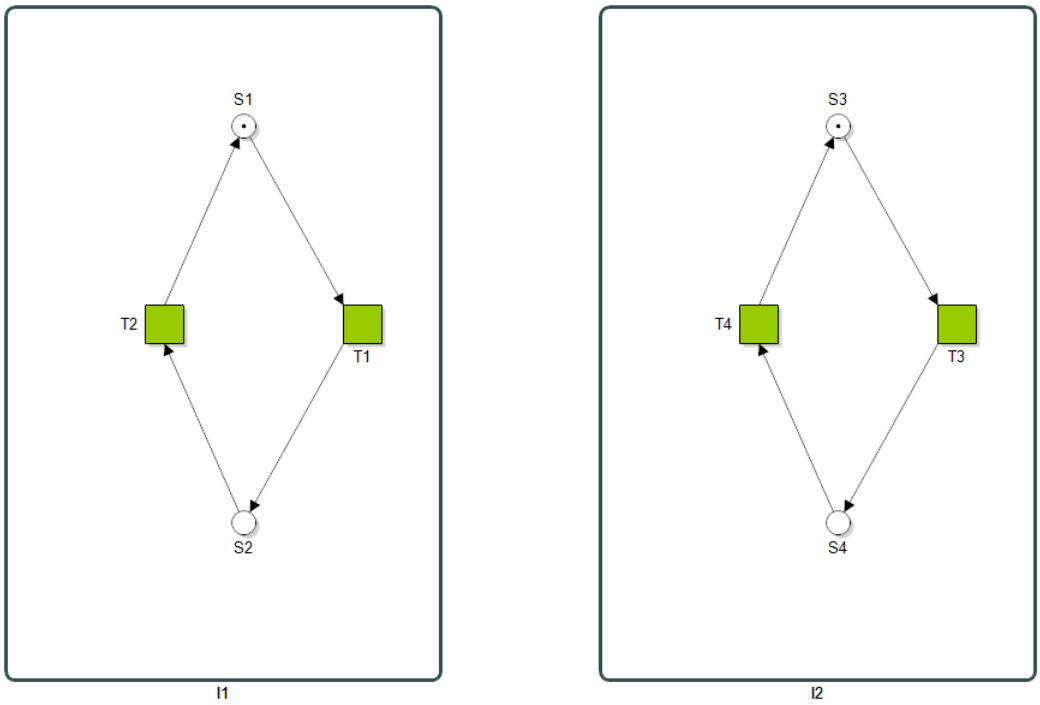

Figure 6.1: Component $\mathcal{O}$

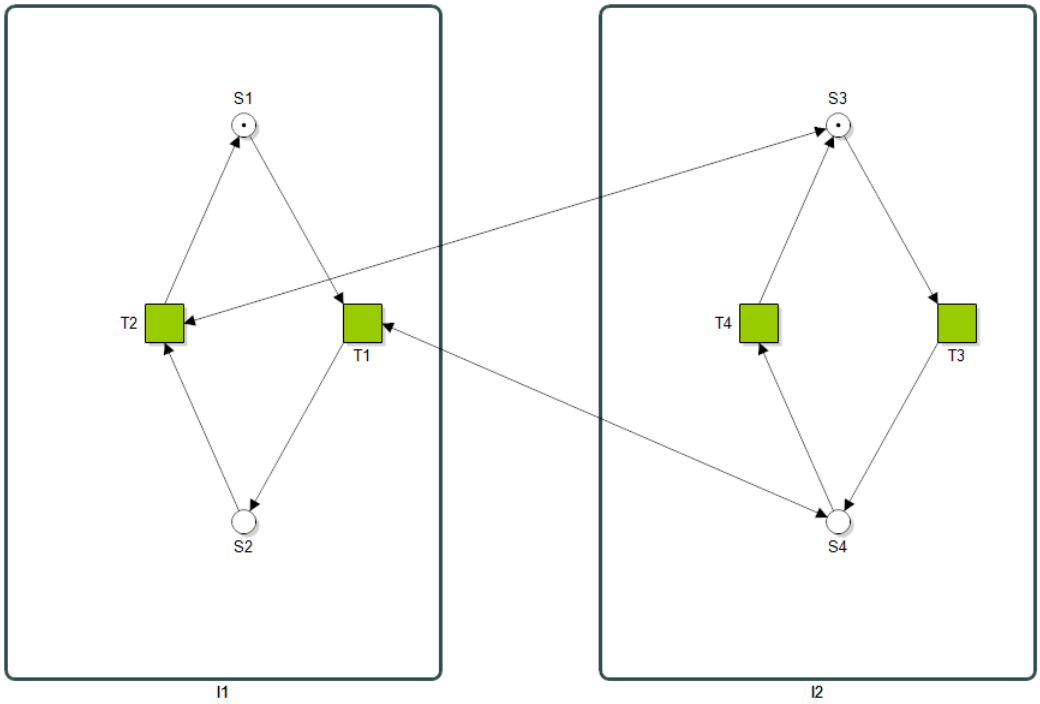

Figure 6.2: Resulting net after applying the enabling constraint algorithm of Figure 6.3 on $\mathcal{O}$

As can be seen, both the interface skeletons in Figure 6.1, as well as the net in Figure 6.2 are weakly terminating.

```
1  for each transition t in T_O:
2      for each place p in C_e(t):
3          create bidirectional arc <p, t>
```

Figure 6.3: Enabling constraint algorithm

Lemma 6.2
The property described in Definition 4.1 always holds for a component $\mathcal{O}$ after applying the algorithm in Figure 6.3 on $\mathcal{O}$.

Proof sketch. Let $\mathcal{O}$ be a component, and t be an arbitrary transition of $N \in \mathcal{O}$, and an arbitrary number of constraints defined by C_e.

Line 3 guarantees a bidirectional arc is created between t and each of the places in $C_e(t)$, meaning that all of the places in $C_e(t)$ are in the preset of t. Therefore, t can only fire if all places in $C_e(t)$ have a token, as required by the property in Definition 4.1.

6.1.2 Disabling Constraint

The algorithm for disabling constraints is in a way very similar to that of the enabling constraint, which makes sense as the disabling constraint is essentially the inverse. When $C_d(t) = \{p\}$ for some transition t a place p, instead of enabling t when p has a token, t must now be disabled instead. One way of making the algorithms almost identical would be to use *inhibitor arcs*. If there is an inhibitor arc between a place p and a transition t, it means that t is disabled if p has a token. Figure 6.4 shows how Example 6.1, now having disabling constraints instead of enabling constraints, could be modeled using inhibitor arcs. The algorithm to generate what is shown in Figure 6.4 would be the same as for the enabling constraint, except an inhibitor arc would be created from p to t, rather than a bidirectional arc.

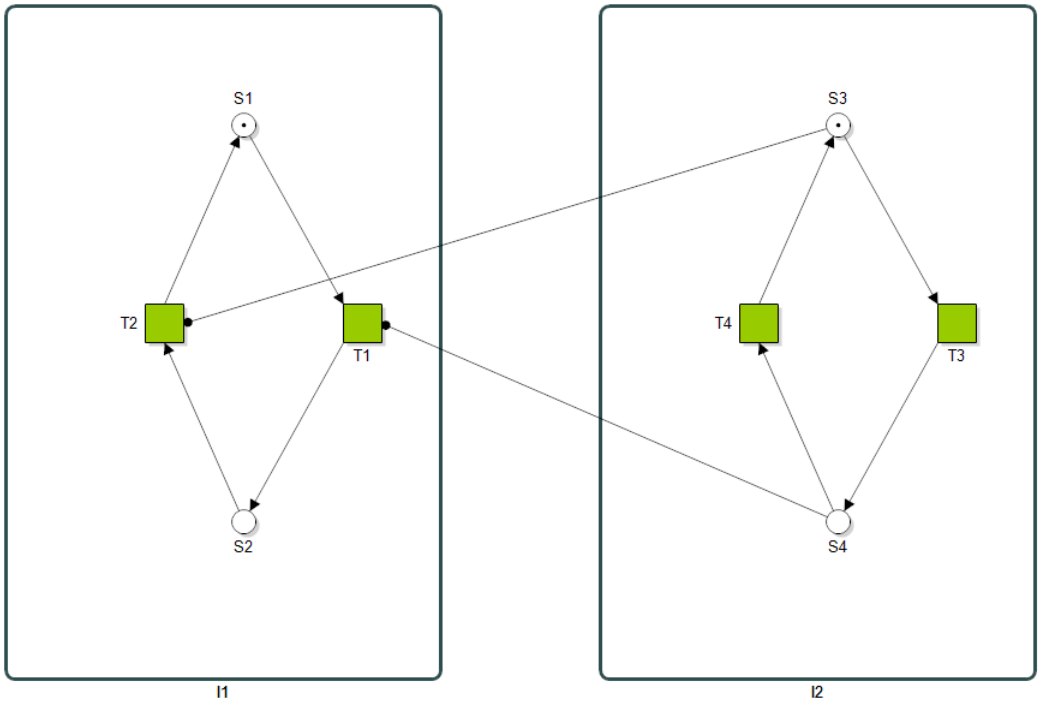

Figure 6.4: Modeling disabling constraints using inhibitor arcs

However, since we restrict ourselves to P/T nets, inhibitor arcs cannot be used. The concept of complement places can be used instead. It is important to note that replacing inhibitor arcs with such places only works for bounded nets [24][25]. This is not a problem as interface skeletons are S-nets, guaranteeing that every place can have at most one token at any time.

For a place p, its complement place $p_{complement}$ has a token only if p does not. With this behaviour, a bidirectional arc can then be created between a transition t and the complement place $p_{complement}$ of a place $p \in C_d(t)$.

For a component $\mathcal{O}$, and a set of disabling constraints defined by C_d, Figure 6.6 shows an algorithm that generates a Petri net that correctly encodes the constraints defined in C_d.

Line 3-6 are responsible for creating complement places, and making sure the complement places have the correct number of initial tokens, depending on whether p has a token initially. Lines 7-10 then make sure that the complement places is correctly updated as the net executes. This is done by creating arcs from the complement place of p, to each transition pt in the preset of p, and by creating arcs from each transition pt in the postset of p, to the complement place of p. Line 11 then creates a bidirectional arc that ensures that a transition can only be enabled if the right complement place has a token.

Example 6.3
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.1. Let C_d denote the disabling constraints on $\mathcal{O}$, where $C_d(T1) = \{S3\}$ and $C_d(T2) = \{S4\}$. Figure 6.5 shows the result of applying the disabling constraint algorithm on $\mathcal{O}$.

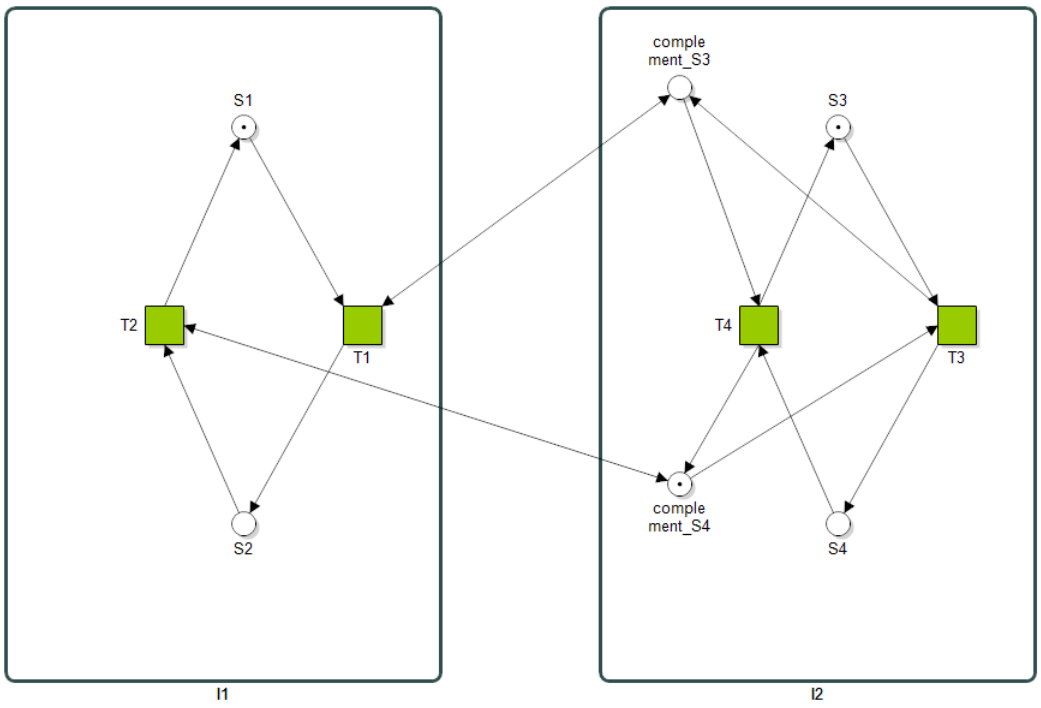

Figure 6.5: Resulting net after applying the disabling constraint algorithm of Figure 6.6 on $\mathcal{O}$

In Figure 6.5, we can see how the two disabling constraints of Example 6.3 are modeled by introducing two complement places for the places $S3$ and $S4$.

```
1  for each transition t in T_O:
2      for each place p in C_d(t):
3          if m_0(p) = 1:
4              create place <complement_p, 0 tokens>
5          else:
6              create place <complement_p, 1 token>
7          for each pt in •p:
8              create arc <complement_p, pt>
9          for each pt in p•:
10             create arc <pt, complement_p>
11         create bidirectional arc <complement_p, t>
```

Figure 6.6: Disabling constraint algorithm

Lemma 6.4
The property described in Definition 4.2 always holds after applying the above algorithm

Proof sketch. Let $\mathcal{O}$ be a component, and t be an arbitrary transition of $N \in \mathcal{O}$, and an arbitrary number of constraints defined by C_d. Because of Lines 3-6, every place in $C_d(t)$ has a complement place, with Lines 3-4 ensuring that the complement place has no token if the corresponding place has a token in the initial marking. For any place p in $C_d(t)$, its complement place $p_{complement}$ can only have a token if p has no token, as by Lines 7-8, any transition in the preset of p removes a token from $p_{complement}$, and by Lines 9-10, any transition in the postset of p adds a token to $p_{complement}$. With Line 11 then guaranteeing that each of these complement places are in the preset of t, t can thus only be enabled if for all p in $C_d(t)$, p has no token, as required by Definition 4.2.

6.2 Sequences

For the causal sequence constraint, four different cases with different assumptions about the sequence constraints are covered. In this section, three algorithms are proposed. The first one being sufficient to cover the first two cases, the second one additionally covering the third case, and the third one covering the fourth case.

6.2.1 Case 1: A Single Causal Sequence Constraint

The first case considers only a single causal sequence constraint. As a result, there can only be single-stage, non-overlapping and non-diverging sequences. For a component $\mathcal{O}$ and a causal sequence constraint $C_s(t) = \langle t_0, ..t_n \rangle$, the following things have to be guaranteed by the algorithm introduced in Figure 6.9 in order to satisfy the property given in Definition 4.3:

1. After t fires, all free transitions must be disabled until the last transition in $C_s(t)$ fires.
2. A transition $t_i \in C_s(t)$ with $1 < i < |C_s(t)|$, must only be enabled after $t_{i-1} \in C_s(t)$ fires.

The first requirement is satisfied by introducing a disabler place with an initial token. Any free transition has a bidirectional arc to this disabler place, which prevents them from being enabled whenever the disabler place has no token. All activation transitions furthermore have an incoming arc coming from this disabler, disabling all free transitions as well as any other activation transitions.

The second requirement is satisfied using enabling places. Every consequence transition has such an enabler place in its preset. Enabler places have no token in the initial marking, and the algorithm ensures that the enabler place of a consequence transition t can only receive a token from the sequence predecessor of t.

Example 6.5
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.7. Let C_s denote the causal sequence constraints on $\mathcal{O}$, where $C_s(T5) = \langle T1, T3, T4 \rangle$ and $C_s(T6) = \langle T2 \rangle$. Figure 6.8 shows the result of applying the enabling constraint algorithm on $\mathcal{O}$.

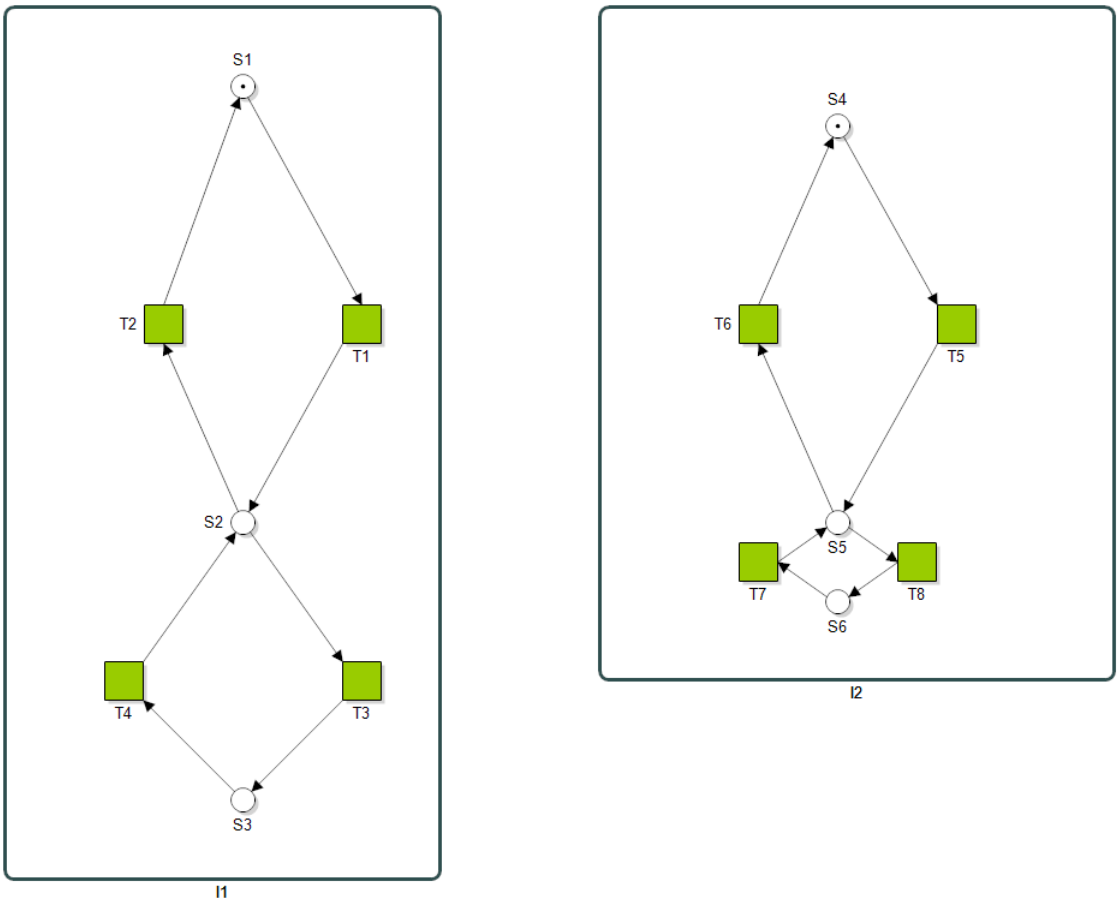

Figure 6.7: Component $\mathcal{O}$

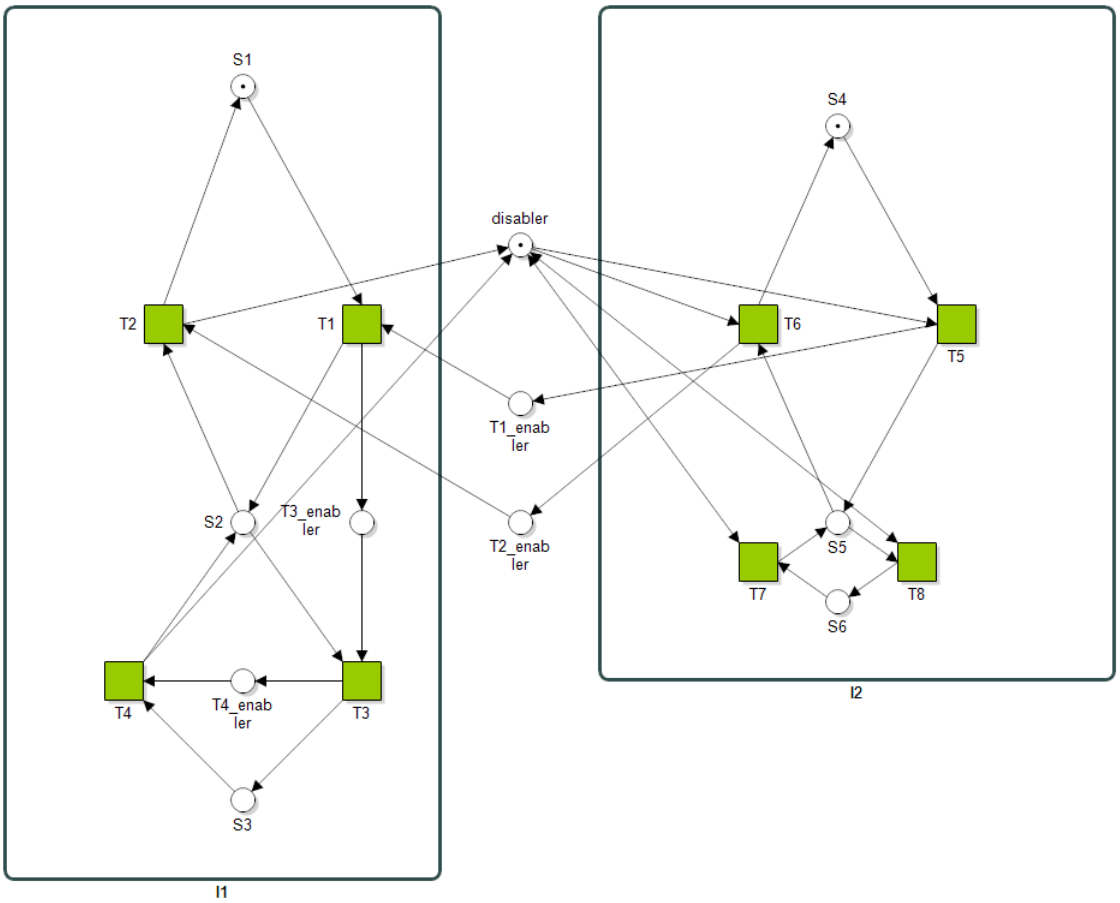

Figure 6.8: Resulting net after applying the sequence constraint algorithm of Figure 6.9 on $\mathcal{O}$, with the following constraints: $C_s(T5) = \langle T1, T3, T4 \rangle$ and $C_s(T6) = \langle T2 \rangle$

In Figure 6.8, we can see the bidirectional arc between the free transitions $T7$ and $T8$, and the disabler place. We can furthermore see the arcs going from the disabler place going to the activation transitions $T5$ and $T6$, ensuring that only one sequence can be active, and that $T7$ and $T8$ are disabled. The arcs going from the final transition of each sequence $T4$ and $T2$ to the disabler place then allow $T7$, $T8$, $T5$ and $T6$ to be enabled again as soon as any of the sequences finishes.

```
1  create place <disabler, 1 token>
2  for each transition t in T_O:
3      if t is free:
4          create bidirectional arc <disabler,t>
5      else if t is an activation transition, leading to a consequence transition
           sequence starting with t_0:
6          create place <t_0_enabler, 0 tokens>
7          create arc <t_0_enabler, t_0>
8          create arc <t, t_0_enabler>
9          create arc <disabler,t>
10     else:
11         for each sequence s that t is a part of:
12             if t is the last transition of the sequence s:
13                 create arc <t, disabler>
14             else:
15                 if enabler place of successor(t) exists:
16                     create arc <t,successor(t)_enabler>
17                 else
18                     create place <successor(t)_enabler, 0 tokens>
19                     create arc <t, succesor(t)_enabler>
20                     create arc <successor(t)_enabler, successor(t)>
```

Figure 6.9: Sequence Algorithm 1

6.2.2 Case 2: Multiple, Non-Diverging Causal Sequence Constraints

When allowing more than one sequence constraint to be defined, the concepts of overlapping and diverging sequences now play a role. For this case, however, diverging sequences are not yet considered. It was found that when not considering diverging sequences, Sequence Algorithm 1 still suffices.

Example 6.6
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.10. Let C_s denote the causal sequence constraints on $\mathcal{O}$, where $C_s(T6) = \langle T1, T4, T5\rangle$, where $C_s(T7) = \langle T2, T4, T5\rangle$ and $C_s(T8) = \langle T3\rangle$. Figure 6.11 shows the result of applying the enabling constraint algorithm on $\mathcal{O}$.

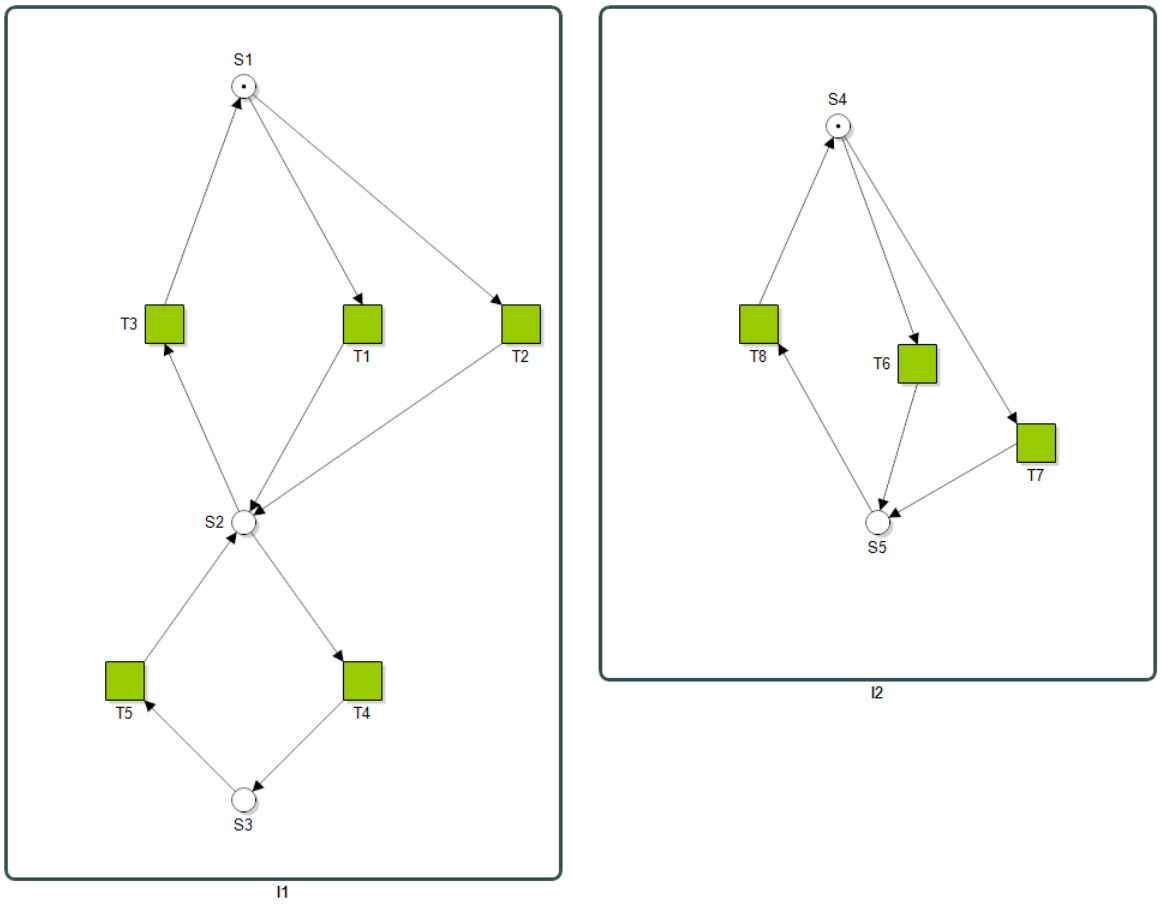

Figure 6.10: Component $\mathcal{O}$

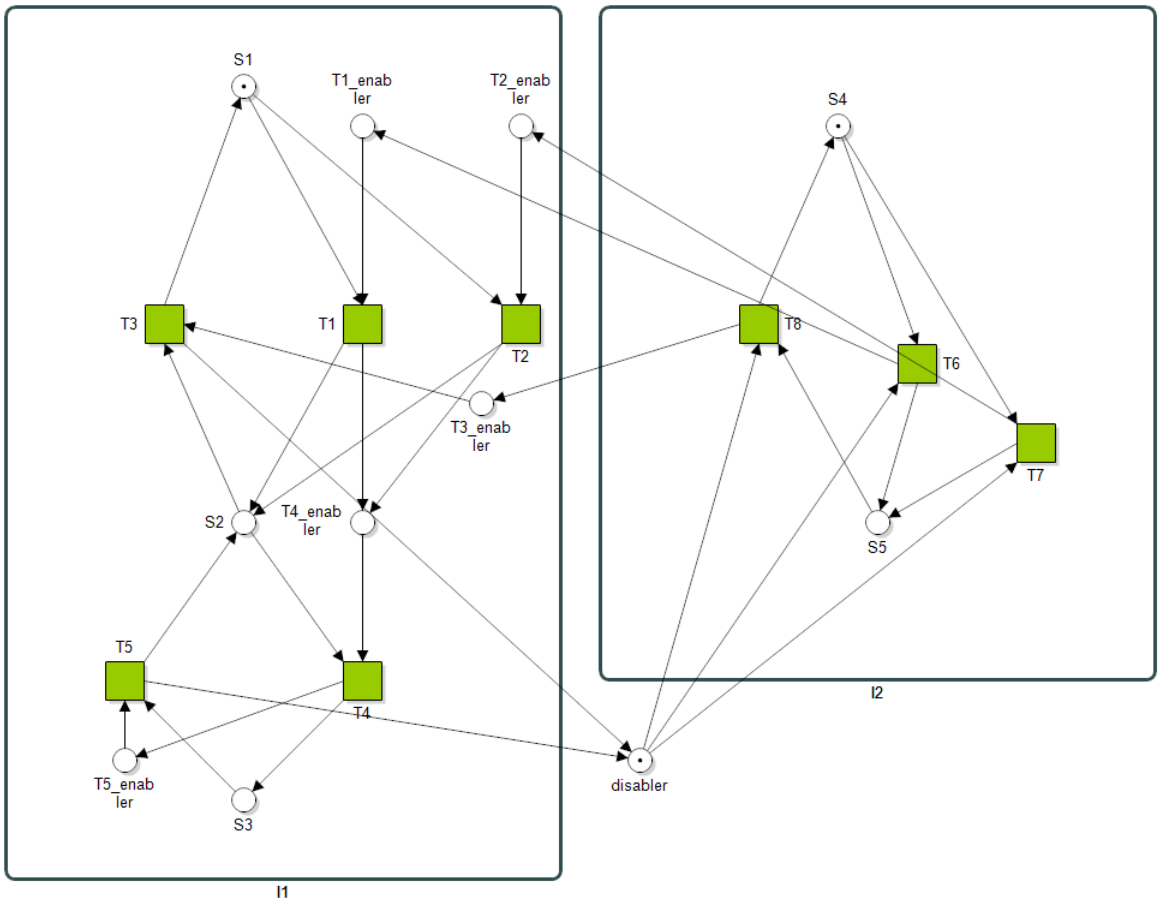

Figure 6.11: Resulting net after applying the sequence constraint algorithm of Figure 6.9 on $\mathcal{O}$, with the following constraints: $C_s(T6) = \langle T1, T4, T5\rangle$, $C_s(T7) = \langle T2, T4, T5\rangle$ and $C_s(T8) = \langle T3\rangle$

6.2.3 Proof sketches for Sequence Algorithm 1

It now needs to be shown that after applying the Sequence Algorithm of Figure 6.9, the resulting net satisfies the causal sequence constraint for a set of non-diverging sequences. The following places and transitions are used in all of the proofs in this section. Let $\mathcal{O}$ be a component, and t be an activation transition part of $N \in \mathcal{O}$, and an arbitrary sequence of consequence transitions defined by $C_s(t)$. Let t_0 and t_{final} be first and last transition of the sequence $C_s(t)$ respectively. Let m_{act} and m_{final} denote the resulting markings after the firing of t and t_{final} respectively. Let p_d be the disabler place introduced on Line 1 of Sequence Algorithm 1. Figure 6.12 illustrates all places, transitions, and arcs used in the proofs.

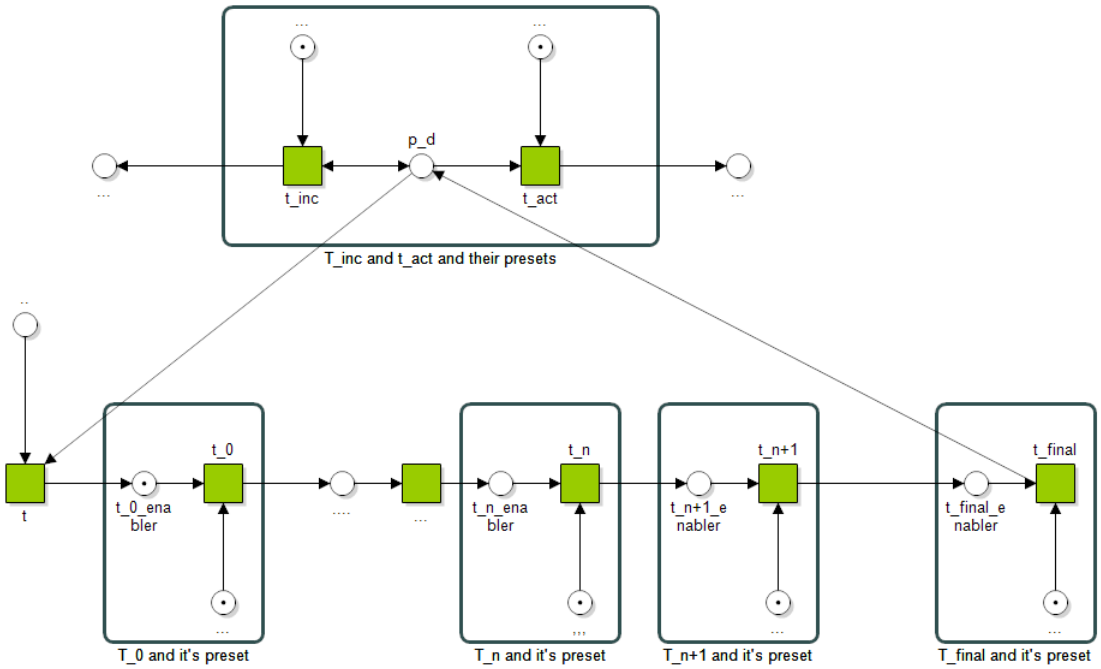

Figure 6.12: Visualization of all places and transitions used in the proof sketches

Lemma 6.7
Any free transition t_{free} or activation transition t_{act} cannot be enabled in m_{act}. For the enabledness of both t_{free} and t_{act} part of an interface N, we must consider the following places in their presets in the marking m_{act}:

1. *A place $p \in P_N$. Only one such p can exist, as N is an S-net and any transition can only be part of one interface.*
2. *A set of places $c1 \in C_e(t_{free})$.*
3. *A set of places $c1' \in C_d(t_{free})$.*
4. *A set of places $c2 \in C_e(t_{act})$.*
5. *A set of places $c2' \in C_d(t_{act})$.*
6. *p_d, which is disabler place introduced Algorithm 1.*

Proof sketch. Line 4 guarantees t_{free} has the disabler place p_d in its preset. Line 9 then guarantees that t_{act} has the disabler place in its preset. Because there is no step in the algorithm creating an arc from t_{act} to p_d, t_{act} never has p_d in its postset. Therefore, after the firing of t that results in the marking m_{act}, the disabler place p_d can never have a token in m_{act}. Therefore, neither t_{free} nor t_{act} can be enabled in m_{act}, regardless of the contents of the places $c1$, $c1'$, $c2$, $c2'$, and p.

Lemma 6.8
The first transition of $C_s(t)$, t_0 is enabled in the marking m_{act}. For the enabledness of t_0 part of an interface N, we must consider the following places its preset in the marking m_{act}:

1. *A place $p \in P_N$. Only one of such p can exist, as N is an S-net and any transition can only be part of one interface.*
2. *A set of places $c \in C_e(t_0)$.*
3. *A set of places $c' \in C_d(t_0)$.*
4. *p_0^e, which is the enabler place of t_0 introduced by Sequence Algorithm 1.*

Proof sketch. Lines 7 and 8 in Sequence Algorithm 1 guarantee that the first consequence transition of a sequence, t_0, has an enabler place p_0^e without an initial token in its preset. Line 9 then guarantees p_0^e is in the postset of the activation transition t.

Using Assumption 5.2, we can assume that $m_{act}(p) = 1$. Using Assumption 5.4, we can furthermore assume that for all places $c \in C_e(t_0)$, $m_{act}(c) = 1$. And lastly, using Assumption 5.5, we can assume

that for all places $c' \in C_d(t_0)$, $m_{act}(c') = 0$. Since p_0^e is in the postset of the activation transition t, p_0^e also has a token in m_{act}. Therefore t_0 is enabled in m_{act}.

Lemma 6.9
Any consequence transition t' that is not t_0 is disabled in m_{act}. For the enabledness of t' part of an interface N, we consider the following places in its preset in the marking m_{act}:

1. *A place $p \in P_N$. This can only be exactly one place, as N is an S-net.*
2. *A set of places $c \in C_e(t')$.*
3. *A set of places $c' \in C_d(t')$.*
4. *p'^e, which is the enabler place of t' introduced by Sequence Algorithm 1.*

Proof sketch. Lines 18 and 19 guarantee that any consequence transition t' has an enabler place without an initial token. Because of Line 20, this enabler place p'^e is not in the postset of t, but rather in the postset of its predecessor transition in the sequence. This means that p'^e cannot have a token in m_{act}. Therefore, t' is disabled in m_{act}.

Lemma 6.10
For any transition $t_n \in C_s(t)$, where $0 \leq n \leq |C_s(t)| - 1$, and t is an activation transition: In any given marking m' that is the result of the firing of t_n, the only transition enabled afterwards is $t_{n+1} \in C_s(t)$. For the enabledness of t_{n+1} part of an interface N, we consider the following places in its preset in the marking m':

1. *A place $p \in P_N$. This can only be exactly one place, as N is an S-net.*
2. *A set of places $c \in C_e(t_{n+1})$.*
3. *A set of places $c' \in C_d(t_{n+1})$.*
4. *p_{n+1}^e, which is the enabler place of t_{n+1} introduced by Sequence Algorithm 1.*

Proof sketch. Line 21 guarantees that the enabler place of t_{n+1}, p_{n+1}^e, is in the postset of t_n. Line 19 furthermore guarantees that there is an arc going from p_{n+1}^e to t_{n+1}. Using Assumption 5.2, we can assume that $m'(p) = 1$. Using Assumption 5.4, we can furthermore assume that for all places $c \in C_e(t_{n+1})$, $m'(c) = 1$. Lastly, using Assumption 5.5, we can assume that for all places $c' \in C_d(t_{n+1})$, $m'(c') = 0$. As p_{n+1}^e is in the postset of t_n, p_{n+1}^e has a token in m', therefore t_{n+1} is enabled in m'.

Since the set of causal sequence constraints are only overlapping, any transition part of a sequence can have at most one sequence successor. Since a transition can only fire after its sequence predecessor has fired, and t_n can only have one possible sequence successor, which is t_{n+1}. t_{n+1} is thus the only transition that is enabled in m'.

Theorem 6.11
A component $\mathcal{O}$ always satisfies a non-diverging, single-stage set of causal sequence constraints after applying Sequence Algorithm 1 to $\mathcal{O}$

Proof sketch. To show that $\mathcal{O}$ satisfies an arbitrary, non-diverging causal sequence constraint after applying Sequence Algorithm 1, it needs to be shown that the property introduced in Definition 4.3 holds. That is, it must be shown that there is exactly one possible firing sequence $C_s(t)$ after any t fires until the last transition of $C_s(t)$ fires.

Following Lemma 6.7, any transition not part of a sequence is disabled after t fires. Furthermore, any transition part of a sequence that is not t_0 is disabled following Lemma 6.9. Because t_0 is guaranteed to be enabled following Lemma 6.8, the only possible transition that be enabled after t fires is t_0. Following Lemma 6.10, the nth transition to fire after t fires, must be the nth element of $C_s(t)$, as any transition not part of a sequence is disabled, and any transition t_n part of a sequence can only be enabled after t_{n-1}. Therefore, for any marking m', if $m' \xrightarrow{t}$, there exists exactly one possible firing sequence from m', and that is $C_s(t)$.

6.2.4 Case 3: Multiple Diverging Sequences

For Case 3, multiple possibly diverging sequences are now also considered. In this case, Sequence Algorithm 1 no longer suffices, and Sequence Algorithm 2 shown in Figure 6.15 subsumes it. The situation can now arise where a transition part of a sequence is a divergence point. For such cases, a decision place is introduced. For a transition t, this decision place has a transition in its postset for each

possible sequence successors of t. Each of these transitions t' then put a token in the enabler place of its corresponding sequence successor. To then know which of these transitions must be enabled, a place is introduced for each defined sequence. Each activation transition t puts a token in a place corresponding to the sequence $C(t)$, while the last transition of $C(t)$ removes it. By then creating a bidirectional arc between each of the transitions in the postset of the decision place of t, only the correct one is enabled.

Introducing these additional places that indicate which sequence is active does introduce another problem, however. Namely, the fact that the final transitions of each sequence must remove the tokens from these places. In this case, ambiguity may arise even when there is only a set of overlapping sequences. More precisely, a transition t may be the final transition of more than one sequence. Therefore, the decision place must also be used in such cases. For a transition t and its decision place p, there is a transition for each of the sequences that has t as its final transition in the postset of p. Each of these transitions then remove a token from the place indicating which sequence is active, and put back a token in the disabler place.

Example 6.12
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.13. Let C_s denote the causal sequence constraints on $\mathcal{O}$, where $C_s(T6) = \langle T1, T2, T5\rangle$, and $C_s(T7) = \langle T1, T4, T5\rangle$ and $C_s(T8) = \langle T3\rangle$. Figure 6.14 shows the result of applying Sequence Algorithm 2 on $\mathcal{O}$.

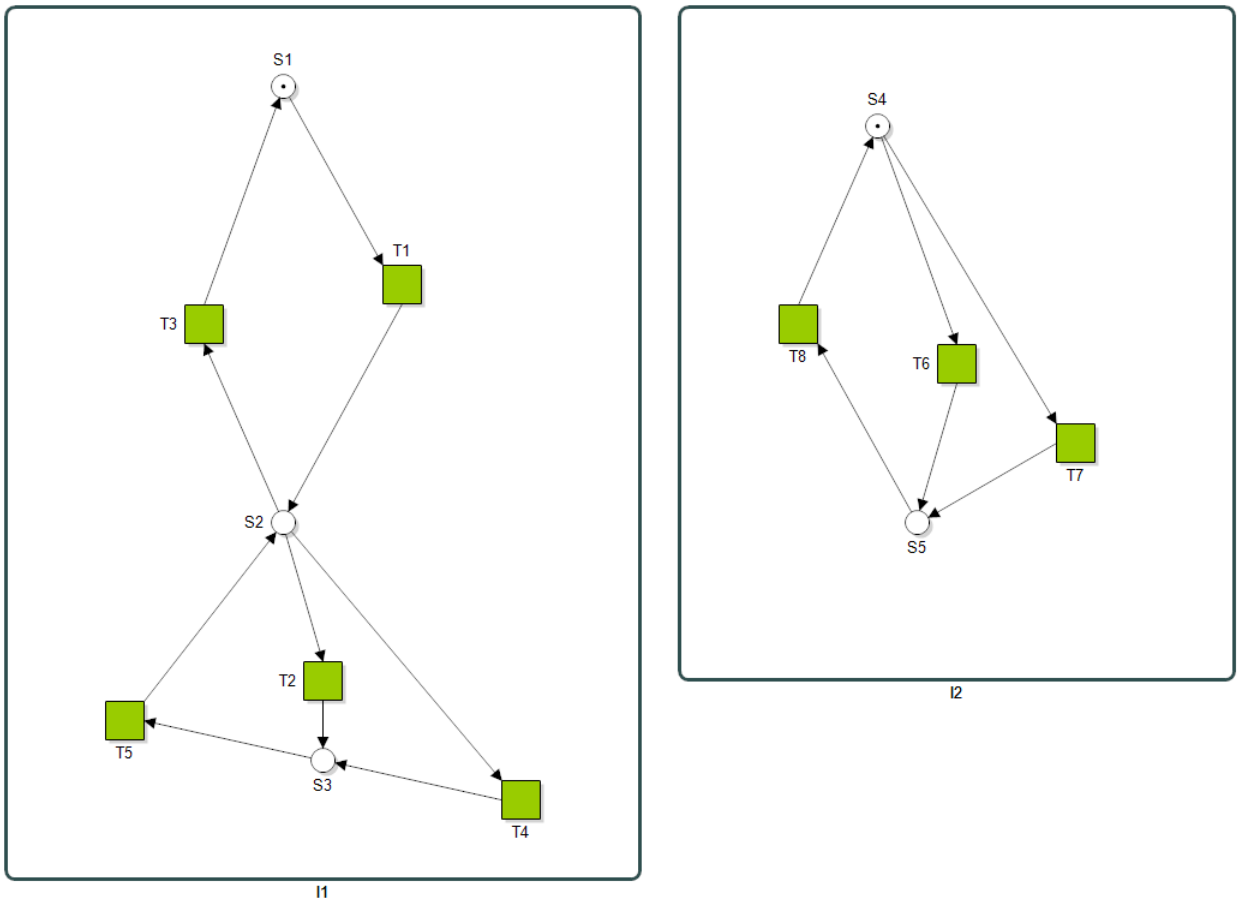

Figure 6.13: Component $\mathcal{O}$

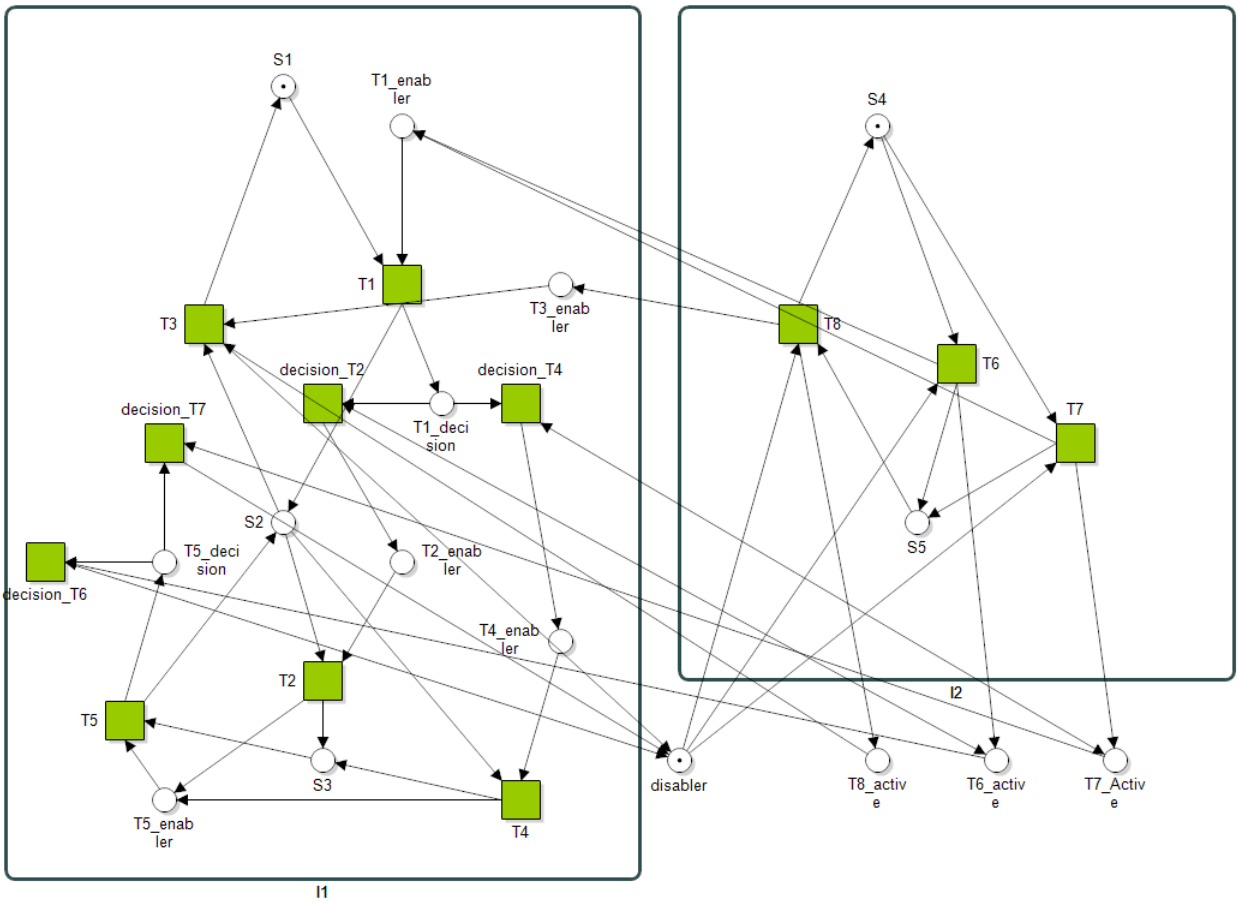

Figure 6.14: Resulting net after applying the sequence constraint algorithm of Figure 6.15 on $\mathcal{O}$, with the following constraints: $C_s(T6) = \langle T1, T2, T5\rangle$, $C_s(T7) = \langle T1, T4, T5\rangle$ and $C_s(T8) = \langle T3\rangle$

In Figure 6.14, we can see the new structure introduced by Sequence Algorithm 2. In the bottom right, we can see that for each of the three sequences defined in Example 6.12, an active place is added indicating whether or not that sequence is active. For the active place of $C_s(T6) = \langle T1, T2, T5\rangle$, we can see that $T6_{active}$ has an outgoing arc to *decision_T6*, an incoming arc from $T6$, which is the final transition of $C_s(T6)$ and $C_s(T7)$, and a bidirectional arc to *decision_T2*. We can see an example of a decision place in the postset of $T5$, as $T5$ is a divergence point. Here, we see that $T5_{decision}$ has two transitions *decision_T6* and *decision_T7* in its postset, one for each of the sequences that of which $T5$ is a final transition. We can see instead of $T5$ now putting back a token in the disabler place, this is now done by the transitions in the postset of its decision place instead. Which of the two transitions fires depends on which sequence is active, indicated by the places in the bottom right. Another divergence point is $T1$, where we can see two transitions in the postset of its decision place, one for each of its sequence successors.

```
1  create place <disabler, 1 token>
2  for each sequence s:
3      create place <s_active, 0 tokens>
4  for each transition t part of an interface skeleton:
5       if t is free:
6          create arc <disabler,t>
7          create arc <t,disabler>
8      else if t is an activation transition for a sequence s starting with t_0:
9          create place <t_0_enabler, 0 tokens>
10         create arc <t_0_enabler, t_0>
11         create arc <t, t_0_enabler>
12         create arc <disabler,t>
13         create arc <t, s_active>
14     else:
15         if t is a divergence point:
16             create place <t_decision, 0 tokens>
17             create arc <t, t_decision>
18             for each successor succ of t as part of the sequence s:
19                 create transition <t_succ>
20                 create arc <t_decision, t_succ>
21                 create bidirectional arc <s_active, t_succ>
22                 if enabler place of successor(t) does not exist:
23                     create place <successor(t)_enabler, 0 tokens>
24                     create arc <successor(t)_enabler, successor(t)>
25                 create arc <t_succ,enabler_succ(t)>
26
27             for each sequence s that t is the final transition of:
28                 create transition <final_s>
29                 create arc <t_decision, final_s>
30                 create arc <s_active, final_s>
31         else:
32             if t is the last transition of the sequence s:
33                 create arc <t, disabler>
34                 create arc <s_active, t>
35             else:
36                 if enabler place of successor(t) does not exist:
37                     create place <successor(t)_enabler, 0 tokens>
38                     create arc <successor(t)_enabler, successor(t)>
39                 else:
40                     create arc <t,enabler_succ(t)>
```

Figure 6.15: Sequence Algorithm 2

6.2.5 Proof sketches for Sequence Algorithm 2

It now needs to be shown that after applying the Sequence Algorithm of Figure 6.15, the resulting net satisfies any causal sequence constraint. Lemmas 6.7, 6.8, and 6.9 still hold for Sequence Algorithm 2. However, for this version of the Sequence Algorithm diverging sequences are now allowed. Therefore, Lemma 6.10 alone no longer suffices and we need Lemmas 6.13 and 6.14 to cover the case of consequence transitions being divergence points.

The following places and transitions are used in the proofs of this section. For a consequence transition t_n, $p_n^{decision}$ refers to the decision place of t_n created on Line 16 in Sequence Algorithm 2. Every transition t in the postset of $p_n^{decision}$ is a *decision transition* of t_n. For a sequence s, s^{active} denotes the place created on Line 3, that is used to indicate whether or not the sequence s is currently active.

Lemma 6.13
For any transition $t_n \in C_s(t)$, where $0 \leq n \leq |C_s(t)| - 1$, t is an activation transition, and t_n is a divergence point: There is a decision transition t' of t_n whose firing leads to a marking m in which only t_{n+1} is enabled.

For the enabledness of t_{n+1} part of an interface N, we consider the following places in its preset in the marking m:

1. *A place $p \in P_N$. This can only be exactly one place, as N is an S-net.*
2. *A set of places $c \in C_e(t_{n+1})$.*
3. *A set of places $c' \in C_d(t_{n+1})$.*
4. *p_{n+1}^e, which is the enabler place of t_{n+1} introduced by Sequence Algorithm 2.*

Proof sketch. Line 19 ensures there exists a decision transition t' of t_n for each sequence successor of t_n. Because of Line 25, p^e_{n+1} is in the postset of t'. For the place $p \in {}^\bullet t_{n+1}$, using Assumption 5.2, we can assume that $m(p) = 1$. Using Assumption 5.4, we can furthermore assume that for all places $c \in C_e(t_{n+1})$, $m(c) = 1$. Lastly, using Assumption 5.5, we can assume that for all places $c' \in C_d(t_{n+1})$, $m(c') = 0$. Because p^e_{n+1} is in the postset of t', p^e_{n+1} will also have a token in the marking m. Therefore, t_{n+1} is enabled in m.

Line 20 then ensures that $p^{decision}_n$ is in the preset of each decision transition. This means that if one of these decision transitions fires, none of the other decision transitions are enabled in the resulting marking. Because any other decision transition could not have fired, any other consequence transition cannot have a token in its enabling place. Therefore, t_{n+1} is the only transition enabled in m.

Lemma 6.14
For any transition $t_n \in C_s(t)$, where $0 \leq n \leq |C_s(t)| - 1$, t is an activation transition, and t_n is a divergence point, Lemma 6.13 showed that there is a decision transition t' of t_n whose firing leads to a marking m' in which only t_{n+1} is enabled. After t_n fires, resulting in the marking m, only t' is enabled in m. For the enabledness of t', we consider the following places in its preset in the marking m:

1. *$p^{decision}_n$. This is the decision place of t_n introduced by Sequence Algorithm 2.*
2. *$p^{active}_{C_s(t)}$, the place indicating that the sequence $C_s(t)$ that t_n is part of is active.*

Proof sketch. Line 17 ensures that $p^{decision}_n$ is in the postset of t_n. Therefore, there is a token in $p^{decision}_n$ in the marking m. Because of Line 21, $p^{decision}_n$ is in both the postset and preset of t'. Knowing now that t is guaranteed to be the last activation transition that has fired, and that the final transition of $C_s(t)$ has not fired yet, Line 21 ensures that there is a token in the place $p^{active}_{C_s(t)}$. Therefore t' is enabled in m. Line 21 also ensures that any other decision transition t'' with $p^{decision}_n$ in its preset will have a different active place s_{active} in its preset. Knowing that any activation transition other than t could not have been the last activation transition to fire, there can be no token in s_{active}. Therefore only t' is enabled in m.

Theorem 6.15
A component always satisfies a set of single-stage causal sequence constraints after applying Sequence Algorithm 2

Proof sketch. To show that $\mathcal{O}$ satisfies an arbitrary causal sequence constraint after applying Sequence Algorithm 2, it needs to be shown that the property introduced in Definition 4.3 holds. That is, it must be shown that there is exactly one possible firing sequence, excluding transitions introduced by the Sequence Algorithm, after any t fires until the last transition of $C_s(t)$ fires. Since Lemmas 6.7, 6.9 and 6.8 still hold, the same proof from Theorem 6.11 can be used to show that the only possible transition that can be enabled after t fires is t_0. Following Lemmas 6.13, 6.14 and 6.10, the nth sequence transition to fire after t fires, must be the nth element of $C_s(t)$, as any transition not part of a sequence is disabled, and any transition t_n part of a sequence can only be enabled after t_{n-1} fires. Therefore, for any marking m', if $m' \xrightarrow{t}$, when excluding any transitions introduced by Sequence Algorithm 2, there exists exactly one firing sequence from m' that is $C_s(t)$.

6.2.6 Case 4: Multiple Diverging, Multi-Stage Sequences

In Case 4, multi-stage sequences are also considered. As such, Sequence Algorithm 2 no longer suffices, and Sequence Algorithm 3 shown in Figure 6.18 must be used instead. The situation may now arise that transitions can be divergence points in the context of a single sequence, and that we can divide a multi-stage sequence into stages. Similarly to Case 3, where we needed to know which sequence was active, we now need to know in which stage a sequence is. We again solve this by using dedicated decision places, and introducing a place for each sequence stage. Using the stage transition sequence introduced in Section 4.3.2, we can then determine the correct ordering of stages, allowing the correct transition to be enabled in the postset of each decision place.

Example 6.16
Let $\mathcal{O} = \{I1, I2\}$ be a component, shown in Figure 6.16. Let C_s denote the causal sequence constraints on $\mathcal{O}$, where $C_s(T1) = \langle T2, T3, T4, T2, T5\rangle$. This sequence has two stages, and its sequence of stage transitions is $\langle T3\rangle$ Figure 6.17 shows the result of applying Sequence Algorithm 2 on $\mathcal{O}$.

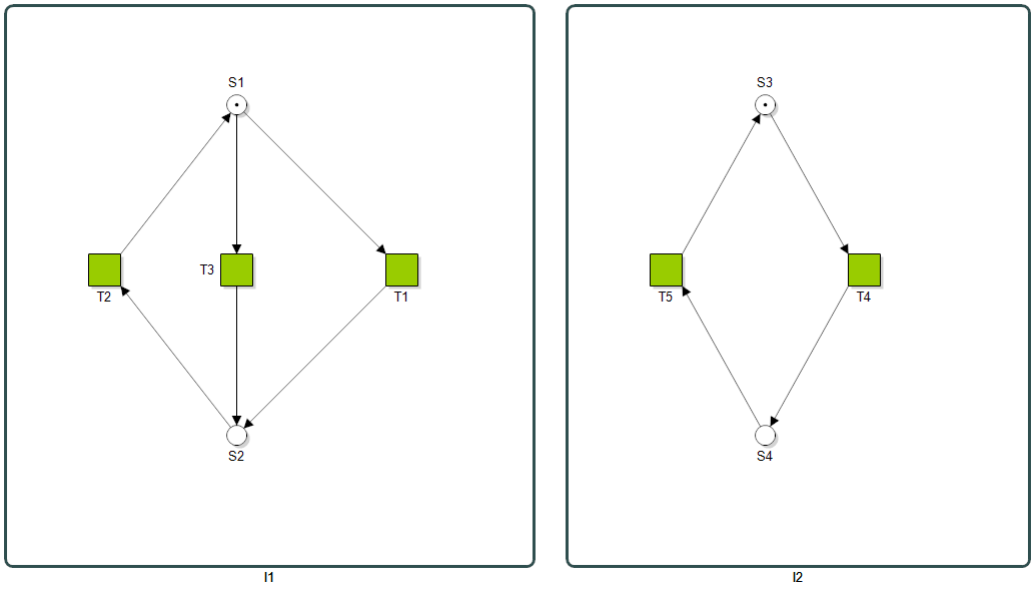

Figure 6.16: Component $\mathcal{O}$

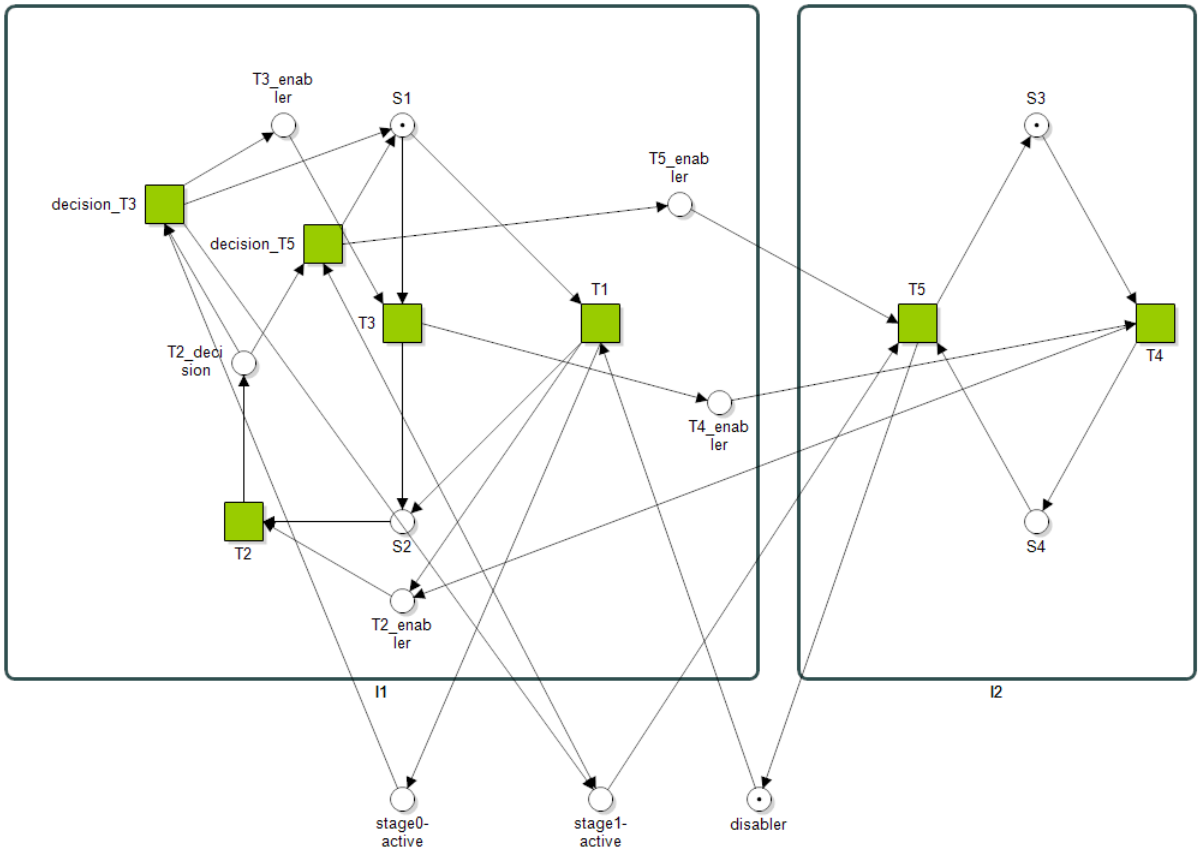

Figure 6.17: Resulting net after applying the sequence constraint algorithm of Figure 6.18 on $\mathcal{O}$, with the following constraint: $C_s(T1) = \langle T2, T3, T4, T2, T5 \rangle$

In Figure 6.17, we can see the new structures introduced by Sequence Algorithm 3. In the bottom, we can see the two places indicating which stage of the sequence $C_s(T1)$ is active. The activation transition $T1$ puts a token in the place *stage0-active*, indicating the first stage of $C_s(T1)$ is active. The consequence

transition $T2$ is a divergence point, and as such there are two decision transitions for each of its sequence successors. As $T3$ is the first sequence successor of $T2$, the first decision transition that should fire is *decision_T3*. This is the case because *decision_T3* has *stage0-active* in its preset, while *decision_T5* has *stage1-active* in its preset. As $T3$ is a stage transition, it is the first transition of the second stage of $C_s(T1)$, which means that *decision_T3* removes a token from *stage0-active*, and puts a token in *stage1-active*. This means that the next time $T2$ fires, only *decision_T5* is enabled. After *decision_T5* fires, only $T5$ will be enabled, and $T5$ in turn removes the token from *stage1-active*, indicating that $C_s(T1)$ is no longer active.

```
1  create place <disabler, 1 token>
2  for each sequence s:
3      for each stage st of C_s(t):
4          create place <st_active, 0 tokens>
5  for each transition t part of an interface skeleton:
6       if t is free:
7          create arc <disabler,t>
8          create arc <t,disabler>
9      else if t is an activation transition for a sequence s starting with t_0:
10         create place <t_0_enabler, 0 tokens>
11         create arc <t_0_enabler, t_0>
12         create arc <t, t_0_enabler>
13         create arc <disabler,t>
14         st1 = first stage of s
15         create arc <t, st1_active>
16
17     if t is a consequence transition:
18         if t is a divergence point:
19             create place <t_decision, 0 tokens>
20             create arc <t, t_decision>
21
22             for each sequence s that t is part of:
23                 for each sequence successor succ of t in s:
24                     create transition <t_succ>
25                     create arc <t_decision, t_succ>
26                     st_succ = stage that succ is part of
27                     st_t = stage that t is part of
28                     if succ is a stage transition:
29                         create arc <st_t_active, t_succ>
30                         create arc <t_succ, st_succ_active>
31                     else:
32                         create bidirectional arc <st_t_active, t_succ>
33
34                     if enabler place of succ does not exist:
35                         create place <succ_enabler, 0 tokens>
36                         create arc <succ_enabler, succ>
37
38                     create arc <t_succ,succ_enabler>
39
40             for each sequence s that t is the final transition of:
41                 create transition <final_s>
42                 create arc <t_decision, final_s>
43                 st_last = last stage of s
44                 create arc <st_last_active, final_s>
45                 create arc <final_s, disabler>
46
47         else:
48             if t is the last transition of the sequence s:
49                 create arc <t, disabler>
50                 st_last = last stage of s
51                 create arc <st_last_active, t>
52             else:
53                 if enabler place of successor(t) does not exist:
54                     create place <successor(t)_enabler, 0 tokens>
55                     create arc <successor(t)_enabler, successor(t)>
56                 else:
57                     create arc <t,enabler_succ(t)>
```

Figure 6.18: Sequence Algorithm 3

6.2.7 Proof sketches for Sequence Algorithm 3

The introduction of multi-stage sequences means that transitions can be divergence points in the context of a single sequence. Because this expands the notion of what a divergence point is, we need to replace Lemma 6.14 with 6.17.

The following places and transitions are used in the proofs of this section. For a consequence transition t_n, $p_n^{decision}$ refers to the decision place of t_n created on Line 19 in Sequence Algorithm 3. For each stage st of a sequence s, p_{st}^{active} denotes the place created on Line 4, and is used to indicate whether or not the stage st of s is active.

Lemma 6.17
For any transition $t_n \in C_s(t)$ part of the stage st, where $0 \leq n \leq |C_s(t)|-1$, t is an activation transition, and t_n is a divergence point, Lemma 6.13 still holds for Sequence Algorithm 3, and showed that there is a decision transition $t'^{decision}$ of t_n whose firing leads to a marking m' in which only t_{n+1} part of the stage st' is enabled. After t_n fires, resulting in the marking m, only $t'^{decision}$ is enabled in m. For the enabledness of $t'^{decision}$, we consider the following places in its preset in the marking m:

1. *$p_n^{decision}$. This is the decision place of t_n introduced by Sequence Algorithm 3.*
2. *p_{st}^{active}, the place indicating that the stage st of $C_s(t)$ is active.*

Proof sketch. Line 17 ensures that $p_n^{decision}$ is in the postset of t_n, therefore there is a token in $p_n^{decision}$ in the marking m. Because of Line 21, $p_n^{decision}$ is in both the postset and preset of $t'^{decision}$. Lines 19-20 guarantee that after t_n fires, there is a token in a place $p_n^{decision}$.

For any other decision transition $t''^{decision}$ with $p_n^{decision}$ in its preset, $t''^{decision}$ also has a place $p_{st''}^{active}$ that is not p_{st}^{active} in its preset. Following Lemma 6.7, any other activation transition that could have possibly put a token in the initial stage of another sequence could not have fired. If t_{n+1} is a stage transition, then Lines 28-30 guarantee p_{st}^{active} is in the preset of $t'^{decision}$, and that $p_{st'}^{active}$ is in the postset of $t'^{decision}$. If t_{n+1} is not a stage transition, Line 32 then guarantees that p_{st}^{active} is in both the postset and preset of $t'^{decision}$. This guarantees that only one stage can be active at a time, and that a new stage can only be activated after a stage transition is fired. With t_n being part of the stage st, it is then guaranteed that p_{st}^{active} and only p_{st}^{active}, will have a token in the marking m. As a result ,$t''^{decision}$ cannot be enabled in m. Therefore, only $t'^{decision}$ is enabled in m'.

Theorem 6.18
A component always satisfies a set of potentially overlapping, diverging, multi-stage causal sequence constraints after applying Sequence Algorithm 3

Proof sketch. To show that $\mathcal{O}$ satisfies an arbitrary causal sequence constraint after applying Sequence Algorithm 3, it needs to be shown that the property introduced in Definition 4.3 holds. That is, it must be shown that there is exactly one possible firing sequence, excluding transitions introduced by the Sequence Algorithm, after any t fires until the last transition of $C_s(t)$ fires. Since Lemmas 6.7, 6.9 and 6.8 still hold, the same proof from Theorem 6.11 can be used to show that the only possible transition that can be enabled after t is t_0. Following Lemmas 6.17 and 6.10, the nth sequence transition to fire after t fires, must be the nth element of $C_s(t)$, as any transition not part of a sequence is disabled, and any transition t_n part of a sequence can only be enabled after t_{n-1} fires. Therefore, for any marking m', if $m' \xrightarrow{t}$, when excluding any transitions introduced by Sequence Algorithm 3, there exists exactly one firing sequence from m' that is $C_s(t)$.

Chapter 7

Case study

This chapter goes over three main topics. First, a case in the form of a ComMA component specification is covered, and the algorithms introduced in Chapter 6 are applied to represent its interface constraints. Secondly, the Neo4J based validation method is introduced. The main contributions here are in the form of Cypher, which is the query language of Neo4J. For each of the constraints, a Cypher query template is introduced, and these are then used to validate the Petri net representation produced for the case. In the implementation, Cypher queries based on these templates are generated automatically from a given ComMA specification. Lastly, the scalability of the methods introduced in Chapter 6 is evaluated, which is done by analyzing the state space of the Petri produced for the case. This is important as the methods of Chapter 6 may produce Petri nets whose state space is too large, in which case it can be infeasible to do any analysis.

7.1 ComMA Specifications of the Case

In this case study, we will look at the ComMA specification of a simple, synthetic model based on an MRI machine. MRI machines operate using superconducting magnets and are cooled at extremely low temperatures, which can only be maintained in a vacuum. The component therefore has a vacuum and a temperature interface, as well as an imaging interface responsible for the imaging. On top of this, there is a monitor interface that monitors the system for any faults. The ComMA definitions of these interfaces are shown in Figures 7.1, 7.2, 7.3 and 7.4, respectively. The Petri net representation of the skeletons of these interfaces is shown in Figure 7.6.

```
1  import "Monitor.signature"
2
3  interface Monitor version "1.0"
4
5  machine StateMachine {
6    initial state Idle {
7
8      transition trigger: checkSystem
9      next state: Checking
10     (tag Idle_checkSystem_Checking)
11   }
12
13   state Checking {
14
15     transition trigger: fault
16     next state: Error
17     (tag Checking_fault_Error)
18
19     transition trigger: finish
20     next state: Idle
21     (tag Checking_finish_Idle)
22
23   }
24
25   state Error {
26     transition do:
27     handleError
28     next state: Checking
29     (tag Error_handleError_Checking)
30   }
31 }
```

Figure 7.1: ComMA monitor interface definition

The monitor interface shown above has three states: Idle, Checking and Error. A checkSystem event will transition the monitor interface from the Idle state to the Checking state. From the Checking state, a fault event will transition the interface to the Error state, while a finish event will transition it back to the Idle state. In the error state, a handleError event will transition the interface back to the checking state.

```
1  import "Vacuum.signature"
2
3  interface Vacuum version "1.0"
4
5  machine StateMachine {
6    initial state Off {
7
8      transition trigger: turnOn
9      next state: On
10     (tag Off_turnOn_On)
11   }
12
13   state On {
14
15     transition trigger: turnOff
16     next state: Off
17     (tag On_turnOff_Off)
18
19     transition trigger: check
20     next state: Checking
21     (tag On_check_Checking)
22
23     transition trigger: check2
24     next state: Checking
25     (tag On_check2_Checking)
26   }
27
28   state Checking {
29     transition do:
30     done
31     next state: On
32     (tag Checking_done_On)
33   }
34 }
```

Figure 7.2: ComMA vacuum interface definition

The Vacuum interface shown above has three states: Off, On and Reading. A turnOn event will transition the vacuum interface from the Off state to the On state. From the On state, a check or check2 event will transition the interface to the Checking state, while a turnOff event will transition it back to the Off state. In the Checking state, a done event will transition the interface back to the On state.

```
1  import "Temperature.signature"
2
3  interface Temperature version "1.0"
4
5  machine StateMachine {
6    initial state Off {
7
8      transition trigger: turnOn
9      next state: On
10     (tag Off_turnOn_On)
11   }
12
13   state On {
14
15     transition trigger: turnOff
16     next state: Off
17     (tag On_turnOff_Off)
18
19     transition trigger: read
20     next state: Reading
21     (tag On_read_Reading)
22
23   }
24
25   state Reading {
26     transition do:
27     done
28     next state: On
29     (tag Reading_done_On)
30   }
31
32 }
```

Figure 7.3: ComMA temperature interface definition

The Temperature interface shown above has three states: Off, On and Reading. A turnOn event will transition the vacuum interface from the Off state to the On state. From the On state, a read event will transition the interface to the Reading state, while a turnOff event will transition it back to the Off state. In the reading state, a done event will transition the interface back to the On state.

```
1  import "Imaging.signature"
2
3  interface Imaging version "1.0"
4
5  machine StateMachine {
6    initial state Off {
7
8      transition trigger: turnOn
9      next state: On
10     (tag Off_turnOn_On)
11   }
12
13   state On {
14
15     transition trigger: turnOff
16     next state: Off
17     (tag On_turnOff_Off)
18
19     transition trigger: image
20     next state: Processing
21     (tag On_image_Processing)
22
23     transition trigger: image2
24     next state: Processing
25     (tag On_image2_Processing)
26
27   }
28
29   state Processing {
30     transition do:
31     done
32     next state: On
33     (tag Processing_done_On)
34   }
35 }
```

Figure 7.4: ComMA Imaging interface definition

The Imaging interface shown above has three states: Off, On and Processing. A turnOn event will transition the vacuum interface from the Off state to the On state. From the On state, an image or image2 event will transition the interface to the Processing state, while a turnOff event will transition it back to the Off state. In the Processing state, a done event will transition the interface back to the On state.

```
1  import "Temperature/Temperature.interface"
2  import "Vacuum/Vacuum.interface"
3  import "Imaging/Imaging.interface"
4  import "Monitor/Monitor.interface"
5  component testModule
6
7  provided port Temperature iTemperaturePort
8  provided port Imaging iImagingPort
9  provided port Vacuum iVacuumPort
10 provided port Monitor iMonitorPort
11
12 functional constraints
13
14 causal-seq init_sequence {
15   iImagingPort::Off_turnOn_On
16   where iTemperaturePort in Off and iVacuumPort in Off
17   leads-to
18   iVacuumPort::Off_turnOn_On
19   iTemperaturePort::Off_turnOn_On
20
21 }
22
23 causal-seq image {
24   iImagingPort::On_image_Processing
25   where iTemperaturePort in On and iVacuumPort in On
26   leads-to
27   iTemperaturePort::On_read_Reading
28   iTemperaturePort::Reading_done_On
29   iVacuumPort::On_check_Checking
30   iVacuumPort::Checking_done_On
31 }
32
33 causal-seq image2 {
34   iImagingPort::On_image2_Processing
35   where iTemperaturePort in On and iVacuumPort in On
36   leads-to
37   iTemperaturePort::On_read_Reading
38   iTemperaturePort::Reading_done_On
39   iVacuumPort::On_check2_Checking
40   iVacuumPort::Checking_done_On
41 }
42
43 causal-seq error {
44   iMonitorPort::Checking_fault_Error
45   leads-to
46   iMonitorPort::Error_handleError_Checking
47   iMonitorPort::Checking_finish_Idle
48 }
49
50 invar error_disable {
51   iImagingPort::On_image_Processing
52   where iMonitorPort in Error
53 }
54
55 invar error_disable2 {
56   iImagingPort::On_image2_Processing
57   where iMonitorPort in Error
58 }
```

Figure 7.5: ComMA component definition

Above is the component definition of the example specification. As can be seen in the definition, all interfaces part of the component are provided interfaces. The first causal sequence constraint init_sequence states that a turnOn in the Imaging interface must be followed by an turnOn in the Vacuum interface, followed by a turnOn in the Temperature Interface. The second causal sequence constraint states that an Image in the image interface must be followed by a read in the temperature interface, followed by a done in the Temperature interface, followed by a check in the vacuum interface, finally followed by a done in the Vacuum interface. The third sequence image2 is almost the same as the image sequence, meaning that they are overlapping. Only the third transition in the sequence is different, as a check2

is now required to happen instead, making the sequences diverging. The fourth sequence states that a fault in the monitor interface must be followed by a handleError in the monitorInterface, followed by a finish in the monitor interface. Followed are two disabling constraints, for which the "invar" keyword is used in ComMA. These disabling constraints state that the image and image2 transition in the Imaging interface cannot happen if the Monitor interface is in the Error state.

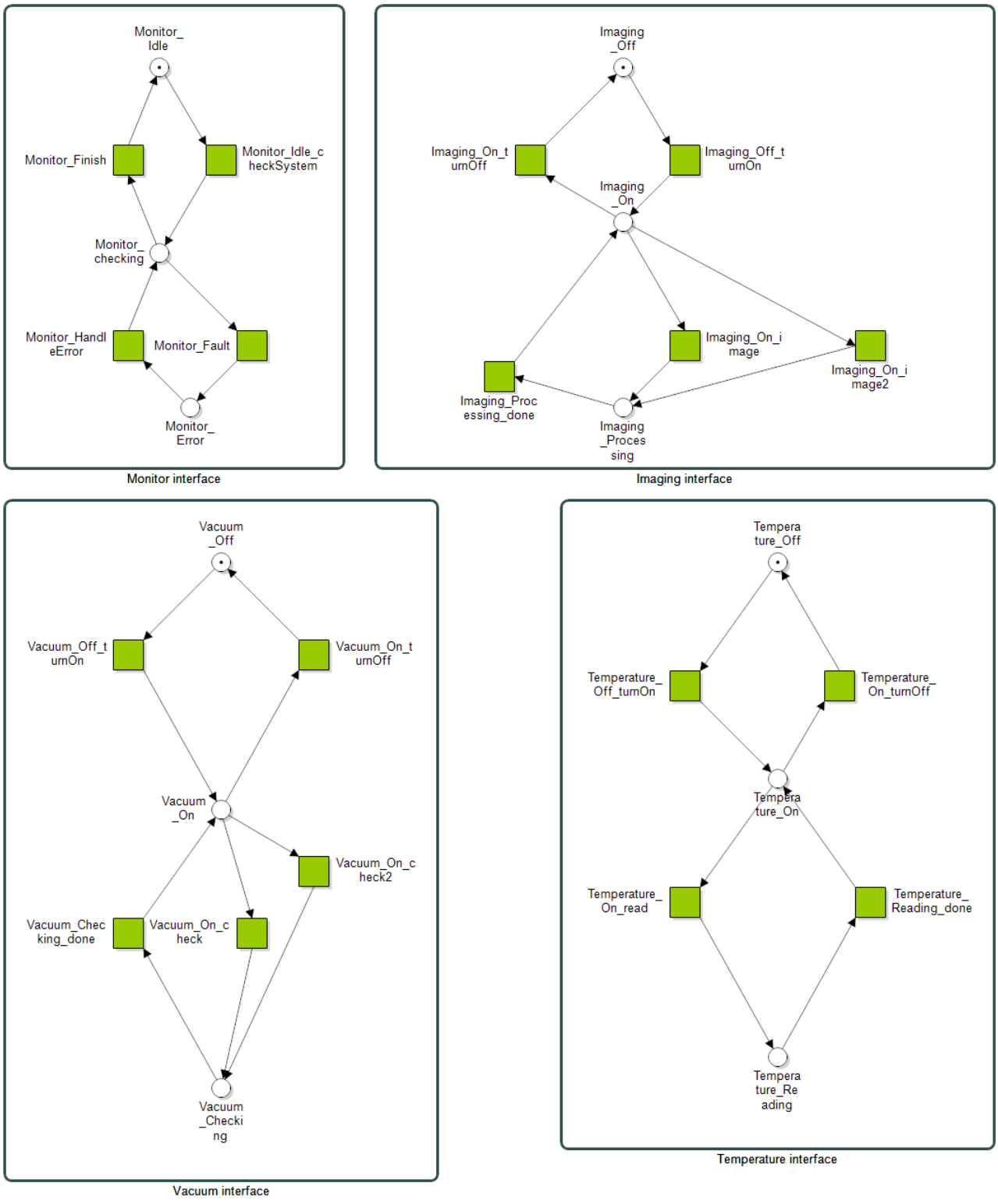

Figure 7.6: Interface skeletons of the interfaces of the example component

The figure above shows the interface skeletons of each of the interfaces represented as a Petri net, which is the result of an existing generator in ComMA.

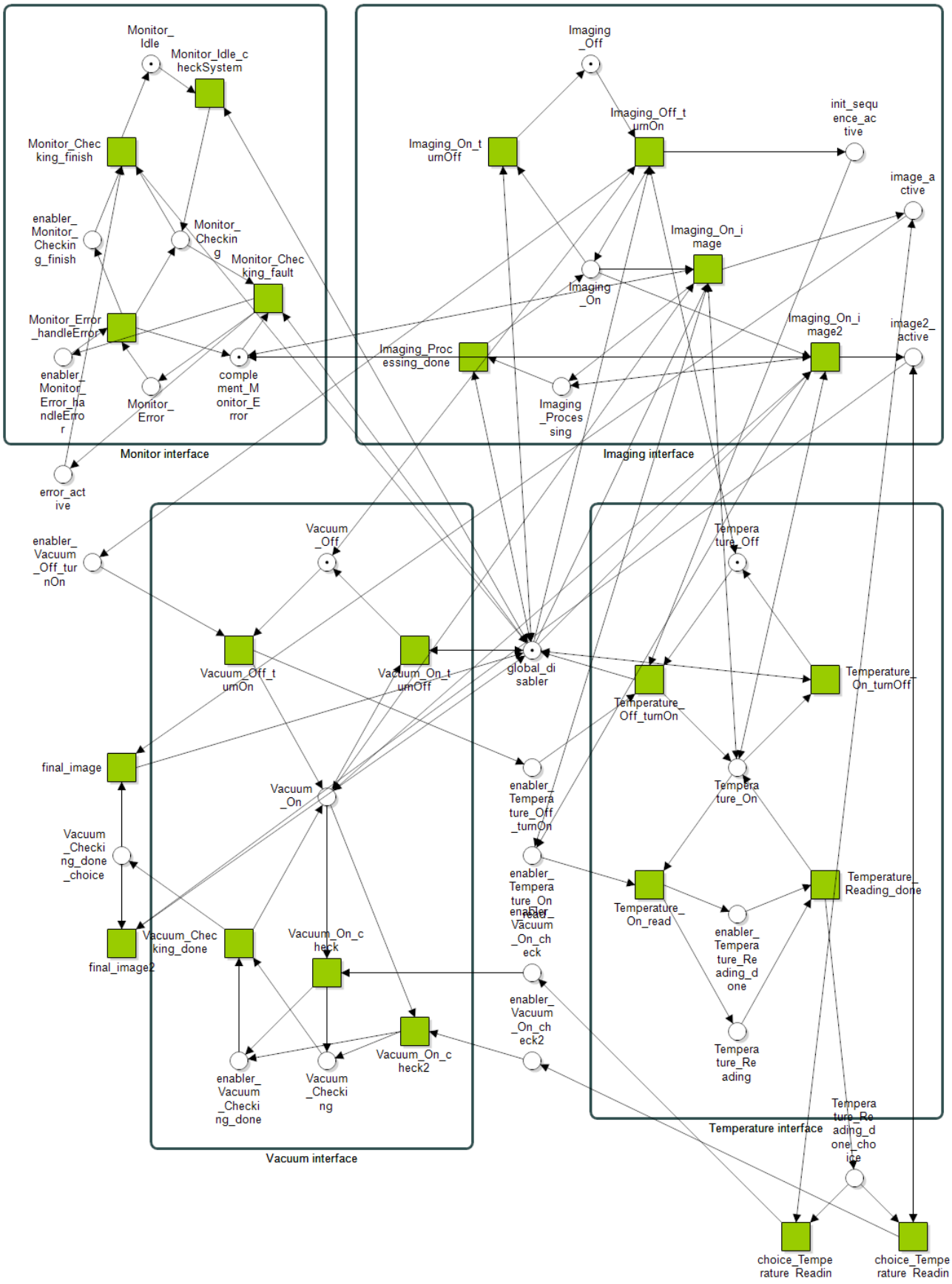

Figure 7.7: Resulting Petri net after applying the sequence constraint algorithm of Figure 6.18 on the example component

7.2 Validation

Validation is done on the reachability graph of the Petri net representation of a component. Although there are proof sketches for each of the algorithms, it is always possible that the proof contains errors. Therefore, some redundancy with regards to validation never hurts to build additional confidence in the solutions. Using reachability analysis, we can verify that the properties introduced in Chapter 4 do indeed hold. Several options were explored, including Tapaal, which is a tool for editing, simulating, and verifying Petri nets [26]. Using Tapaal, the verification of enabling constraints and disabling constraints was possible. However, it was not expressive enough to be able to verify causal sequence constraints. This is because the query language does not allow for the verification of the existence and nonexistence of specific firing sequences, which is required to verify causal sequence constraints. In the end, the graph database platform Neo4J was found to be sufficiently expressive to cover all three constraints.

7.2.1 Neo4J

Neo4J [1] is a graph database platform, used to both store and leverage relationships between data. Graphs consist of nodes, and nodes can have relationships with other nodes. A Petri net reachability graph could then be represented as a Neo4J graph. A node represents a marking, and a relationship to another node represents a transitions that leads the net to another marking. Nodes in Neo4J can furthermore have properties, and these can be used to represent the marking each node represents. A node property is simply a key-value pair, thus a marking can be represented by letting place names map to a number of tokens.

From the internal representation used to generate the net, a set of Cypher scripts is then also generated that verify the properties of Definitions 4.1, 4.2 and 4.3 for each of the defined constraints.

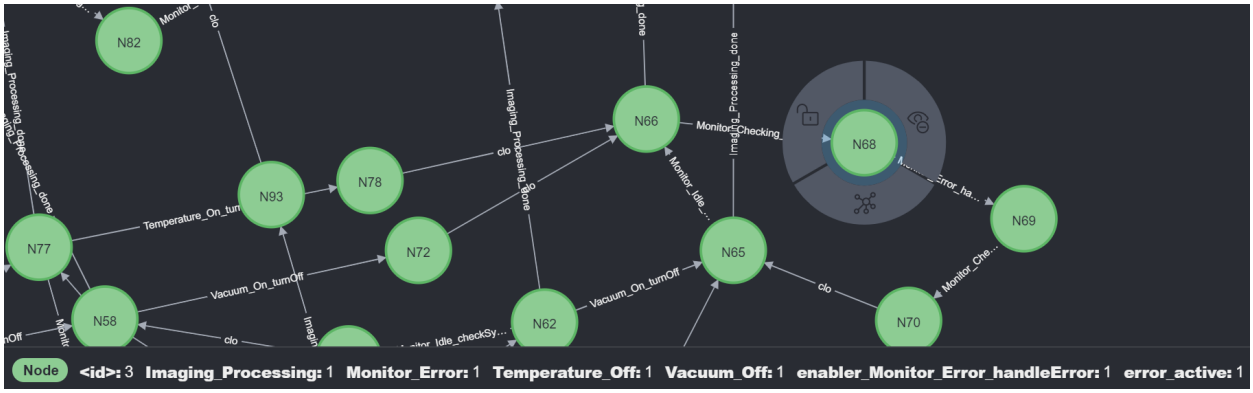

Figure 7.8: Part of a neo4j reachability graph

Cypher is Neo4J's SQL-like query language that uses pattern matching to match for certain nodes and relationships.

```
1 MATCH (n)
2 WHERE (n)-[:Transition]->()
```

The above Cypher matches for all nodes n, such that n has an outgoing relationship called Transition. If we then want to check the marking of all nodes n with an outgoing transition TurnOff to see if the place *On* has at least one token, we can do the following:

```
1 MATCH (n)
2 WHERE (n)-[:TurnOff]->()
3 RETURN exists(n.On)
```

Where exists(n.On) returns True if the node n has a property with the key On.

7.3 Cypher Templates Used to Validate the Case

This section introduces the Cypher templates used for the validation of each interface constraint. Each template is followed by a corresponding Cypher query that was used to validate the case.

[1] Neo4J documentation

7.3.1 Enabling Constraint

For an enabling constraint C_e and a transition t on which the constraint is applied, the template for generating the validation Cypher script is as follows:

```
1 MATCH (n)
2 WHERE (n)-[:<t>]->()
3 RETURN <For each c in Ce(t): exists(n.<c>)>
```

The script matches for some node n, where n has an outgoing edge named t. Any n that matches, must then have a property for each place c in $C_e(t)$, indicating that c has a token. Note that this merely checks for the existence of a property, and not the value of the property itself. This is done because each node only contains a property for each of the places that has a token, not for all possible places of the net, which is a more compact representation. We can do this because the nets we are verifying are safe, which means that any place can either have one token, or none at all.

The following Cypher query was used to verify the enabling constraint on the turnOn transition part of the case:

```
1 MATCH (n)
2 WHERE (n)-[:Imaging_Off_turnOn]->()
3 RETURN exists(n.Temperature_Off) AND
4  exists(n.Vacuum_Off)
```

If the result is valid then for each of n found in each of the above Cypher queries, True must be returned. The rest of the queries for enabling constraints can be found in Appendix B.1 and B.2.

7.3.2 Disabling Constraint

For a disabling constraint C_d and a transition t on which the constraint is applied, the template for generating the validation Cypher script is as follows:

```
1 MATCH (n)
2 WHERE (n)-[:<t>]->()
3 RETURN <For each c in Ce(t): NOT exists(n.<c>) >
```

The Cypher for a disabling constraint is almost exactly the same as for the enabling constraint. The only difference is now that for any node n that matches, n must not have a property for each c in $C_e(t)$. Like with the Cypher for enabling constraints, this is again an existence check, except now it checks for the absence of a property.

The following Cypher query was used to verify the disabling constraint on the image transition part of the case:

```
1 MATCH (n)
2 WHERE (n)-[:Imaging_On_image]->()
3 RETURN  NOT exists(n.Monitor_Error)
```

If the result is valid, then for each of n found in each of the above Cypher queries, True must be returned. The rest of the queries for disabling constraints can be found in Appendix B.3.

7.3.3 Causal Sequence Constraint

The template for validating a causal sequence constraint is more complicated. When analyzing the reachability graph, we first have to consider that when a consequence transition is a divergence point, it is followed by a choice transition that is introduced why by Sequence Algorithm 2 or 3. If we then want to match the full path of a sequence, we need to include these transitions in the matching expression. For a causal sequence constraint $C_s(t)$ where t is an activation transition, let S_d denote the sequence consisting of $C_s(t)$ and all the choice transitions corresponding to each element of $C_s(t)$. For each $t_n \in S_d$ where $n \in \mathbb{N}$, t_{n+1} is a choice transition of t_n if it has one. It does not matter what t_{n+1} actually is, as t_{n+1} is just there to account for the fact that t_n is not directly followed by its sequence successor in the reachability graph. For a causal sequence constraint C_s and a transition t on which the constraint is applied, the template for generating the validation Cypher script is as follows:

```
1  MATCH activation = (n)-[a]->()
2  WHERE a.name='Monitor_Checking_fault'
3  WITH activation, n,a
4  MATCH sequence = (n)-[a]->(m)- <for i from 0 to |S_d|: [e_i]-(n_i)-> > [e6]->(final)
5  WHERE <for each t_n in C_s(t): e_j.name=t_n> //here, j is the index of t_n in S_d
6  WITH activation, sequence, m, e0, <for i from 0 to |S_d|: n_i > ,final
7  OPTIONAL MATCH invalid = ()-[e0]->()-[*1..|S_d|]->(ifinal)
8  WHERE <for i from 0 to |S_d|: ifinal <> n_i > AND ifinal <> final
9  WITH activation, sequence,invalid
10 RETURN [seq IN RELATIONSHIPS(sequence) | seq.name] AS pathValid, [seq IN
       RELATIONSHIPS(invalid) | seq.name]  AS PathsInvalid
```

It is important to clarify that for each e_j generated on Line 2, and each e_i generated on Line 1, $i = j$, as they refer to the same consequence transition.

When validating sequences, a chain of nodes and transitions must now be checked. That is, for any activation transition t, there must exist at least one path in the reachability graph that containts the exact firing sequence $C_s(t)$. It must then also be validated that no invalid paths exist. More specifically, the existence of branching in the previously found valid paths must be checked. This is because branches indicate that more than one transition is enabled while a sequence is active, which is a violation of the property defined in Definition 4.3. This is validated by matching any node that has t as an outgoing arc, and then checks for all paths with at most a length of $|C_s(t)|$ that are not $C_s(t)$. The following Cypher query was used to verify the causal sequence constraint on the image transition part of the case:

```
1  // Validate sequence
2  MATCH activation = (n)-[a]->()
3  WHERE a.name='Imaging_On_image'
4  WITH activation, n,a
5  MATCH sequence = (n)-[a]->(m)-[e0]->(n0)-[e1]->(n1)-[e2]->(n2)-[e3]->(n3)-[e4]->(n4)
       -[e5]->(n5)-[e6]->(final)
6  WHERE
7  e0.name='Temperature_On_read' AND
8  e2.name='Temperature_Reading_done' AND
9  e4.name='Vacuum_On_check' AND
10 e5.name='Vacuum_Checking_done'
11 WITH activation, sequence, m, e0, n0 , n1 , n2 , n3 , n4 , n5  ,final
12 OPTIONAL MATCH invalid = (m)-[e0]->()-[*1..6]->(ifinal)
13 WHERE ifinal <> n0 AND  ifinal <> n1 AND  ifinal <> n2 AND  ifinal <> n3 AND  ifinal
       <> n4 AND  ifinal <> n5 AND   ifinal <> final
14 WITH activation, sequence, invalid
15 RETURN [seq IN RELATIONSHIPS(activation) | seq.name] AS Activation, [seq IN
       RELATIONSHIPS(sequence) | seq.name] AS pathValid,
16 [seq IN RELATIONSHIPS(invalid) | seq.name]  AS PathsInvalid
```

The queries above return three things: Each instance of the activation transition that occurs in the reachability graph, corresponding to each instance a list of relationships that should be the exact sequence of transitions of the causal sequence that is being validated, as well as a list of invalid paths. The first list of relationships can only be null, or the exact matching sequence of transitions. If the result is valid, for each match that was found, the first list of relationships cannot be null, and the list of invalid paths must be null. The rest of the queries for causal sequence constraints can be found in Appendix B.4 and B.5. Figure 7.9 shows the result of a query on a valid model, while Figure 7.10 shows the result for an invalid one.

Figure 7.9: Query result for a valid model

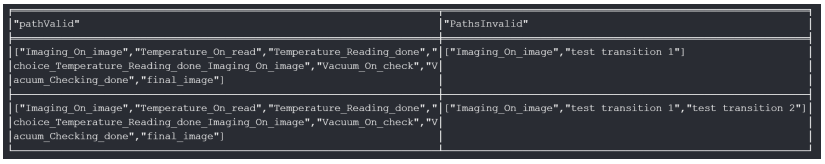

Figure 7.10: Query result for an invalid model

7.4 Scalability

As the purpose of introducing constraints on a component is to reduce the amount of possible behaviour, one would think that the size of the state space would also reduce as a consequence. How much it reduces or if it reduces at all depends on the defined constraints. However, since the algorithms that encode the constraints add new places and transitions to the net, it is possible that the state space of resulting net could increase.

The state space of the net shown in Figure 7.6 should simply be the product of the size of the state spaces of the individual interfaces. Since each interface has three possible markings, the combined state space contains $3^4 = 81$ possible markings. Using Pnat[2], the state space of the net in Figure 7.7 can also be obtained by counting the number of nodes in the reachability graph.

Doing this, it was found that the net of Figure 7.7 has 80 possible markings, which so far suggests that the provided methods are scalable. We can furthermore only look at the places part of a given component. For our case, this would mean counting only the places present in Figure 7.6. When considering only the places part of the component, the number of possible markings is 46. So while only a small decrease of possible markings can be observed for the overall net, we can see a significant reduction in possible behaviour.

This does raise the question of what would happen if the number of constraints were to decrease. Figure 7.11 shows the Petri net resulting from a specification with only one causal sequence constraint defined.

[2]Pnat GitHub

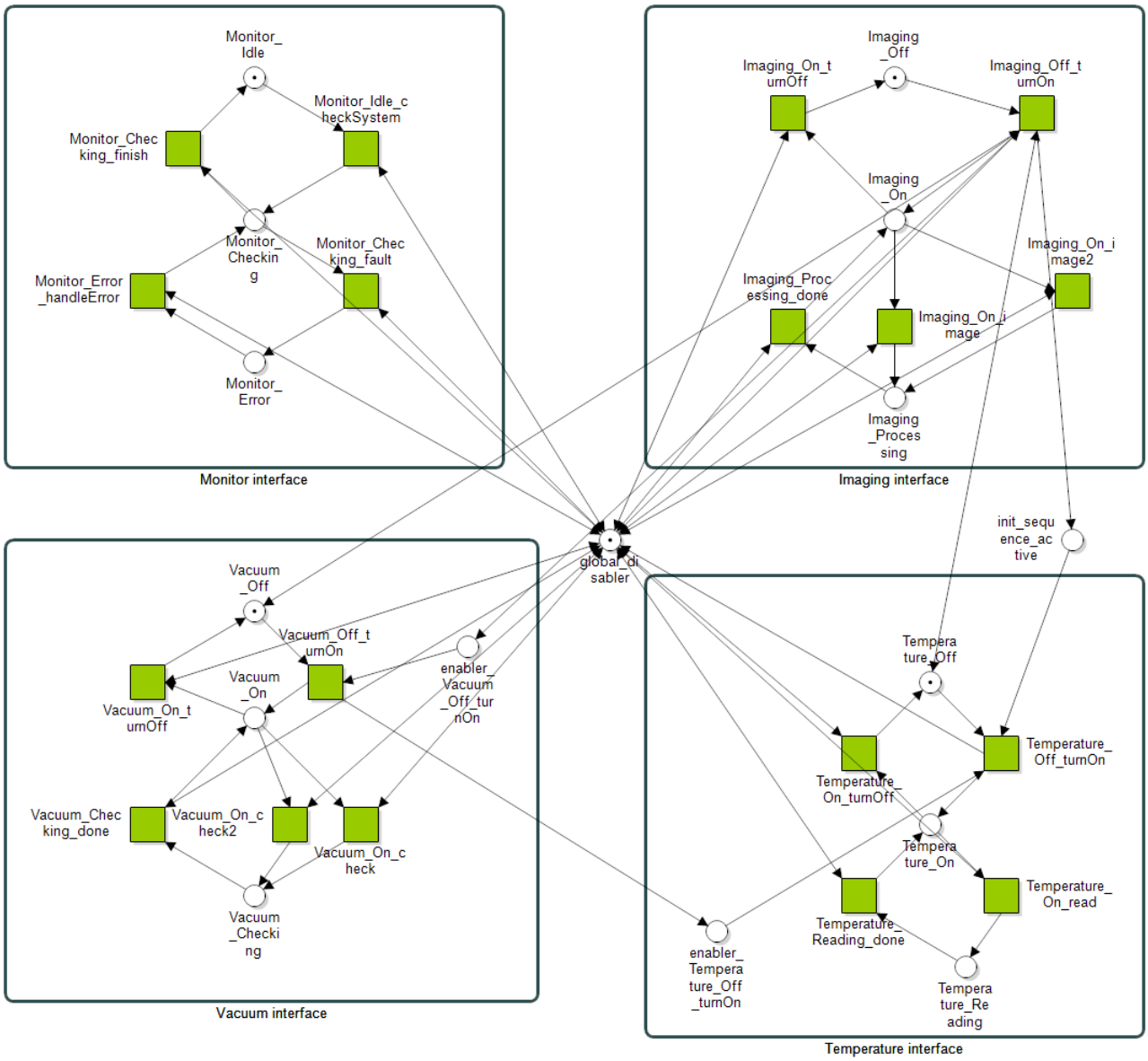

Figure 7.11: Resulting net with only the initialization sequence constraint

The net in Figure 7.11 actually has 87 possible markings, making its state space bigger than the state space of the net in Figure 7.6. When not considering places introduced to encode the constraints, the net in Figure 7.11, still has 81 possible markings. In this particular example, the goal could have been to make sure that the vacuum Interface and the Temperature interface are both in the On state when the Imaging interface is in the On state. However, after the sequence ends, both the Vacuum Interface and the Temperature interface can simply transition to the Off state on their own. By accompanying the startup sequence with a shutdown sequence, we end up with a specification that does achieve the original goal. As a result, the state space is also reduced as expected, with the resulting net only having 39 possible markings.

The specification that resulted in the net of Figure 7.11 is an example of a set of constraints that does not reduce the amount of possible behaviour of the net. It then also makes sense that the resulting net could have an overall larger state space, as the number of possible markings is not reduced by the constraint, while the structure encoding the constraints only allow for a larger number possible markings. So while the state space can become bigger as a result of constraint enconding, its size is still manageable, and it does not suggest that the constraint encodings lead to an exponential increase in the size of the state space.

Chapter 8

Implementation

So far, only the theoretical contributions have been covered. This chapter gives an overview of the implementations that were created, and how these implementations are integrated into, or work together with existing tools to create a validation pipeline. An overview of this pipeline is shown in Figure 8.1.

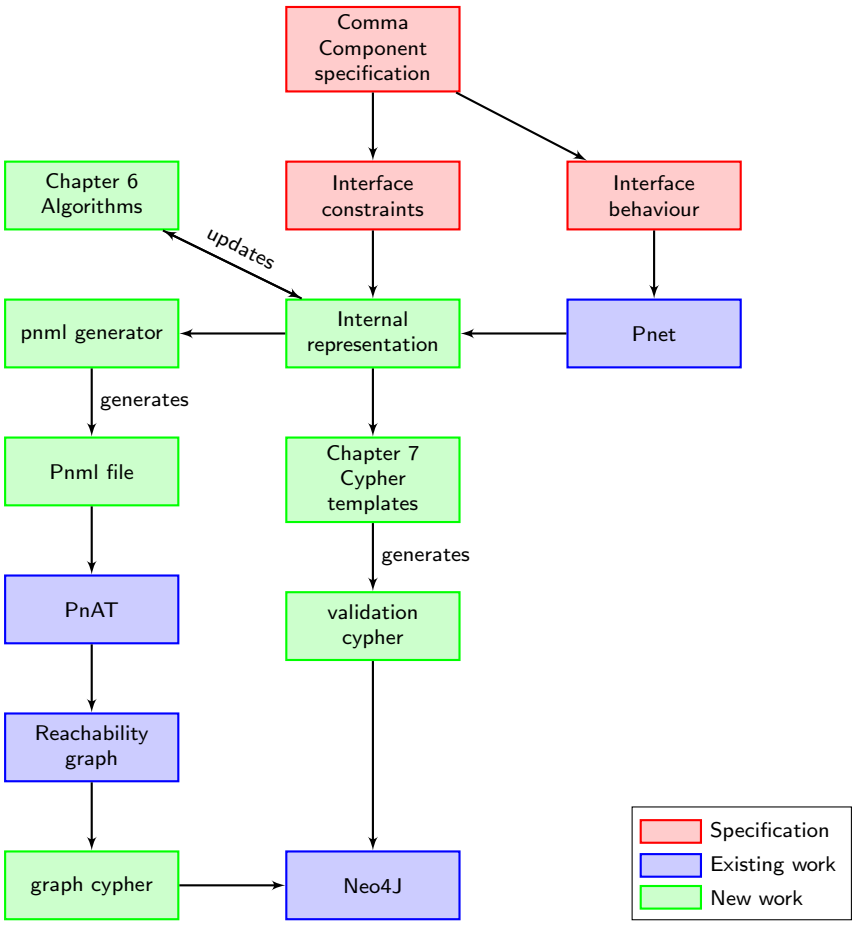

Figure 8.1: Validation pipeline overview

8.1 Validation Pipeline

The implementation starts with the parsing of a ComMA component specification, which results in an internal representation. This internal representation, as well as all the generators, are implemented in Xtend [1], which is the language that a large part of ComMA is implemented in. The internal represen-

[1] Xtend documentation

tation contains three major elements:

1. A Petri net representation. This is simply a set of places, transitions, and arcs.
2. A constraint representation. Given a transition t, indicates which enabling and disabling constraints are applied on t, as well as the sequence of transitions that t leads to if t is an activation transition.
3. Some additional data structures storing things used by the algorithms of Chapter 6, such as the presets and postsets of nodes, as well as the sequences of stage transitions introduced in Section 4.3.2.

The initial Petri net representation can be easily obtained from the Pnet representations of the interfaces of the component. After this is done, the constraints are parsed, and the additional data structures used by the algorithms of Chapter 6 are constructed. As indicated by the bidirectional arc in Figure 8.1, the algorithms of Chapter 6 take the data from the internal representation as an input, and update the internal representation with the set of places, transitions and arcs required to represent the defined constraints. After the internal representation gets updated with these new structures, a pnml generator is first called to produce the pnml [23] file representing the component in a Petri net. Secondly, a Cypher generator that uses the templates introduced in Chapter 7 is called, to produce the Cypher validation files. Having the complete Petri net representation, the pnml generator can simply iterate over all the places, transitions, and arcs in the internal representation to produce the pnml that represents the ComMA component. Similarly, the Cypher generator can iterate over the constraints defined in the constraint representation, producing a Cypher file for each of these constraints.

We now have a Petri net in the form of a pnml file representing the ComMA component, and a set of Cypher files that can be used to validate whether the constraints are represented correctly. What is missing is a Neo4J representation of the reachability graph of the Petri net. To obtain this, the pnml file is first fed into the Pnat tool, which produces a reachability graph. Using a Python script, this reachability graph is parsed and an internal graph representation is created. From this, a Cypher script is generated that constructs a Neo4J graph containing the right nodes, relationships, and node properties needed to represent the reachability graph. The script to create the graph, as well as the validation scripts, can then be imported into the Neo4J desktop environment. Using the Neo4J APOC library, all the scripts could be executed automatically, starting off with the creation of the graph, followed by all the validation scripts.

Chapter 9

Conclusions & Future Work

This concluding chapter starts off with a summary of the main contributions of this thesis. The contributions are discussed for each of the chapters separately. This is followed by several limitations of the existing contributions and points of future work, primarily focused on the guidelines of Chapter 5, the methods introduced in Chapter 6, scalability, and the range of the ComMA language that is not supported.

9.1 Conclusions

The main goal of the thesis was to provide a method for encoding ComMA interface constraints as Petri nets, and to integrate these encodings into existing Petri net representations. To do this, a formalization was first provided for each of the categorized constraints, defining the behaviour of each of these constraints. The concepts of overlapping, diverging, and multi-stage sequences were furthermore introduced to serve as a basis for the algorithms introduced in Chapter 6. From the formalization, a set of assumptions was furthermore defined that act as guidelines for the specification of interface constraints, helping users avoid creating deadlocking specifications.

While multiple options were considered, the goal initially was to create a method that could encode the categorized interface constraints as a P/T net. In the end, it was found that P/T nets were sufficiently expressive to encode the constraints, meaning that the Petri nets produced by the proposed solutions are compatible with existing methods and tools for verification. Two methods were proposed for the enabling constraint and the disabling constraint, respectively, and three for causal sequence constraints. Sequence Algorithm 1 provides a compact encoding of causal sequence constraints, but is limited to non-diverging, single-stage sequences. Sequence Algorithm 2 was then introduced to furthermore deal with diverging sequences. Finally, Sequence Algorithm 3 was introduced to supersede Sequence Algorithm 2, furthermore supporting multi-stage sequences, at the cost of providing a larger, more complex representation. For each of the proposed algorithms, proof sketches were provided that showed that the Petri net encoding of the constraints satisfied the validation properties defined in Chapter 4.

In the case study, several examples were shown on how the size of the state space is affected by different sets of specifications. An example of a specification that did not reduce the possible behaviour of a component was then shown. Using this example, it was furthermore shown that some constraint specifications can cause an overall increase of the state space. This makes it possible for a Petri net representation of a component with interface constraints, to be larger than the state space of a representation without any constraints. However, there was no indication that this increase in state space is exponential, suggesting that the solution is scalable.

The Petri net that represents a component and its interface constraints are automatically generated in ComMA. Using the graph database platform Neo4j, a method for automatically validating this Petri net was provided. A generation template is provided for each of the three classes of constraints. Using these templates, the Cypher queries that can be run in Neo4j are automatically generated in ComMA based on a component's specification. Using a Python script, Cypher that creates a Neo4J graph representing the reachability graph of a Petri net can be generated. Having a Neo4J representation of a reachability graph and the Cypher validation scripts generated in ComMA, it can be verified whether a Petri net correctly encodes the interface constraints defined in a ComMA specification.

9.2 Future work

While some examples were shown on how the structure introduced by the algorithms proposed in Section 6 affects size of the state space, this behaviour has not been formally characterized yet. So while the solution appears to be scalable, formally showing that it is remains an open question. This could also be done empirically by synthetically generating interfaces. The method proposed in [27] can be used as a starting point, and support for the generation of valid interface constraints would have to be added.

There is also room for optimization in the sequence algorithm, namely to reduce the clutter in the resulting net. In the current solution, every free transition is connected to a disabler place with a bidirectional arc. This is in order to guarantee that they cannot fire while a sequence is active. However, not every free transition has to be connected to this disabler place for them to be disabled. Some transitions may be disabled by design, because the interface they are a part of can never be in a state in which it can be fired. It should be possible to do a reachability analysis and detect this, allowing the removal of unnecessary arcs, and making the resulting net more readable. The sequence algorithm could furthermore be optimized to remove unnecessary enabler places, which is the case when all the predecessors of a consequence transition t are part of the same interface as t.

A third point of future work would to extend the list of assumptions given in Chapter 5, as well as giving users feedback regarding on a provided specification. At the moment, following the assumptions is only necessary to avoid deadlock, but does not guarantee deadlock-freedom.

Furthermore, Assumption 5.8 and Assumption 5.10 are based on an algorithm that detects cyclical dependencies, and determines whether a specification can lead to a weakly terminating net. However, a formal proof or argument about its correctness is still lacking. For the other assumptions, only a possible validation property is provided, and methods for verifying these properties given a specification are still missing.

Lastly, the full range of the ComMA language is not supported yet. The current solution does not consider required interfaces and compound transitions. Furthermore, the current solution for causal sequence constraints does not support the recurrence of activation transitions as consequence transitions in the sequence they activate. To make this work with the current solution, it would involve representing a ComMA transition as two separate transitions in a Petri net. For enabling and disabling constraints, only expressions that are purely conjunctive are considered currently, while ComMA also allows for disjunctive expressions. For example, a transition t may depend on place p and p' having at least one token, but could also depend on p or $'$ having at least one token. Lastly, the use of Colored Petri nets to model constraints is a point of future work, which has the advantage of requiring a more compact representation. Furthermore, since the modeling of data aspects in ComMA specifications will require the use of Colored Petri nets, being able to model constraints in Colored Petri nets allows for the representations to be integrated.

Bibliography

[1] (2020). "Htsm systems engineering roadmap," [Online]. Available: https://www.hollandhightech.nl/sites/www.hollandhightech.nl/files/Documenten/Roadmaps/Roadmap-Systems-Engineering-2020.pdf.

[2] C. Y. Baldwin and K. B. Clark, "Modularity in the design of complex engineering systems," in *Complex engineered systems*, Springer, 2006, pp. 175–205.

[3] R. N. Langlois, "Modularity in technology and organization," *Journal of economic behavior & organization*, vol. 49, no. 1, pp. 19–37, 2002.

[4] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman, "Integrating interface modeling and analysis in an industrial setting," Jan. 2017, pp. 345–352. DOI: 10.5220/0006133103450352.

[5] W. Reisig, *Understanding Petri nets. Modeling techniques, analysis methods, case studies. Translated from the German by the author.* Jul. 2013, ISBN: 978-3-642-33277-7. DOI: 10.1007/978-3-642-33278-4.

[6] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, Apr. 1983, ISSN: 0004-5411. DOI: 10.1145/322374.322380. [Online]. Available: https://doi.org/10.1145/322374.322380.

[7] C. Gierds, A. Mooij, and K. Wolf, "Specifying and generating behavioral service adapters based on transformation rules," 2008.

[8] B. Akesson, J. Sleuters, S. Weiss, and R. Begeer, "Towards continuous evolution through automatic detection and correction of service incompatibilities.," in *MDE4IoT/ModComp@ MoDELS*, 2019, pp. 65–72.

[9] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, Sep. 1977, ISSN: 0360-0300. DOI: 10.1145/356698.356702. [Online]. Available: https://doi.org/10.1145/356698.356702.

[10] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem, "Runtime verification of component-based systems," in *Software Engineering and Formal Methods*, G. Barthe, A. Pardo, and G. Schneider, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 204–220, ISBN: 978-3-642-24690-6.

[11] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, 2006, pp. 3–12. DOI: 10.1109/SEFM.2006.27.

[12] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss, "A survey of challenges for runtime verification from advanced application domains (beyond software)," *Formal Methods in System Design*, vol. 54, no. 3, pp. 279–335, Nov. 2019, ISSN: 1572-8102. DOI: 10.1007/s10703-019-00337-w. [Online]. Available: https://doi.org/10.1007/s10703-019-00337-w.

[13] I. Kurtev, J. Hooman, and M. Schuts, "Runtime monitoring based on interface specifications," in *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, J.-P. Katoen, R. Langerak, and A. Rensink, Eds. Cham: Springer International Publishing, 2017, pp. 335–356, ISBN: 978-3-319-68270-9. DOI: 10.1007/978-3-319-68270-9_17. [Online]. Available: https://doi.org/10.1007/978-3-319-68270-9_17.

[14] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *Automated Technology for Verification and Analysis*, S. (Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 64–79, ISBN: 978-3-540-88387-6.

[15] B. Metzler, H. Wehrheim, and D. Wonisch, "Decomposition for compositional verification," in *Formal Methods and Software Engineering*, S. Liu, T. Maibaum, and K. Araki, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–125, ISBN: 978-3-540-88194-0.

[16] R. van Beusekom, J. Groote, P. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, and T. Willemse, "Formalising the dezyne modelling language in mcrl2," English, in *Critical Systems: Formal Methods and Automated Verification*, Germany: Springer, 2017, pp. 217–233, ISBN: 978-3-319-67112-3. DOI: 10.1007/978-3-319-67113-0_14.

[17] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, "The mcrl2 toolset for analysing concurrent systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds., Cham: Springer International Publishing, 2019, pp. 21–39, ISBN: 978-3-030-17465-1.

[18] D. Bera, K. Hee, van, M. Osch, van, and J. Werf, van der, *A component framework where port compatibility implies weak termination*, English, ser. Computer science reports. Technische Universiteit Eindhoven, 2011.

[19] D. Craig and W. Zuberek, "Compatibility of software components - modeling and verification," in *2006 International Conference on Dependability of Computer Systems*, 2006, pp. 11–18. DOI: 10.1109/DEPCOS-RELCOMEX.2006.13.

[20] D. Bera, K. M. van Hee, and J. M. van der Werf, "Designing weakly terminating ros systems," in *Application and Theory of Petri Nets*, S. Haddad and L. Pomello, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 328–347, ISBN: 978-3-642-31131-4.

[21] C. A. Petri, "Kommunikation mit automaten," ger, Ph.D. dissertation, Universität Hamburg, 1962.

[22] J. Baez and J. Master, "Open petri nets," *Mathematical Structures in Computer Science*, vol. 30, pp. 1–28, Apr. 2020. DOI: 10.1017/S0960129520000043.

[23] L. M. Hillah, F. Kordon, L. Petrucci, and N. Trèves, "Pnml framework: An extendable reference implementation of the petri net markup language," in *Applications and Theory of Petri Nets*, J. Lilius and W. Penczek, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 318–327, ISBN: 978-3-642-13675-7.

[24] N. Busi, "Analysis issues in petri nets with inhibitor arcs," *Theoretical Computer Science*, vol. 275, no. 1, pp. 127–177, 2002, ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(01)00127-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030439750100127X.

[25] W. M. P. van der Aalst, C. Stahl, and M. Westergaard, "Strategies for modeling complex processes using colored petri nets," in *Transactions on Petri Nets and Other Models of Concurrency VII*, K. Jensen, W. M. P. van der Aalst, G. Balbo, M. Koutny, and K. Wolf, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 6–55, ISBN: 978-3-642-38143-0.

[26] J. Byg, K. Y. Jørgensen, and J. Srba, "Tapaal: Editor, simulator and verifier of timed-arc petri nets," in *International Symposium on Automated Technology for Verification and Analysis*, Springer, 2009, pp. 84–89.

[27] M. Diallo, B. Akesson, D. Bera, and R. Begeer, "Synthetic portnet generation with controllable complexity for testing and benchmarking," in *International Workshop on Petri Nets and Software Engineering (PNSE)*, 2021.

Appendix A

Cycle Detection Algorithm for Disabling Constraints

```
1  function findCycles(t, visited_places, visited_interfaces):
2      results = ∅
3      new_visited_places = visited_places.add(C_e(t))
4      new_visited_interfaces = visited_interfaces
5      for each c in C_e(t):
6          new_visited_interfaces.add(interface of c)
7      for each c in C_e(t):
8          if(c in visited_places):
9              results.add(True)
10         else if(interface of c in visited_interfaces):
11             v = place in visited_places part of interface of c:
12             if c in t'• for each t' in v•:
13                 for each t in c•:
14                     if(findCycles(t, new_visited_places, new_visited_interfaces):
15                         results.add(True)
16                     else:
17                         results.add(False)
18             else:
19                 results.add(True)
20         else:
21             for each t in c•:
22                 if(findCycles(t, new_visited_places, new_visited_interfaces):
23                     results.add(True)
24                 else:
25                     results.add(False)
26         return (all(results, True) && results != ∅)
27
28
29
30
31 for each i in O:
32     for each p in P_i
33         results = ∅
34         for each t in p•:
35             if(findCycles(t, {p}, {i})):
36                 results.add(True)
37             else:
38                 results.add(False)
39
40         if all(results, True):
41             print(Deadlock. For all transitions in p•, a cyclical dependency was
                   found)
```

Figure A.1: Dependency cycles algorithm for disabling constraints

Appendix B

Cypher Queries Generated for the Case Study

```
1 MATCH (n)
2 WHERE (n)-[:Imaging_On_image]->()
3 RETURN exists(n.Temperature_On) AND
4   exists(n.Vacuum_On)
```

Figure B.1: Cypher validation for the enabling constraint on the image transition

```
1 MATCH (n)
2 WHERE (n)-[:Imaging_On_image2]->()
3 RETURN exists(n.Temperature_On) AND
4   exists(n.Vacuum_On)
```

Figure B.2: Cypher validation for the enabling constraint on the image2 transition

```
1 MATCH (n)
2 WHERE (n)-[:Imaging_On_image2]->()
3 RETURN  NOT exists(n.Monitor_Error)
```

Figure B.3: Cypher validation for the disabling constraint on the image2 transition

```
MATCH activation = (n)-[a]->()
WHERE a.name='Imaging_On_image2'
WITH activation, n,a
MATCH sequence = (n)-[a]->(m)-[e0]->(n0)-[e1]->(n1)-[e2]->(n2)-[e3]->(n3)-[e4]->(n4)
    -[e5]->(n5)-[e6]->(final)
WHERE
e0.name='Temperature_On_read' AND
e2.name='Temperature_Reading_done' AND
e4.name='Vacuum_On_check2' AND
e5.name='Vacuum_Checking_done'
WITH activation, sequence, m, e0, n0 , n1 , n2 , n3 , n4 , n5 ,final
OPTIONAL MATCH invalid = (m)-[e0]->()-[*1..6]->(ifinal)
WHERE ifinal <> n0 AND ifinal <> n1 AND ifinal <> n2 AND ifinal <> n3 AND ifinal
    <> n4 AND ifinal <> n5 AND ifinal <> final
WITH activation, sequence, invalid
RETURN [seq IN RELATIONSHIPS(activation) | seq.name] AS Activation, [seq IN
    RELATIONSHIPS(sequence) | seq.name] AS pathValid,
[seq IN RELATIONSHIPS(invalid) | seq.name] AS PathsInvalid
```

Figure B.4: Cypher validation for the causal sequence constraint on the image2 transition

```
MATCH activation = (n)-[a]->()
WHERE a.name='Imaging_Off_turnOn'
WITH activation, n,a
MATCH sequence = (n)-[a]->(m)-[e0]->(n0)-[e1]->(final)
WHERE
e0.name='Vacuum_Off_turnOn' AND
e1.name='Temperature_Off_turnOn'
WITH activation, sequence, m, e0, n0 ,final
OPTIONAL MATCH invalid = (m)-[e0]->()-[*1..1]->(ifinal)
WHERE ifinal <> n0 AND ifinal <> final
WITH activation, sequence, invalid
RETURN [seq IN RELATIONSHIPS(activation) | seq.name] AS Activation, [seq IN
    RELATIONSHIPS(sequence) | seq.name] AS pathValid,
[seq IN RELATIONSHIPS(invalid) | seq.name] AS PathsInvalid
```

Figure B.5: Cypher validation for the causal sequence constraint on the turnOn transition

```
MATCH activation = (n)-[a]->()
WHERE a.name='Monitor_Checking_fault'
WITH activation, n,a
MATCH sequence = (n)-[a]->(m)-[e0]->(n0)-[e1]->(final)
WHERE
e0.name='Monitor_Error_handleError' AND
e1.name='Monitor_Checking_finish'
WITH activation, sequence, m, e0, n0 ,final
OPTIONAL MATCH invalid = (m)-[e0]->()-[*1..1]->(ifinal)
WHERE ifinal <> n0 AND ifinal <> final
WITH activation, sequence, invalid
RETURN [seq IN RELATIONSHIPS(activation) | seq.name] AS Activation, [seq IN
    RELATIONSHIPS(sequence) | seq.name] AS pathValid,
[seq IN RELATIONSHIPS(invalid) | seq.name] AS PathsInvalid
```

Figure B.6: Cypher validation for the causal sequence constraint on the fault transition