

Partial Specifications of Component-based Systems using Petri Nets

Bart-Jan Hilbrands^{1,2}, Debjyoti Bera¹ and Benny Akesson^{1,2}

¹ESI (TNO), Eindhoven, the Netherlands

²University of Amsterdam, Amsterdam, the Netherlands

Abstract

Component-based architectures are commonly used in industry to manage the increasing complexity of systems. In such architectures, components interact with each other to achieve the desired functionality. They do so by providing and consuming services to and from each other over their defined interfaces. Interfaces play a key role in managing complexity by abstracting away from implementation details and describing only externally observable behavior. To describe the behavior of an interface and analyze them for correctness, formalisms, such as finite state machines and Petri nets, are commonly used. However, components usually have multiple interfaces and their behavior may depend on each other. Most approaches so far have focused on fully specifying the behavior of a component as a single state machine. We consider partial specification of dependencies between interfaces, expressed as a set of functional constraints. In this paper, we present and formalize three commonly occurring functional constraints. Algorithms are proposed to generate Petri nets satisfying each of these constraints.

1. Introduction

Systems are becoming increasingly complex, making them challenging to develop, maintain, and evolve over time [1]. This complexity is usually managed by taking a component-based approach to achieve modularity [2, 3] by decomposing the system into asynchronously communicating *components*. The components provide or consume services to and from each other over *interfaces* to realize the functionality of the system. An interface specification must capture, next to a list of events, also the externally observable behavior by abstracting away from implementation details. It is common practice in industry to define interfaces in natural text, interface description languages (IDLs), or in code (header files), focusing mostly on the static structural aspects, e.g. a list of possible events. Some domain-specific languages (DSLs) exist, such as Component Modelling and Analysis (ComMA) [4] and Dezyne [5], which capture both the structural and behavioral aspects of an interface.


Capturing behavioral aspects of an interface, usually as protocol state machines, has many benefits in a component-based approach. For instance, the behavioral description can be used as contracts between teams allowing them to work independently, or as contracts to external suppliers. They may also be used to generate code stubs, or even online test clients or offline test scenarios to check if the implementation conforms to the specified contracts. Furthermore, since

PNSE'22, International Workshop on Petri Nets and Software Engineering, Bergen, Norway, 2022

 bj.hilbrands@gmail.com (B. Hilbrands)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

interfaces describe the expected interactions between components of a system, it is valuable to check their correctness at design time, e.g. if the interactions are free from deadlocks, livelocks, and unbounded behavior. This allows design errors to be detected early during system specification rather than later during integration and testing, reducing cost.

A component typically has multiple interfaces, and it is quite common that the behavior of one interface on a component depends on the state of another. Such dependencies may be captured as *functional constraints* defined over states and transitions of one or more protocol state machines. For example, a transition in one interface may or may not happen if another interface is in a particular state. Another example could be that a transition in one interface must be followed by a sequence of transitions in other interfaces. Most work in literature have focused on defining the complete behavior of a component as one or more state machines. In practice, defining the complete behavior of a component is usually not possible, due to complexity and limited knowledge of the implementation. Instead, we focus on the fact that interfaces serve as abstractions of the underlying component implementation by allowing for non-determinism, e.g. the response to an initialization request may be either a success or fail, but we do not say when one or the other is possible. Our goal is to exploit this level of abstraction to capture partial specifications of a component as functional constraints over one or more interfaces.

The three main contributions of this paper are: 1) A formalization of three types of commonly occurring functional constraints, namely *enabling*, *disabling*, and *causal sequence constraints*. The formalization describes how each constraint should limit the behaviour of a set of P/T nets, each corresponding to a protocol state machine in an interface. 2) A set of assumptions on specific constraints to avoid creating problematic specifications. 3) A method to represent and incorporate these functional constraints into an existing Petri net representation of component interfaces, in our case generated from a ComMA specification, which enables absence of deadlock, livelock, and unbounded behavior of dependent interfaces to be verified using reachability analysis. The method is accompanied by proof sketches, showing that the results produced by the method satisfy the properties defined in the formalization. Throughout the paper, a case study of a vending machine is used as a running example to demonstrate the approach.

The rest of the paper is organized as follows. Section 2 presents related work, while Section 3 introduces relevant preliminaries, as well as the vending machine case study that is used as a running example. In Section 4, a formalization of the three types of constraints is presented. This is followed by a series of assumptions about the behaviour of a given specification, showing that some specifications of constraints may cause termination issues. Section 5 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 Section 4. Finally, conclusions are drawn in Section 6.

2. Related Work

The main goal of this work is to synthesize, for each specified constraint, additional Petri net constructs between existing nets (each describing an interface protocol state machine) to constrain the space of possible behaviors in their reachability graph. The challenge is to ensure that such a graph correctly encodes the specified constraints. Based on such a reachability

graph, it is possible to verify properties, such as absence of deadlocks, livelocks, and unbounded behavior, at design time.

Since model checking monolithic models describing a system is typically infeasible due to state space explosion, a compositional approach is usually taken. Both [6] and [7] propose such an approach. The work in [8] considers another compositional approach based on interface compliance to ensure that the behavior of the whole is guaranteed to be correct, i.e. absence of deadlocks, livelocks and unbounded behavior. In contrast to the work presented in this paper, all the above approaches focus on fully specifying the behavior of a component as a state machine. The latter goes even further to generate production code from such models. The work in [9] is similar to the work presented in this paper in the sense that a partial specification is translated to a formal specification language in order to do verification. This specification is given in UML-RT [10], and is translated to a CSP [11] model to verify deadlock-freedom.

There are also several works [12, 13, 14] that propose correct by construction methods to design component-based systems. Such an approach avoids the drawbacks of having to model check large models. However, there are two main differences to the approach proposed in this paper. Firstly, they propose design methods that guarantee certain properties by construction, such as absence of deadlocks and livelocks. The second difference is once again in the focus on fully specifying the behavior of a component, as opposed to partial specifications considered in this paper.

In conclusion, no method currently exists to encode commonly occurring functional constraints over interfaces of a component for design-time analysis.

3. Preliminaries

This section introduces the preliminaries for this work. Firstly, Section 3.1 starts off with the introduction of ComMA, the modelling and analysis framework in which the component and interfaces specifications are defined. Secondly, relevant Petri net concepts are introduced in Section 3.2, followed by a description of how ComMA interfaces are currently represented as Petri nets.

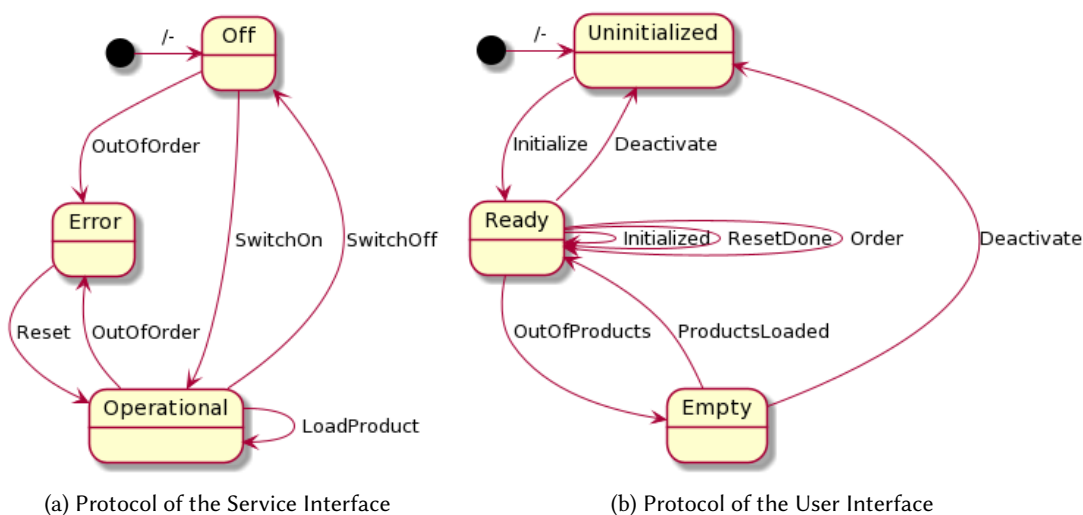
3.1. ComMA Component and Interface Specification

Component Modelling and Analysis (ComMA) [4] is a supporting tool for modelling and analysis of interfaces in component-based systems. It provides a family of domain-specific languages for specification of structure and behavior of component interfaces and their constraints from which documentation, analysis, and monitors can be generated[15]. ComMA is available as an open-source tool under the name Eclipse CommaSuite¹.

The behaviour of an interface is specified in ComMA as a protocol state machine. Figures 1a and 1b show two example interfaces, a Service Interface and a User Interface, that belong to a vending machine component that is used as a running example in this paper. Figure 1a shows an interface with three states: OFF, OPERATIONAL, and ERROR. Through this interface, a technician may turn the machine on or off, and may restock the machine with LOADPRODUCT.

¹<https://www.eclipse.org/comma/>

The machine may also go out of order from any state, in which case it can be reset. Figure 1b shows an interface with the following three states: UNINITIALIZED, READY and EMPTY. Initially, the interface is in the UNINITIALIZED state. After initializing, it is in the ready state. Here, an INITIALIZED or RESETDONE message can be sent, depending on whether the machine was initialized after a RESET or a SWITCHON. If the machine is out of products, it will go to the EMPTY state, and can go back to the READY state if the machine is restocked.



1. LOADPRODUCT shall only happen if the User interface is in the EMPTY state
2. ORDER shall not happen if the Service interface is in the ERROR state
3. SWITCHON shall be followed by INITIALIZE followed by INITIALIZED
4. SWITCHON shall only happen if the User interface is in the UNINITIALIZED state
5. RESET shall be followed by INITIALIZE followed by RESETDONE
6. RESET shall only happen if the User interface is in the UNINITIALIZED state

(c) Functional constraints on the service and user Interface

Figure 1: Vending machine example.

A ComMA component specification describes how events on one interface depend on events of other interfaces. These component specifications are referred to as *functional constraints*.

Functional constraints are of three types. Firstly, the *enabling constraint*, which defines an additional enabling condition for transitions that depends on the current state of one or more other interfaces. For our vending machine, constraints 1, 4 and 6 in Figure 1c are enabling constraints. Secondly, the *disabling constraint*, which defines a *disabling* condition for transitions that depends on the current state of one or more other interfaces. In Figure 1c, this is constraint 2. Lastly, the *causal sequence constraint* requires that when certain transitions on one interface are fired, they are always followed by a sequence of transitions from other interfaces. In our vending machine example, constraints 3 and 5 fall under this category.

3.2. Petri Nets

A Petri net is a tuple $N = (P, T, F)$, where P is the set of *places*; T is the set of *transitions* such that $P \cap T = \emptyset$ and F is the *flow relation* $F \subseteq (P \times T) \cup (T \times P)$. We refer to elements from $P \cup T$ as *nodes* and elements from F as *arcs*. We define the *preset* of a node n as $\bullet n = \{m \mid (m, n) \in F\}$ and the *postset* as $m \bullet = \{n \mid (m, n) \in F\}$. A finite sequence over a set S with length $n \in \mathbb{N}$ is denoted by σ , where σ is a function $\sigma : \{1, \dots, n\} \rightarrow S$. We denote the set of all finite sequences over S by S^* . We say that a function $m : S \rightarrow \mathbb{N}$ 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 $B(S)$.

A P/T net N is a *state machine* (S-net) iff: $\forall t \in T_N : |\bullet t| \leq 1 \wedge |t \bullet| \leq 1$ and all markings have exactly one token [16]. This means there is a one-to-one mapping between the states of the corresponding state machine and places in its S-net representation. Since S-net markings only have a single token, a place p having a token represents the protocol of the interface being in a state s .

While *open Petri nets* (OPNs) [17] are sometimes used to model interfaces and their interactions [14, 12], we do not need them for this paper. This is because the functional constraints are defined on *internal behavior of a single component with multiple interfaces*. Communication with other components through designated interface places introduced by OPNs is hence out of scope. We therefore drop these designated interface places, leaving us only with interface *skeletons*, which we refer to as *interface nets*. A component is therefore defined as a set of S-Nets representing the interfaces that it implements. For a component \mathcal{O} , all interfaces $N \in \mathcal{O}$ 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 $\bigcup_{N \in \mathcal{O}} (T_N)$.

Figure 2 shows the interface skeletons of the two vending machine interfaces represented as S-nets, which is the result of an existing transformation in ComMA. As each transition in Figure 2 represents an event on an interface, we will use the term *event* and *transition* interchangeably. Note that there are no connections between places and transitions in the two interfaces. This is because the transformation does not consider functional constraints during the transformation. That is a novel contribution of this paper.

4. Formalization of Functional Constraints

In this section, we formalize the three types of constraints presented in Section 3: 1) *Enabling constraint*: an event of an interface can only occur if one or more other interfaces are in a certain specified state, 2) *Disabling constraint*: an event of an interface cannot occur if one or more other interfaces are in a certain specified state, and 3) *Causal sequence constraint*: An action must be followed by a sequence of events across multiple interfaces. In this section we will formalize the three constraints as reachability properties, starting with enabling constraints in Section 4.1 and continuing with disabling constraints and causal sequence constraints in Sections 4.2 and 4.3, respectively.

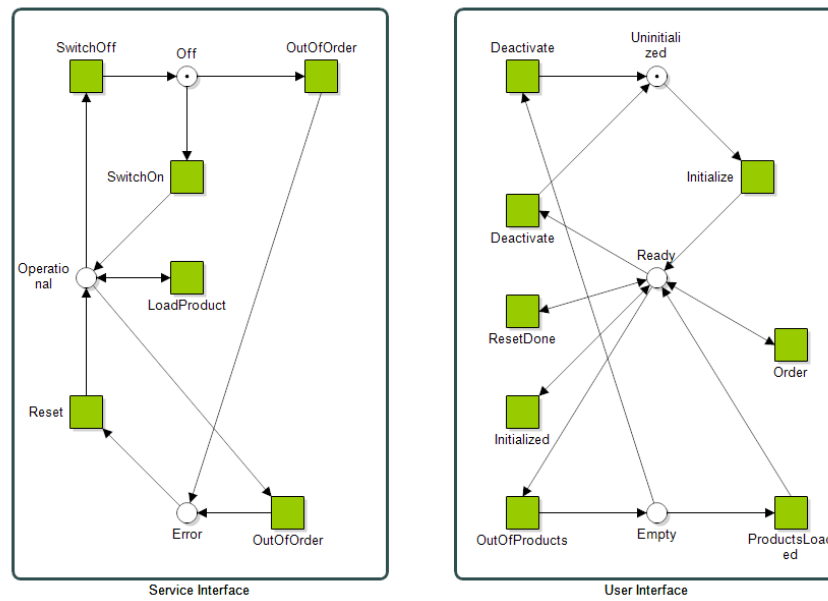


Figure 2: S-net representations of the Service Interface and User Interface of the Vending Machine component.

4.1. Enabling Constraint

An enabling constraint for a transition is defined as an additional enabling condition for that transition based on the current states of one or more other interfaces, i.e. an enabling constraint on a transition requires places of other interfaces to contain a token in order to be enabled. Currently, only conjunctive enabling constraints are supported by the encoding method. For example, the constraint that a token must be in places P and P' for T to be enabled is supported, but the constraint requiring a token must be in either P or P' for T to be enabled is not. While considering both conjunctive and disjunctive enabling constraints would enable a wider range of models to be supported, it would also add more complexity to the encoding. It is therefore something to consider in the future.

Definition 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*

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

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

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

Given constraint 1 of Figure 1c, we can then define $C_e(\text{LoadProduct}) = \{\text{Empty}\}$.

4.2. Disabling Constraint

A disabling constraint for a transition is also defined as an additional enabling condition for that transition based on the current states of one or more other interfaces. Contrary to the enabling constraint, places must now act as a disabling condition. A disabling constraint on a transition hence requires that this transition is disabled if a set of places in other interfaces contain a token.

Definition 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*

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

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

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

Given constraint 2 of Figure 1c, we can then define $C_d(\text{Order}) = \{\text{Error}\}$.

4.3. Causal Sequence Constraint

A causal sequence constraint requires that whenever a certain transition on an interface is fired then a sequence of transitions on other interfaces of that component is executed, and no other firing sequence is possible.

Definition 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:*

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

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

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

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

Given constraint 3 of Figure 1c, we can then define the causal sequence constraint $C_s(\text{SwitchOn}) = \langle \text{Initialize}, \text{Initialized} \rangle$. 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. For constraints 3 and 5 of Figure 1c, these would be SWITCHON and RESET. Any transition $t_n \in C_s(t)$ where $0 \leq n \leq |C_s(t)|$, is a *consequence transition*. These would be INITIALIZE, INITIALIZED, and RESETDONE belonging to constraint 3 and 5. We require that 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 \mathcal{O} , we refer to it as

a *free transition*. The ORDER, DEACTIVATE, and LOADPRODUCT transitions are free transitions in our example. 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} . For example, RESETDONE is a sequence successor of INITIALIZE. It is important to note that consequence transitions can only be fired after the firing of one of its predecessors, meaning they cannot be fired independently outside the context of a sequence. 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.

There are some constraint specification that could make the encoding more complex, and should be considered separately. These cases are now introduced, starting off with the notion of overlapping and diverging causal sequences. It is possible that a 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 $C_s(t) = \langle t_0, t_1, t_2, s_0, s_1, t_3 \rangle$ and $C_s(t') = \langle t'_0, t'_1, s_0, s_1, t'_2 \rangle$, this is illustrated in Figure 3. The behaviour of the sequences shown in Figure 3 is clear, at least until we reach transition s_1 . At this point, s_1 is followed by either transition t_3 or t'_2 , depending on which sequence is active. In this case, we say that the set of sequences are *diverging*, and that s_1 is a *divergence point*.

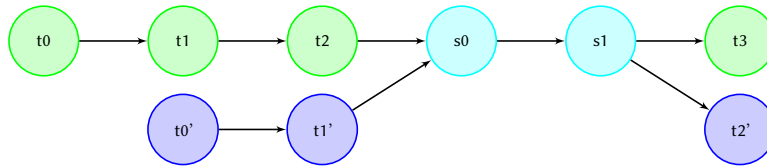


Figure 3: Overlapping and diverging sequences

When specifying causal sequence constraints, it is also possible that there are recurring elements in the sequence. While a sequence is active, it is then possible that transitions may have to be followed by different transitions in the context of this single sequence. For example, in the sequence $C_s(t) = \langle t_0, t_1, t_2, t_0, t_3 \rangle$, the first time t_0 is taken it is followed by t_1 and the second time, by t_3 . Such sequences are referred to as *multi-stage sequences*, but are out of scope in this paper. Every sequence considered in this paper is a *single-stage sequence*, as there is no divergence within the context of a single sequence. For more information about multi-stage sequences, refer to [18].

4.4. Assumptions on Functional Constraints

It is possible that adding functional constraints to a component specification causes termination problems, such as deadlock or livelock.

In the proof-sketches of Section 5, we reason about the enabledness of transitions belonging to the Petri nets produced by the encoding algorithms. If a given specification has inherent

termination problems, it could cause certain transitions to be disabled indefinitely as a result of the specified functional constraints, regardless of how they are encoded as a Petri net. Because this would create problems for the proof sketches, this section introduces three assumptions on the behaviour of a given specification.

Let $C_s(t) = \langle t_0, \dots, t_n \rangle$ where $n \in \mathcal{N}$ be causal sequence constraint on a transition t belonging to an interface N . Whenever a transition $t_i \in C_s(t)$ where $0 \leq i \leq n$ fires, the transition t_{i+1} belonging to an interface N' must be enabled in the resulting marking M . Because N' is an S-net, t_{i+1} will have exactly one place p belonging to N' in its preset. For t_{i+1} to be enabled, p must therefore have a token in M , which may not always be the case. In other words, after t_i fires, N' may not be in a state in which t_{i+1} can be fired.

Recall the causal sequence constraints 3 and 5, and enabling constraints 4 and 6 of Figure 1c. If these two enabling constraints were absent, it would be possible for SWITCHON and RESET to fire while the place UNINITIALIZED does not have a token. If these transitions were to fire in this case, the only transition that is allowed to be enabled in the resulting marking is INITIALIZE, as it is the sequence successor of both of these transitions. However, since UNINITIALIZED has no token, and is in the preset of INITIALIZE, there is no way for INITIALIZE to be enabled, resulting in a deadlock. Assumption 1 formalizes this by requiring that after a consequence or activation transition fires, the interface that its sequence successor t_{succ} belongs to is in a state where t_{succ} is enabled.

Assumption 1. *Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, \dots, t_n \rangle$ where $n \in \mathbb{N}$. For any marking m that is the result of the firing of t : t_0 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} is enabled in m' .*

Enabling and disabling constraints combined with causal sequence constraints may also lead to situations in which it is impossible to satisfy all constraints. Consider the vending machine example. Suppose that we were to define an enabling constraint requiring the Service Interface to be in the OFF state for INITIALIZE to be enabled. As INITIALIZE is a sequence successor of RESET and SWITCHON, both of which transition the Service interface to the OPERATIONAL state, firing either RESET or SWITCHON would lead to deadlock. Assumptions 2 and 3 formalize this for the enabling and disabling constraint, respectively.

Assumption 2. *Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, \dots, t_n \rangle$ 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 $t_{i+1} \in C_s(t)$ belonging to some interface N : $\forall c \in C_e(t_{i+1}) : m'(c) = 1$.*

Assumption 3. *Let t be an activation transition for a causal sequence $C_s(t) = \langle t_0, \dots, t_n \rangle$ 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 $t_{i+1} \in C_s(t)$ belonging to some interface N : $\forall c \in C_d(t_{i+1}) : m'(c) = 0$.*

5. Encoding Constraints as Petri Nets

Given the S-net representation of a set of interface skeletons, and a set of functional constraints, we will now present algorithms to add transitions and places to existing interface nets such that

they satisfy a set of given functional constraints. There are obviously a lot of different ways to this. However, for the purposes of this paper, which is gaining access to the reachability graph of a given component and verifying the absence of deadlock, livelock and unbounded behaviour, only one possible encoding has been proposed. Making a comparison between different encoding is something to consider for the future.

Figure 4 shows an example of adding functional constraints described earlier in Section 3.1 to a set of interface nets described in Figure 2. Throughout this section, different parts of Figure 4 will be highlighted and discussed to exemplify the output of the method.

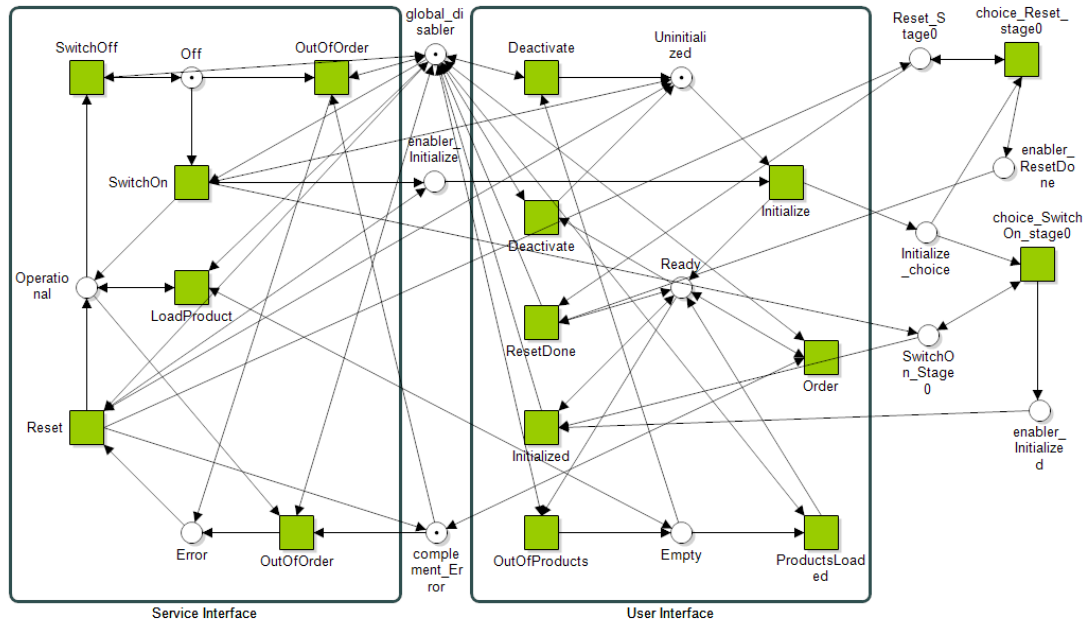


Figure 4: Resulting Petri net after applying the method in this section on the two vending machine interfaces

5.1. Enabling constraint

For a component \mathcal{O} , and a set of enabling constraints defined by C_e , Algorithm 1 generates a Petri net that 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 $C_e(t)$. The arcs are bidirectional to ensure that when t fires, the token in p is not consumed. We can see how this works out for the enabling constraints defined on the interfaces of our vending machine component. In Figure 4, we can see a bidirectional arc created between the `LOADPRODUCT` transition, and the `EMPTY` place to satisfy constraint 1 of Figure 1c. Similarly, we can see a bidirectional arc created between both the `SWITCHON` and `RESET` transition, and the `UNINITIALIZED` place to satisfy constraints 4 and 6. Next, Lemma 1 shows that Algorithm 1 encodes enabling constraints such that the reachability property in Definition 1 is satisfied.

```

1 foreach transition  $t \in T_{\mathcal{O}}$  do
2   | foreach place  $p \in C_e(t)$  do
3   |   | create bidirectional arc( $p, t$ )
4   | end
5 end

```

Algorithm 1: Enabling Constraints

Lemma 1. *The reachability property defined for the enabling constraint in Definition 1 always holds for a component \mathcal{O} after applying Algorithm 1 on \mathcal{O} .*

Proof sketch. Let \mathcal{O} be a component with the constraints C_e , and t a transition of $T_{\mathcal{O}}$. 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 1.

5.2. Disabling Constraint

The algorithm for disabling constraints is similar to that of the enabling constraint, which makes sense as the disabling constraint is essentially its inverse. When $C_d(t) = \{p\}$ for some transition t and a place p , instead of enabling t when p has a token, t must now be disabled instead. While using inhibitor arcs is an intuitive solution to this problem, we restrict ourselves to P/T nets to support reachability analysis with a broader range of tools. Instead, complement places are used, which allows us to model the functionality of inhibitor arcs. We can do this because all interfaces are S-nets and therefore safe [19]. For a place p , its complement place \bar{p} 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 \bar{p} of a place $p \in C_d(t)$. For a component \mathcal{O} , and a set of disabling constraints defined by C_d , Algorithm 2 generates a Petri net that 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 are 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. It is shown in Lemma 2 that Algorithm 2 encodes disabling constraints in a way that satisfies the reachability property in Definition 2.

Lemma 2. *The reachability property defined for the disabling constraint in Definition 2 always holds for a component \mathcal{O} after applying Algorithm 2 on \mathcal{O} .*

Proof sketch. Let \mathcal{O} be a component with the constraints C_d , and t be a transition of $T_{\mathcal{O}}$. 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 \bar{p} can only have a token if p has no token, as

```

1 foreach transition  $t \in T_{\mathcal{O}}$  do
2   foreach place  $p \in C_d(t)$  do
3     if  $m_0(p) = 1$  then
4       | create place( $\bar{p}$ , 0 tokens)
5     else
6       | create place( $\bar{p}$ , 1 token)
7     foreach place  $pt \in \bullet p$  do
8       | create arc( $\bar{p}$ ,  $pt$ )
9     end
10    foreach place  $pt \in p^\bullet$  do
11      | create arc( $pt$ ,  $\bar{p}$ )
12    end
13    create bidirectional arc( $\text{complement}_p$ ,  $t$ )
14  end
15 end

```

Algorithm 2: Disabling Constraints

by Lines 7-8, any transition in the preset of p removes a token from \bar{p} , and by Lines 10-11, any transition in the postset of p adds a token to \bar{p} . With Line 13 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 2.

We demonstrate Algorithm 2 constraint 2 of Figure 1c. Figure 4 shows that there now exists a complement place for the ERROR state, and that there is a bidirectional arc created between this complement place and the ORDER transition. The only transition that produces a token to the ERROR place, OUTOFORDER, removes a token from the complement place, and likewise the only transition that removes a token from the ERROR place, RESET, produces a token to the complement place.

5.3. Causal Sequence constraints

For a component \mathcal{O} and a causal sequence constraint $C_s(t) = \langle t_0, ..t_n \rangle$, the following two things have to be guaranteed by Algorithm 3 in order to satisfy the property given in Definition 3. Firstly, after t fires, all free transitions must be disabled until the last transition in $C_s(t)$ fires. Secondly, any 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 a token in the initial marking. 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 final transition of each sequence is then responsible for producing a token back to the disabler place. The downside of this approach is that by connecting many transitions of different interfaces to one place, we go against the locality principle of P/T nets. Structural reduction methods that can be used to reduce the size of given P/T nets, often rely on exploiting locality properties, are therefore less applicable to nets

produced by this method.

The second requirement is satisfied using distinct *enabler places* that each consequence transition will have 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 . However, if there are divergence points this is not enough, and for such cases a *decision place* is introduced. For a transition t , this decision place has a transition referred to as *decision transitions* in its postset for each possible sequence successors of t . Each of these transitions put a token in the enabler place of its corresponding sequence successor. To know which of these transitions must be enabled, an *active 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 decision transition is enabled. Introducing these additional places that indicate which sequence is active does introduce another problem. Ambiguity may now arise when when a transition is the final transition of multiple sequences, or is the final transition of one sequence while being a non-final transition of another. 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 corresponding active place.

As an example, Figure 4 shows how the causal sequence constraints 3 and 5 of Figure 1c are encoded. In our specification, INITIALIZE is a divergence point, as it is followed by either a RESETDONE or INITIALIZED. We thus need two places indicating which sequence is active, which are RESET_STAGE0 and SWITCHON_STAGE0, present on the right of Figure 4. We can see that the activation transitions SWITCHON and RESET, put a token in SWITCHON_STAGE0 and RESET_STAGE0, respectively. INITIALIZED then removes the token from SWITCHON_STAGE0, as it is the final transition of the sequence activated by SWITCHON. Likewise, RESETDONE removes the token from RESET_STAGE0, as it is the final transition of the sequence activated by RESET. As both of these transitions are final transitions, they add a token to the disabler place. Because INITIALIZE is a divergence point, we can see that it now has INITIALIZE_CHOICE in its postset, shown on the right side of Figure 4. In the postset of this place, we can see a transition corresponding to each of the two defined sequences. CHOICE_RESET_STAGE0 has a bidirectional arc to RESET_STAGE0, while CHOICE_SWITCHON_STAGE0 has a bidirectional arc to SWITCHON_STAGE0. For each of the consequence transitions INITIALIZE, INITIALIZED and RESETDONE, we can see that there is now an enabler place in their preset. Only the sequence predecessors, which for INITIALIZE are RESET and SWITCHON, put a token in these places. For the divergence point INITIALIZE, we can see that the decision transitions in the postset of INITIALIZE_CHOICE are now responsible for this, rather than INITIALIZE itself.

It now needs to be shown that after applying Algorithm 3, the resulting net satisfies any single-stage causal sequence constraint. The following places and transitions are used in the proof sketches of this section. For a consequence transition t_n , $p_n^{decision}$ refers to the decision place of t_n created on Line 16 of Algorithm 3. 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. m_{act} denotes the marking that is the result the firing of an activation transition t for a sequence $C_s(t)$.

```

1 create place(disabler, 1 token);
2 foreach sequence s do
3   | create place(sactive, 0 tokens)
4 end
5 foreach transition  $t \in T_{\mathcal{O}}$  do
6   | if t is free then
7     | create bidirectional arc(disabler, t)
8   | else if t is an activation transition for a sequence s starting with  $t_0$  then
9     | create place(t0_enabler, 0 tokens);
10    | create arc(t0_enabler,  $t_0$ );
11    | create arc(t, t0_enabler);
12    | create arc(disabler, t);
13    | create arc(t, sactive)
14  | else
15    | if t is a divergence point then
16      | create place(tdecision, 0 tokens);
17      | create arc(t, tdecision);
18      | foreach successor  $succ \in T_{\mathcal{O}}$ , where succ is a successor of t, and succ belongs
19      | to the sequence s do
20        | create transition(tsucc);
21        | create arc(tdecision, tsucc);
22        | create bidirectional arc(sactive, tsucc);
23        | if enabler place of successor(t) does not exist then
24          | create place(successor(t)_enabler, 0 tokens);
25          | create arc(successor(t)_enabler, successor(t));
26        | create arc(tsucc, successor(t)_enabler)
27      | end
28      | foreach sequence s, where t is the final transition of s do
29        | create transition(finals);
30        | create arc(tdecision, finals);
31        | create arc(sactive, finals);
32        | create arc(finals, disabler)
33      | end
34    | else
35      | if t is the last transition of the sequence s then
36        | create arc(t, disabler);
37        | create arc(sactive, t);
38      | else
39        | if if enabler place of successor(t) does not exist then
40          | create place(successor(t)_enabler, 0 tokens);
41          | create arc(successor(t)_enabler, successor(t));
42        | create arc(t, successor(t)_enabler)
43    | end
44  | end
45 end

```

Algorithm 3: Causal Sequence Constraints

Lemma 3. *The first transition of $C_s(t)$, t_0 is enabled in the marking m_{act} . For the enabledness of t_0 belonging to an interface N , we must consider the following places its preset in the marking m_{act} : 1) Exactly one place $p \in P_N$. This is because t' can only belong to one interface, N , and has only one place in its preset while belonging to N , as N is an S-net, 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 Algorithm 3 on either Line 23 or 39.*

Proof sketch. Lines 9 and 10 in Algorithm 3 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 11 then guarantees p_0^e is in the postset of the activation transition t . Using Assumption 1, we can assume that $m_{act}(p) = 1$. Using Assumption 2, we can furthermore assume that for all places $c \in C_e(t_0)$, $m_{act}(c) = 1$. Lastly, using Assumption 3, 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 4. *Let t be an activation transition for a sequence $C_s(t)$, whose firing leads to the marking m_{act} . Any consequence transition t' that is not t_0 is disabled in m_{act} .*

To show that t' belonging to an interface N is disabled in m_{act} , we only have to look at the place p'^e , which is the enabler place of t' introduced by Algorithm 3 on either Line 23 or 39.

Proof sketch. Lines 23 and 24, as well as Lines 39 and 40, guarantee that any consequence transition t' has an enabler place in its preset without a token in the initial marking. Because of Line 41, this enabler place p'^e is not in the postset of t , but rather in the postset of its predecessor transition t_{pred} if t_{pred} is not a divergence point. If t_{pred} is a divergence point, Line 25 then ensures p'^e is in the postset of a the decision transitions of t_{pred} . There are no other lines that create transitions that are in the preset of p'^e , which means means that p'^e cannot have a token in m_{act} . Therefore, t' is disabled in m_{act} .

Lemma 5. *Any free transition t_{free} or activation transition t_{act} cannot be enabled in m_{act} . To show that both t_{free} and t_{act} belonging an interface N are disabled in m_{act} , we only have to look at the disabler p_d introduced by Algorithm 3 on Line 1.*

Proof sketch. Line 7 guarantees t_{free} has the disabler place p_d in its preset, Line 12 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. Line 1 guarantees that p_d has a token in the initial marking. This guarantees only activation transitions can remove a token from p_d , while Lines 31 and 35 guarantee that the last consequence transitions of a sequence can produce a token to p_d . Because we assume that no sequence is currently active, p_d therefore has a token. Therefore, after the firing of t that results in the marking m_{act} , the disabler place p_d does not have a token in m_{act} . Therefore, t_{free} and t_{act} are disabled in m_{act} .

Lemma 6. *For any transition $t_n \in C_s(t)$, where $0 \leq n \leq |C_s(t)| - 1$, and t is an activation transition, and t_n is not a divergence point: 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} belonging to an interface N , we consider the following places in its preset in the marking m' :*

1) Exactly one place $p \in P_N$, 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' introduced by Algorithm 3 on either Line 23 or 39.

Proof sketch. Line 25 guarantees that the enabler place of t_{n+1} , p_{n+1}^e , is in the postset of t_n . Line 24 furthermore guarantees that there is an arc going from p_{n+1}^e to t_{n+1} . Because N is an S-Net, only one place belonging to N can have a token. Using Assumption 1, we can assume that this place is p , and that therefore $m'(p) = 1$. Using Assumption 2, we can furthermore assume that for all places $c \in C_e(t_{n+1})$, $m'(c) = 1$. Lastly, using Assumption 3, 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' . Furthermore, since t_n is not a divergence point and therefore has exactly one successor, it has exactly one enabler place p_{n+1}^e in its postset. Any other enabler place therefore cannot have a token while p_{n+1}^e does. Therefore, only t_{n+1} is enabled in m' .

Lemma 7. *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} belonging to an interface N , we consider the following places in its preset in the marking m : 1) Exactly one place $p \in P_N$, 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_{n+1}^e is in the postset of t' . For the place $p \in \bullet t_{n+1}$, using Assumption 1, we can assume that $m(p) = 1$. Using Assumption 2, we can furthermore assume that for all places $c \in C_e(t_{n+1})$, $m(c) = 1$. Lastly, using Assumption 3, we can assume that for all places $c' \in C_d(t_{n+1})$, $m(c') = 0$. Because p_{n+1}^e is in the postset of t' , p_{n+1}^e will also have a token in the marking m . Therefore, t_{n+1} is enabled in m .

Line 20 then ensures that $p_n^{decision}$ 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 8. *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 7 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) $p_n^{decision}$, which is the decision place of t_n introduced by Algorithm 3 on Line 16, 2) $p_{C_s(t)}^{active}$, the place indicating that the sequence $C_s(t)$ that t_n belongs to 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 preset and postset of t' . Knowing 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 decision transitions for non-final transitions do not consume a token from $p_{C_s(t)}^{active}$, therefore is a token in the place $p_{C_s(t)}^{active}$. Therefore t' is enabled in m . Line 21 also ensures that any other decision transition t'' with $p_n^{decision}$ 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, any transition t'' cannot be enabled in m . Therefore, only t' is enabled in m .

Theorem 1. *A component always satisfies a set of diverging, 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 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. Following Lemma 5, any transition not belonging to a sequence is disabled after t fires. Furthermore, any transition belonging to a sequence that is not t_0 is disabled following Lemma 4. Because t_0 is guaranteed to be enabled following Lemma 3, the only possible transition that be enabled after t fires is t_0 . Following Lemmas 6, 7 and 8, the n th sequence transition to fire after t fires, must be the n th element of $C_s(t)$, as any transition not belonging to a sequence is disabled, and any transition t_n belonging to 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. Conclusions

This paper addressed the challenge of correctly and cost-effectively designing complex component-based systems, built from reusable components that provide and consume services to and from each other over a set of interfaces. The state of an interface on a component may be dependent on the state of another, which may result in deadlock, livelock, or unbounded behavior. To address this challenge, the paper proposed a way to capture partial specifications of component behavior as sets of commonly occurring functional constraints. We formalized such functional constraints and presented algorithms to generate Petri nets satisfying the constraints, supported by proof sketches to show their correctness. The approach was demonstrated through a running example of a vending machine.

Future work involves adding support for the complete ComMA language, as the current solution does not cover required interfaces and compound transitions. Other possible directions include adding support for more types of functional constraints, extensions with data, and take into account commonly occurring middleware patterns, such as blocking calls. Furthermore, a comparison could be made between multiple encoding methods. For example, a compositional approach using label based synchronization could be considered as an alternative approach, making for a more modular solution that is easier to extend.

References

- [1] HTSM Systems Engineering Roadmap, 2020. URL: https://hollandhightech.nl/_asset/_public/Innovatie/Technologieen/z_pdf_roadmaps/Roadmap-Systems-Engineering-update-2020-final-v20200724.pdf.

- [2] C. Y. Baldwin, 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* 49 (2002) 19–37.
- [4] I. Kurtev, M. Schuts, J. Hooman, D.-J. Swagerman, Integrating interface modeling and analysis in an industrial setting, 2017, pp. 345–352.
- [5] R. van Beusekom, B. de Jonge, P. Hoogendijk, J. Nieuwenhuizen, Dezyne: Paving the way to practical formal software engineering, *Electronic Proceedings in Theoretical Computer Science* 338 (2021) 19–30.
- [6] S. Bensalem, M. Bozga, J. Sifakis, T.-H. Nguyen, Compositional verification for component-based systems and application, in: S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, M. Viswanathan (Eds.), *Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 64–79.
- [7] B. Metzler, H. Wehrheim, D. Wonisch, Decomposition for compositional verification, in: S. Liu, T. Maibaum, K. Araki (Eds.), *Formal Methods and Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 105–125.
- [8] R. van Beusekom, J. Groote, P. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, T. Willemse, Formalising the dezyne modelling language in mcrl2, in: *Critical Systems: Formal Methods and Automated Verification*, Springer, Germany, 2017, pp. 217–233.
- [9] G. Engels, J. M. Küster, R. Heckel, M. Lohmann, Model-based verification and validation of properties, *Electronic Notes in Theoretical Computer Science* 82 (2003) 133–150. UNIGRA’03, Uniform Approaches to Graphical Process Specification Techniques.
- [10] B. Selic, Using uml for modeling complex real-time systems, in: F. Mueller, A. Bestavros (Eds.), *Languages, Compilers, and Tools for Embedded Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 250–260.
- [11] C. A. R. Hoare, Communicating sequential processes, *Commun. ACM* 21 (1978) 666–677.
- [12] D. Bera, K. Hee, van, M. Osch, van, J. Werf, van der, A component framework where port compatibility implies weak termination, *Computer science reports*, Technische Universiteit Eindhoven, 2011.
- [13] D. Craig, W. Zuberek, Compatibility of software components - modeling and verification, in: *2006 International Conference on Dependability of Computer Systems*, 2006, pp. 11–18.
- [14] D. Bera, K. M. van Hee, J. M. van der Werf, Designing weakly terminating ros systems, in: S. Haddad, L. Pomello (Eds.), *Application and Theory of Petri Nets*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 328–347.
- [15] I. Kurtev, J. Hooman, M. Schuts, *Runtime Monitoring Based on Interface Specifications*, Springer International Publishing, Cham, 2017, pp. 335–356.
- [16] J. L. Peterson, Petri nets, *ACM Comput. Surv.* 9 (1977) 223–252.
- [17] J. Baez, J. Master, Open petri nets, *Mathematical Structures in Computer Science* 30 (2020) 1–28.
- [18] B.-J. Hilbrands, *Verification of Inter-Dependent Interfaces in Component-Based Architectures*, Master’s thesis, University of Amsterdam, Amsterdam, 2021. URL: <https://www.akesson.nl/files/students/hilbrands-thesis.pdf>.
- [19] W. M. P. van der Aalst, C. Stahl, M. Westergaard, Strategies for modeling complex processes using colored petri nets, in: K. Jensen, W. M. P. van der Aalst, G. Balbo, M. Koutny, K. Wolf

(Eds.), *Transactions on Petri Nets and Other Models of Concurrency VII*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 6–55.