On Abstraction-Based Deadlock Analysis of Service-Oriented Systems Modeled with Recursive Petri Nets

Erik Jonas Hartnick[1,*,†], Mandy Weißbach[1,*,†] and Thomas Kühn[1,*,†]

[1]Institute of Computer Science, Martin-Luther-University Halle-Wittenberg, Von-Seckendorff-Platz 1, 06120 Halle (Saale), Germany

Abstract

Previous research has highlighted limitations in Petri net-based approaches for deadlock detection of service-oriented systems, particularly because the recursive nature of synchronous and asynchronous service calls is not preserved in the abstraction, leading to a failure to detect certain deadlocks. To address this issue, prior work proposed a recursive Petri net abstraction. This paper presents a case study applying such an abstraction to a specific service-oriented system. A reachability-based breadth-first search procedure is employed to detect deadlocks in the resulting model. The results demonstrate the effectiveness of the approach by successfully detecting a known deadlock that is not captured by classical Petri net–based methods.

1. Introduction

Service-oriented architectures are increasingly employed in modern software systems across a wide range of domains, including medical systems [1], smart home and Internet of Things (IoT) environments [2, 3, 4], and robotics frameworks, e.g., ROS [5]. In safety-critical domains, such as medical systems and assistive smart home environments, it is essential that these systems operate according to their specifications and without critical faults, as human well-being may depend on their correct behavior.

A particular challenge in ensuring system correctness is the detection of deadlocks. In service-oriented systems, the conditions leading to deadlocks are often obscured by the inherent complexity, modularity, loose coupling, and distribution of services over several devices, sensors, or actors. This issue is particularly evident in multi-robot systems, where two or more robots may block each other while attempting to accomplish their tasks. Consequently, deadlock avoidance strategies tailored to robotics are studied extensively [6, 7]. However, deadlocks may also arise at the level of the underlying service-oriented systems. Since services may be provided by third-party vendors and source code may remain inaccessible, models must capture their behavior using interaction constraints alone. Interactions between services can be described by synchronous and asynchronous procedure calls. Since procedure calls can be recursive, the model must capture unbound concurrency as well as unbound recursion [8]. An abstraction-based approach with Petri nets has been considered by Weißbach and Zimmermann [8], showing that due to the lack of recursion in Petri nets, deadlocks may remain undetected. Another approach based on (P,G)-process rewrite systems (PANs), as introduced by Mayr [9], was able to detect deadlocks correctly [10]. Although an abstraction for service-oriented systems to recursive Petri nets (RPNs) has been proposed in [11], neither the composition of services nor the application of deadlock analysis in the resulting model has been demonstrated.

We address this gap by describing the missing composition of RPN service abstractions and a breadth-first search algorithm for the deadlock analysis of recursive Petri nets. We apply our approach to a representative example from [10], for which classic Petri net-based deadlock analysis would fail.

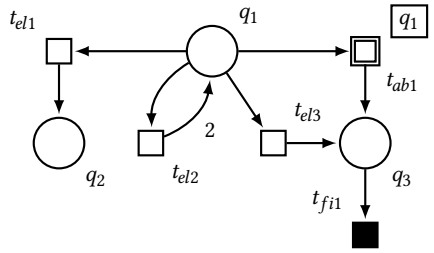
PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

*Corresponding author.

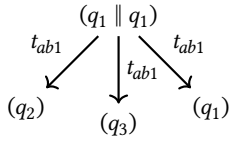
†These authors contributed equally.

erik.hartnick@student.uni-halle.de (E. J. Hartnick); mandy.weissbach@informatik.uni-halle.de (M. Weißbach); thomas.kuehn@informatik.uni-halle.de (T. Kühn)

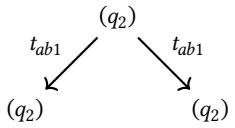
0009-0003-2377-7349 (E. J. Hartnick); 0009-0007-7458-3658 (M. Weißbach); 0000-0001-7312-2891 (T. Kühn)



(a) Graphical represenation of a small recursive Petri net N.



(b) Notation for an extended marking tr_1 for N.



(c) An extended marking tr_2 for N in a deadlock.

Figure 1: Graphical representation for a small recursive Petri net N (a) and two extended markings tr_1 (b) and tr_2 (c) of N, whereas the latter shows a deadlock in N.

In Sect. 2, we will formally introduce RPNs and the corresponding notion of deadlocks as well as service-oriented systems. Afterward in Sect. 3, we present our approach to model and compose service-oriented systems as RPNs as well as our deadlock analysis procedure. The presented deadlock analysis is then applied to an example of a service-oriented system with a deadlock in Sect. 4. We discuss the results and limitations of the presented approach in Sect. 5 and related approaches in Sect. 6. Sect. 7 concludes this paper.

2. Preliminaries

2.1. Recursive Petri Nets

Several definitions of recursive Petri nets (RPNs) have been proposed, e.g., [12, 13, 14, 15, 16]. While most definitions employ final markings and an associated *cut*-step [14, 15], Haddad and Poitrenaud [13] only require transitions as a means to model the recursive invocation of the Petri net. As this simplifies the formalization of their semantics, we follow their definition of recursive Petri nets [13]. For brevity, we will disregard other definitions of RPNs. Here, $\uplus$ denotes the disjoint union of two sets, and R^* denotes the reflexive, transitive closure of a homogeneous relation R.

Definition 1 (Recursive Petri Net [13])**.** A *recursive Petri net* is a tuple $N \triangleq (P, T, W^-, W^+, \Omega)$ where

(i) P is a finite set of places;
(ii) T is a finite set of transitions;
(iii) Every $t \in T$ is either elementary $t \in T_{el}$, abstract $t \in T_{ab}$, or final $t \in T_{fi}$ with $T = T_{el} \uplus T_{ab} \uplus T_{fi}$;
(iv) $W^+ : P \times T \to \mathbb{N}$ and $W^- : P \times T \to \mathbb{N}$ are the pre- and post-flow functions; and
(v) $\Omega : T_{ab} \to \mathbb{N}^{|P|}$ is a labeling function mapping each abstract transition to a Petri net marking.

The main addition of RPNs is the distinction between elementary, abstract, and final transitions. While the former are regular Petri net transitions, abstract transitions model the recursive invocation of the Petri net with the (ordinary) initial marking given in Ω. The final transition denotes the end of a recursive invocation comparable to a return statement. Figure 1a shows an RPN, where t_{ab_1} is an abstract transition and t_{fi_1} is a final transition. The box with q_1 next to the abstract transition denotes the initial marking $\Omega(t_{ab1}) = (1, 0, 0)^T$. Notably, though, the ordinary notion of a Petri net marking (i.e., $\mu \in \mathbb{N}^{|P|}$) cannot capture the recursion introduced by firing abstract transitions. Thus, Haddad and Poitrenaud [13] extended this notion as follows:

Definition 2 (Extended marking of an RPN [13])**.** An *extended marking tr* of a recursive Petri net $N \triangleq (P, T, W^-, W^+, \Omega)$ is a labeled directed tree $tr \triangleq (V, M, E, A)$ where

(i) V is the set of vertices of tr;
(ii) $M : V \to \mathbb{N}^{|P|}$ maps each vertex to an ordinary marking of this Petri net;
(iii) $E \subseteq V \times V$ is the set of edges of tr forming a tree with exactly one root vertex $root \in V$;
(iv) $A : E \to T_{ab}$ associates each edge of tr with the invoking abstract transition of this Petri net.

For a vertex $v \in V$, $pred(v)$ denotes the predecessor u with $(u, v) \in E$. For the root vertex, $pred(root)$ is undefined. Similarly, $Succ(v) \triangleq \{w \mid (v, w) \in E^*\}$ collects all direct and indirect successors of v.

Initially, an extended marking is an ordinary Petri net marking $M(root)$ denoting the state of a single process. Once this process recursively invokes itself, the subprocess is again represented with an ordinary marking $M(v')$ that is linked to the parent process $(root, v') \in E$. Consequently, the extended marking captures the state of the root process and all subprocesses as a tree of Petri net markings. For brevity, we employ a shorthand for Petri net markings inspired by the algebraic expressions of processes by Mayr [9]. In detail, a marking is represented as a parallel composition $\parallel$ of places p,[1] such that the number of occurrences of p corresponds to the number of tokens in place p. For instance, in Figure 1, the extended marking tr_1 represents the marking $\mu = (2, 0, 0)^T$ with the expression $q_1 \parallel q_1$, which invoked three subprocesses. As each process can progress independently, i.e., each process is asynchronous wrt. $(P, P) - PRS$ [9], the notion of enabled transition and firing semantics are augmented as follows:

Definition 3 (Enabled transition in an RPN [13])**.** Let $N \triangleq (P, T, W^-, W^+, \Omega)$ be recursive Petri net with an extended marking $tr \triangleq (V, M, E, A)$. A transition $t_j \in T$ is *enabled in* $M(v)$ (of vertex $v \in V$) iff for all $p_i \in P : M(v)(p_i) \geq W^-((p_i, t_j))$.

Definition 4 (Firing semantics of an RPN [13])**.** Let $N \triangleq (P, T, W^-, W^+, \Omega)$ be recursive Petri net with an extended marking $tr \triangleq (V, M, E, A)$ and t be an enabled transition in the Petri net marking $M(v)$ of vertex $v \in V$. *Firing* transition t derives a new extended marking tr' from tr, denoted $tr \xrightarrow{t} tr'$, taking the kind of transition into account:

(i) Let $t \in T_{el}$ be an elementary transition. Then t fires in $M(v)$ according to the Petri net semantics, maintaining the tree structure. The resulting extended marking $tr' \triangleq (V, M', E, A)$ is defined with:
- $\forall w \in V \setminus \{v\} : M'(w) \triangleq M(w)$
- $\forall p \in P : M'(v)(p) \triangleq M(v)(p) - W^-((p, t)) + W^+((p, t))$

(ii) Let $t \in T_{ab}$ be an abstract transition. Then t adds a new vertex $v' \notin V$ as a child of v, marking $M(v')$ with the initial marking $\Omega(t)$. A new edge (v, v') is added with the label t. The tokens consumed by t are *held* until a final transition fires in v'. The resulting extended marking $tr' \triangleq (V', M', E', A')$ is defined as:
- $V' \triangleq V \cup \{v'\}$
- $\forall w \in V \setminus \{v\} : M'(w) \triangleq M(w)$
- $\forall p \in P : M'(v)(p) \triangleq M(v)(p) - W^-((p, t))$
- $M'(v') \triangleq \Omega(t)$
- $E' \triangleq E \cup \{(v, v')\}$
- $\forall e \in E : A'(e) \triangleq A(e)$
- $A'((v, v')) \triangleq t$

(iii) Let $t \in T_{fi}$ be a final transition. Then t removes the subprocesses rooted in v from the extended marking. For the *root* vertex in tr, this results in the empty extended marking $tr' \triangleq \perp$. Otherwise, the corresponding abstract transition $A((pred(v), v)) = t_{ab}$, now *releases* the *held* tokens into $M(pred(v))$. The resulting extended marking $tr' \triangleq (V', M', E', A')$ is defined as:
- $V' \triangleq V \setminus Succ(v)$
- $\forall w \in V' \setminus \{pred(v)\} : M'(w) \triangleq M(w)$
- $\forall p \in P : M'(pred(v))(p) \triangleq M(pred(v))(p) + W^+((p, A((pred(v), v))))$
- $E' \triangleq E \cap (V' \times V')$
- $\forall e \in E' : A'(e) \triangleq A(e)$

The former definition ensures that a transition is always enabled local to a specific process (vertex v) in the extended marking. The latter definition distinguishes the semantics of firing a transition depending on its kind. While firing an elementary transition only changes the Petri net marking $M(v)$

[1] The $\parallel$ operator is commutative and associative.

of the corresponding process v in the new extended marking, firing an abstract transition changes the Petri net marking of the corresponding process and spawns a new subprocess, whereas $\Omega(t)$ determines the initial Petri net marking of the subprocess. In contrast, firing a final transition in a specific process v removes all subprocesses $w \in Succ(v)$ from the new extended marking and changes the Petri net marking of the parent process $pred(v)$ wrt. outgoing arcs of the corresponding abstract transition (given by the mapping $A(\,(pred(v), v)\,)$).

Taking inspiration from the definition of the language of recursive Petri nets in [13], we denote a marked recursive Petri net as an RPN with an initial state and several final states, as follows:

Definition 5 (Marked RPN)**.** A *marked recursive Petri net* is a quadruple (N, tr_0, Tr_f, tr), where

(i) $N \triangleq (P, T, W^-, W^+, \Omega)$ is a recursive Petri net;
(ii) tr_0 is an initial extended marking of N;
(iii) Tr_f is a set of final extended markings of N; and
(iv) $tr \triangleq (V, M, E, A)$ is the current extended marking of N reachable from tr_0 by $tr_0 \xrightarrow{\sigma} tr, \sigma = t_1 t_2 \cdots t_n$.

Following [13], a marked RPN has exactly one initial extended marking tr_0 from which its execution starts. The marked RPN is executed by firing one enabled transition t in the extended marking tr, creating a new marked RPN (N, tr_0, Tr_f, tr') with $tr \xrightarrow{t} tr'$. However, execution halts if one of the final extended markings is reached $tr' \in Tr_f$, regardless of any enabled transitions. Please note that an initial marking refers to $\Omega(t_{ab})$ for an abstract transition $t_{ab} \in T_{ab}$, whereas an initial *extended* marking refers to tr_0 of a marked RPN. The notion of a marked RPN together with the firing semantics of RPN gives rise to a reachability graph where extended markings form the graph's nodes, defined as follows:

Definition 6 (Reachability graph of an RPN)**.** Let (N, tr_0, Tr_f, tr) be a marked recursive Petri net. Then the set of extended markings Tr reachable from tr_0 in RPN N is inductively defined as:

1. $tr_0 \in Tr$
2. for any $tr_i \in Tr \setminus Tr_f$ and transition t enabled in tr_i from $tr_i \xrightarrow{t} tr_j$ follows $tr_j \in Tr$.

Conversely, the induced *reachability graph* is the directed graph $G \triangleq (Tr, D, trans, vert)$ where:

(i) Tr is the set of reachable extended markings;
(ii) $D \subseteq Tr \times Tr$ is the set of edges, such that $(tr_i, tr_j) \in D$ iff $tr_i \xrightarrow{t} tr_j$, i.e., if there exists an enabled transition $t \in T$ that is fired in vertex $v \in V$ of $tr_i \notin Tr_f$ creating the extended marking tr_j;
(iii) $trans : D \to T$ is a labeling function assigning the corresponding transition t to each edge $(tr_i, tr_j) \in D$; and
(iv) $vert : D \to V_{tr_i}$ is a labeling function annotating each edge $(tr_i, tr_j) \in D$ with the vertex v of tr_i in which t was fired.

Similar to the reachability graph of a Petri net, this definition represents the states of an RPN and state transitions by means of extended markings $tr_i \in Tr$ and, respectively, edges $(tr_i, tr_j) \in D$ between extended markings labeled with the fired transition and corresponding vertex (or process). However, in contrast to the typical notion of reachability, our reachability graph is truncated whenever a final extended marking is reached, i.e., no further reachable extended markings are considered. Conversely, we only consider extended markings of a marked RPN as a deadlock if the extended marking is not final. Still, the original definition of a deadlock in Petri nets does not directly apply.

2.2. Deadlocks in Recursive Petri Nets

To analyze deadlocks in recursive Petri nets, we require a definition of deadlocks in recursive Petri nets. However, as the semantics of RPN (cf. 4) permit asynchronous execution of each process, i.e., firing an enabled transition in vertex v of an extended marking tr, each process v (and subprocess) can be in a local deadlock. Following the naming scheme in [13],[2] we denote such a local deadlock as a *threadlock* and define it as follows:

[2]Haddad and Poitrenaud [13] refer to the marking $M(v)$ of a vertex v in an extended marking as a thread.

Definition 7 (Threadlock in an RPN)**.** Let (N, tr_0, Tr_f, tr) be a marked recursive Petri net with the current extended marking $tr \triangleq (V, M, E, A)$, whereas $tr \notin Tr_f$. A Petri net marking $M(v)$ of vertex $v \in V$ in the extended marking tr is in a *threadlock* iff no transitions $t \in T$ are enabled in $M(v)$, i.e., $\forall t \in T : \exists p \in P : M(v)(p) < W^-((p,t))$.

While a threadlock also constitutes a deadlock in the underlying Petri net $\overline{N} \triangleq (P, T, W^-, W^+)$ marked by $M(v)$, the execution of the marked RPN can still proceed, as other markings $M(w)$ with $w \in V$ and $w \neq v$ might still have enabled transitions. In particular, firing a final transition t_f in vertex w might remove vertex v, where $M(v)$ is in a threadlock from the extended marking, if $v \in Succ(w)$. As an example, the extended marking tr_1 in Figure 1b has a threadlock in the vertex labeled (q_2). Consequently, a deadlock in an RPN can only occur if all processes in an extended marking are in a threadlock.

Definition 8 (Deadlock in an RPN)**.** Let (N, tr_0, Tr_f, tr) be a marked recursive Petri net with the current extended marking $tr \triangleq (V, M, E, A)$ and $tr \notin Tr_f$. The marked recursive Petri net is in a *deadlock* for tr iff for all $v \in V$, $M(v)$ is in a threadlock.

In general, this defintion captures the intended meaning of a deadlock for extended markings in a marked RPN, i.e., that there are no enabled transitions in any marking of any vertex. The extended marking tr_2 shown in Figure 1c illustrates an extended marking in a deadlock. Here, all vertices of the extended marking tr_2 have the same marking (q_2) $(\mu = (0,1,0)^T)$ for which no transition is enabled. As a result, the RPN of Figure 1a is in a deadlock for the extended marking tr_2 (cf. Figure 1c).

2.3. Service-Oriented Systems

A *service-oriented system* denotes a system built using the service-oriented architecture style. According to [17, 18], such a system "*must provide visibility of needs and capabilities [by means of services]; must include means for [service] consumers and [service] providers to interact; and must produce real-world effects that address a [client's] business problem*" [17, p. 103]. Although this definition is very general, for the analysis of service-oriented systems, we adhere to the more formal definition proposed in [10].

Definition 9 (Service-oriented system [10])**.** A service-oriented system *sos* is definied as the tuple $sos \triangleq (\mathcal{S}, \mathcal{R}, \mathcal{I}, \mathcal{B})$, with $n \in \mathbb{N}$ where

(i) $\mathcal{S} \triangleq \{S_1, \dots, S_n\}$ is a finite set of services $S_1, \dots, S_n$;
(ii) $\mathcal{R} \triangleq \{R_{S_1}, \dots, R_{S_n}\}$ is the finite set of required interfaces associated with each service $S_1, \dots, S_n$, where R_{S_i} denotes a set of procedure signatures that S_i requires, $i \in \{1, \dots n\}$;
(iii) $\mathcal{I} \triangleq \{I_{S_1}, \dots, I_{S_n}\}$ is the finite set of provided interfaces associated with each service $S_1, \dots, S_n$, where I_{S_i} denotes a set of the procedure signatures that S_i provides, $i \in \{1, \dots n\}$; and
(iv) $\mathcal{B} \subseteq \mathcal{R} \times \mathcal{I}$ is the set of bindings where for every binding $(S_i, S_j) \in \mathcal{B}$ exists at least one common procedure $p \in R_{S_i} \cap I_{S_j}$ (i.e., $R_{S_i} \cap I_{S_j} \neq \emptyset$).

Without loss of generality, in this definition we assume that each service has one provided and one required interface. However, as an interface can be empty, we can model services without a required/provided interface. Likewise, if a service has multiple required/provided interfaces, their procedures can be collected in a single required/provided interface. This is possible because, in this definition, a binding between S_i and S_j can be established if there is at least one procedure p in the required interface R_{S_i} that is included in the provided interface I_{S_j}. As a result, a required interface can be bound to multiple provided interfaces of different services, each binding different methods. Overall, the binding relation $\mathcal{B}$ specifies the composition of the services of the service-oriented system. Based on these foundations, we can now describe the approach for our analysis in the next section.

3. Deadlock Analysis of Service-Oriented Systems

Deadlock analyses can be performed both statically, prior to the execution of a system, and dynamically, recognizing the deadlock when it happens. Our approach is an abstraction-based static analysis on

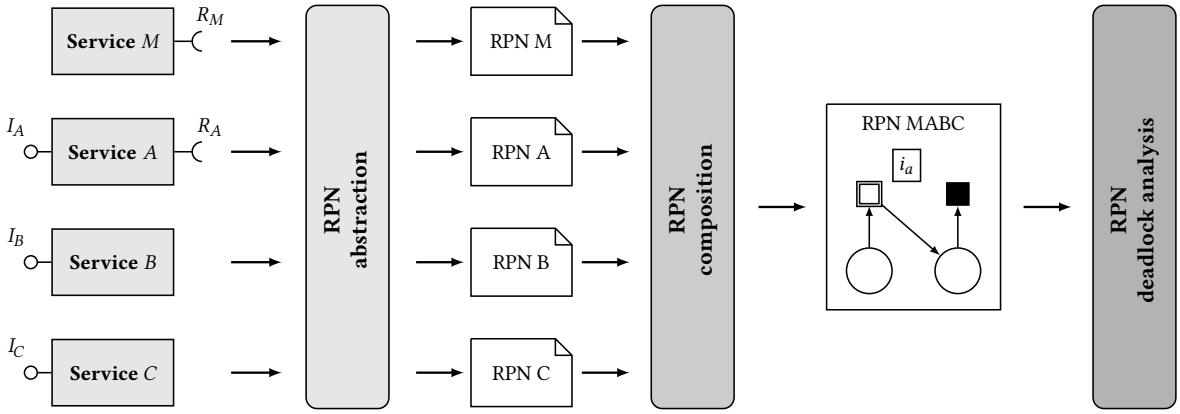


Figure 2: Process of the abstraction and deadlock analysis

the control flow of the system represented in recursive Petri nets, recognizing deadlocks before the system is executed in alignment with safety requirements for service-oriented systems. In refinement-based approaches, models are created during the initial development to obtain an implementation, but during later maintenance and enhancements, those models are not likely to be updated. As such, abstraction-based approaches that only rely on the current, actual implementation are better suited for service-oriented systems. The knowledge of implementations can be reduced to the interfaces and their constraints and conditions, in which services may be called through their interfaces.

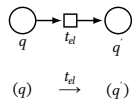
Regardless of how the model is obtained, the deadlock analysis is then performed in the model and has to be validated in the actual system. Thus, we structure the analysis into three steps, as described by Figure 2. The implementation of services and their interfaces are taken as inputs, represented by the UML components on the left. In the first step, each service is abstracted separately to an incomplete recursive Petri net fragment to account for the composability of services in the model, represented by the document boxes and described in Sect. 3.1. The second step is the composition of the fragments into a single recursive Petri net, symbolized by the box with the N-shaped recursive Petri net, explained in Sect. 3.2. This is the complete model on which the deadlock analysis is performed in the third step, for which a procedure is described in Sect. 3.3.

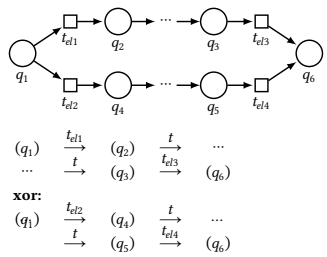
3.1. Abstraction to Recursive Petri Net Services

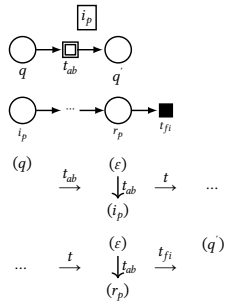
The abstraction assumes that the implementation of interfaces as well as the constraints on when to call them can be described in a common service-oriented language, such as the example language $\mathcal{XYZ}$ by Weißbach [10]. Since only the interactions of conditions and procedure calls across services are relevant to the control flow-based deadlock analysis [19, 8], the abstraction $\alpha_{RPN} : S \to N_S$ as described in Table 1a and Table 1b for assignments, if-then-else statements, synchronous procedure calls, asynchronous procedure calls, and synchronization is sufficient. In that case, S is the service, and $N_S \triangleq (P_S \uplus DUMMY_S, T_S, W_S^-, W_S^+, \Omega_S)$ is the recursive Petri net fragment. In $\mathcal{XYZ}$, procedures are declared asynchronous with the keyword **async**. An asynchronous call is assigned to a variable of data type future, providing a reference to the asynchronous call for explicit synchronization in a **sync** statement. The caller of synchronous procedures must wait until the callee has returned, using the classical procedure semantics of a stack. Both the caller and the callee in asynchronous procedures may be executed concurrently from its call until a synchronization point is reached. Synchronization happens when the caller reaches either a sync statement or a return statement. In both cases, the caller will then wait for the callee to return [8]. Since a repeated call on the same future has the semantics of replacing the function object previously bound, loops like while can be reduced to their last iteration and replaced by the conditional if-then-else statement with the same semantics.

All procedure calls in Table 1a and Table 1b are modeled by abstract transitions. While the abstraction is described to have initializations with the first program point of the called procedure i_p, we will

Table 1
Abstraction of instructions from $\mathcal{XYZ}$ to recursive Petri nets.

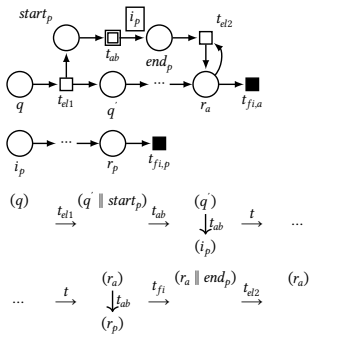
Control flow	Representation as RPN
q : $x := e$; q' : ...	

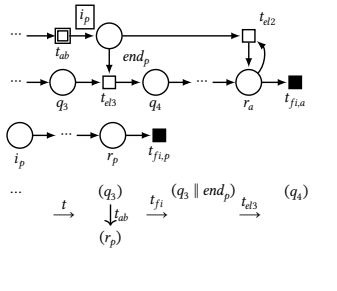
q_1 : **if** e {
q_2 : ...
...
q_3 : } **else** {
q_4 : ...
...
q_5 : }
q_6 : ...
	

Synchronous
Procedure p:
q : $p()$;
q' : ...
$p()$ {
i_p : ...
r_p : **return**;
}
	

(a) Abstraction of assignment, if-else and synchronous procedure call control flows to recursive Petri nets [11].

Control flow	Representation as RPN

Asynchronous
Procedure p:
$a()$ {
future f;
q : $f := p()$;
q' : ...
r_a : **return**;
}
async $p()$ {
i_p : ...
r_p : **return**;
}
	

Synchronisation:
$a()$ {
future f;
q_1 : $f := p()$;
q_2 : ...
q_3 : **sync** f;
q_4 : ...
r_a : **return**;
}
async $p()$ {
i_p : ...
r_p : **return**;
}
	

(b) Abstraction of asynchronous procedure call control flows to recursive Petri nets [11].

abstract this to a dummy place $init_p \in DUMMY_S$, with $\mathscr{D}_{(S,S')}(init_p) = i_p$ instead, acting as a placeholder until the composition is applied and the procedure p is bound by a binding $(S, S') \in \mathscr{B}$. To ensure the concurrency and synchronization of asynchronous procedures, the places $start_p$ and end_p are introduced. This turns asynchronous procedure calls and their return and synchronization into a two-step process [11]. These recursive Petri net fragments representing the services now require composition, described in the next subsection.

3.2. Composition to a Recursive Petri Net

To compose the model, we have abstracted services S_i to their recursive Petri net fragments N_{S_i} in the previous subsection. The composition of the fragments is now defined as the union of all RPN fragments, replacing the initialization dummy places with the corresponding first places of each procedure described by the binding of the underlying service-oriented system:

Definition 10 (Composition of RPN service fragments)**.** Let $sos \triangleq (\mathcal{S}, \mathscr{R}, \mathscr{I}, \mathscr{B})$ be a service-oriented system. For each service $S \in \mathcal{S}$, let $N_S \triangleq (P_S \uplus DUMMY_S, T_S, W_S^-, W_S^+, \Omega_S)$ be the RPN fragment obtained by the abstraction α_{RPN} as $\alpha_{RPN}(S) = N_S$. The composition of all RPN fragments N_S is the RPN $N_{sos} \triangleq (P_{sos}, T_{sos}, W_{sos}^-, W_{sos}^+, \Omega)$ where:

(i) $P_{sos} \triangleq \biguplus_S P_S$ is the union of all sets of places that are not placeholders;
(ii) $T_{sos} \triangleq \biguplus_S T_S$ is the union of all sets of transitions;
(iii) $W_{sos}^- \triangleq P_{sos} \times T_{sos} \to \mathbb{N}, (p,t) \mapsto n$ iff $W_S^-((p,t)) = n$ for all S;
(iv) $W_{sos}^+ \triangleq P_{sos} \times T_{sos} \to \mathbb{N}, (p,t) \mapsto n$ iff $W_S^+((p,t)) = n$ for all S; and
(v) $\Omega_{sos} : T_{ab,sos} \to \mathbb{N}^{|P_{sos}|}, t_{ab} \mapsto (i_p)$ for all bindings $(S, S') \in \mathscr{B}$ and all common procedures $p \in R_S \cap I_{S'}$ whereas $\mathscr{D}_{(S,S')}(init_p) = i_p$.

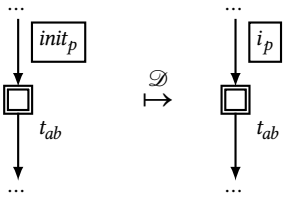
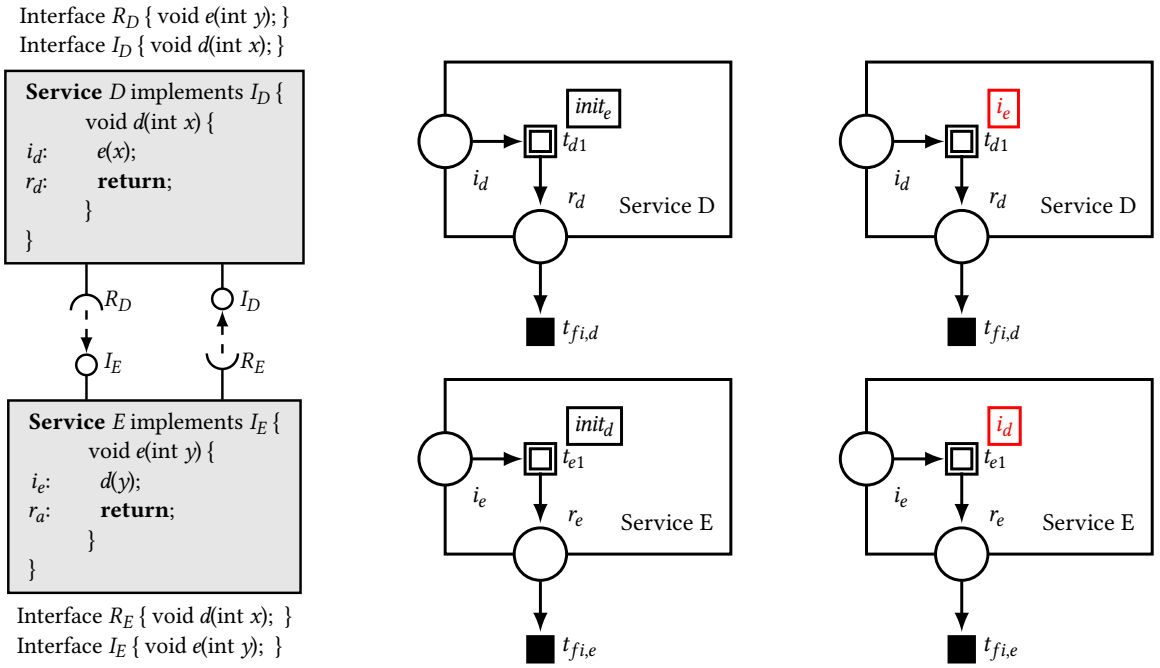


Figure 3: Replacement of dummy places in the initial markings of abstract transitions



(a) Services D and E, with required interfaces R_D and R_E, provided interfaces I_D and I_E.

(b) Abstraction of service D to RPN N_D and of service E to RPN N_E.

(c) Composition of RPN N_D and RPN N_E to one RPN N_{sos}, $\mathscr{D}(init_d) = i_d$, $\mathscr{D}(init_e) = i_e$,.

Figure 4: Example for the abstraction and composition of a service-oriented System with Services D and E

Note, the assigned marking (i_p) represents the first program point in the RPN fragment $N_{S'} = (P_{S'}, T_{S'}, W^-_{S'}, W^+_{S'}, \Omega_{S'})$ of service S', i.e. a marking where only i_p has a token. Moreover, i_p is obtained from $\mathscr{D}_{(S,S')}(init_p)$. Here, $T_{ab,sos} \subset T_{sos}$ denotes the set of all abstract transitions in the *sos*.

The application replacement of the dummies in the initial marking of a procedure $p \in R_S \cap I_{S'}$ for a binding $(S, S') \in \mathscr{B}$ by $\mathscr{D}_{(S,S')}(init_p) \triangleq i_p$ is shown in Figure 3. The placeholder for the initial marking $\Omega_S(t_{ab}) = init_p$ in the box top right of the abstract transition t_{ab} on the left is replaced by the actual marking with place i_p in the box top right of the abstract transition on the right.

In the case of a small service-oriented system with two services, D and E, Figure 4 illustrates the abstraction of services (Figure 4b) and their composition (Figure 4c). Both services and their interfaces are implemented in the formal $\mathscr{XYZ}$ language [10], depicted in the component diagram in Figure 4a. Service D has a required interface R_D with the synchronous procedure e and provides the synchronous procedure d through the provided interface I_D. In its implementation, d calls the required procedure e as its only statement before returning. Likewise, service E has a required interface R_E with the synchronous procedure d and provides the synchronous procedure e through I_E. In turn, its implementation of e only calls the required procedure d. Because procedure d is bound by $(E, D) \in \mathscr{B}_{DE}$ and procedure e by $(D, E) \in \mathscr{B}_{DE}$, they will call one another in an unbound recursion.

The services D and E are abstracted to RPN fragments N_D and N_E, shown in Figure 4b. The program points i_d and r_d are represented by places in N_D. The abstract transition t_{d1} representing the procedure

Procedure 1: BFS deadlock analysis

Input : (N, tr_0, Tr_f, tr_i) a marked recursive Petri net with $N \triangleq (P, T, W^-, W^+, \Omega)$
Output : *true* if a deadlock was found in N, $false$ otherwise

```
1  begin
2      q := new Queue<Tr, T, V>();
3      q.enqueueAll({(tr_0, t, v) : t enabled in v of v ∈ V_0, tr_0 = (V_0, E_0, M_0, A_0)});
4      Tr := {tr_0}; D := ∅; trans := ∅; vert := ∅;
5      G := (Tr, D, trans, vert);
6      if tr_0 ∈ Tr_f then
7          return false;
8      if ∄t ∈ T enabled in tr_0 then
9          return true;
10     while not q.isEmpty() do
11         s := q.dequeue();
12         tr' := fire(s.tr, s.t, s.v);
13         if tr' ∈ G.Tr then
14             G := (Tr, D ⊎ {(s.tr, tr')}, trans ⊎ {(s.tr, tr', s.t)}, vert ⊎ {(s.tr, tr', s.v)});
15         else
16             G := (Tr ⊎ {tr'}, D ⊎ {(s.tr, tr')}, trans ⊎ {(s.tr, tr', s.t)}, vert ⊎ {(s.tr, tr', s.v)});
17             if tr' ∉ Tr_f then
18                 if ∄t ∈ T enabled in tr' then
19                     return true;
20                 else
21                     q.enqueueAll({(tr', t', v') : t' enabled transition of N in v' of v' ∈ V', tr' = (V', E', M', A')});
22     return false;
```

call is annotated with the placeholder $init_e \in DUMMY_D$ for i_e. The return statement of the procedure d is represented by the final transition $t_{fi,d}$. In the same way, for service E, the RPN fragment N_E represents the program points i_e and r_e with places that carry the same names. The abstract transition t_{e1} is the procedure call of d in e, and the return statement of e is represented by $t_{fi,e}$. The composition of N_D and N_E to RPN N_{sos}, shown in Figure 4c, is simply the union of the sets. Graphically, only the placeholders are replaced using the function $\mathscr{D}_{(D,E)}$ with $\mathscr{D}_{(D,E)}(init_e) = i_e$ and $\mathscr{D}_{(E,D)}(init_d) = i_d$ change (highlighted in red).

3.3. Procedural Search for RPN-Deadlocks

In order to find deadlocks in the recursive Petri net composition, we present a breadth-first search procedure that iteratively constructs the reachability graph of the recursive Petri net (Procedure 1).

Starting from an initial extended marking tr_0 of a marked recursive Petri net (N, tr_0, Tr_f, tr_i), all enabled transitions t for all vertices v in tr_0 are added to the queue q as tuples (ll. 1-3). The constructed reachability graph G consists initially only of the initial extended marking tr_0 (ll. 4-5). We check if tr_0 is already in the set of final extended markings Tr_f (ll. 6-7), as tr_0 is the only reachable state. This allows for an early return of $false$, since tr_0 is not a deadlock by definition of Tr_f, and tr_0 is the only reachable extended marking from tr_0. Likewise, if tr_0 was not in the set of final extended markings Tr_f but no transition is enabled, tr_0 is a deadlock, which returns *true* (ll. 8-9). Otherwise, the reachability graph is constructed iteratively (ll. 10-22). As long as an enabled transition in any vertex of any non-final extended marking constructed so far has not been checked, there is still a tuple (tr, t, v) in the queue that will be removed as s (l. 11). Since the transition t is enabled in v, a vertex in tr, the transition is fired, resulting in a new extended marking tr' (l. 12). If tr is already a node in the reachability graph, an edge (tr, tr') is added. The new edge is annotated with the transtion t and the vertex v that fired as $tr \xrightarrow{t} tr'$ (ll. 13-14). If tr' is not already in the reachability graph (ll. 15-21), a new node is added to the reachability graph instead, along with the edge (tr, tr'). The new edge is annotated with t and v, the

transition t that fired in vertex v as $tr \xrightarrow{t} tr'$ (l. 16). If the new node tr' is not in the set of final extended markings Tr_f, then tr' is checked for enabled transitions (ll. 17-21). If no transition is enabled in tr', a deadlock is found and *true* is returned (ll. 18-19). Otherwise, for each enabled transition t' in any vertex v' of the extended marking tr', a tuple (tr', t', v') is added to the queue. If the new node tr' is in the set of final extended markings Tr_f, no transition will fire from tr', regardless of whether a transition might be enabled or not (skipping ll. 18-21). When the last tuple was removed from the queue and no new tuples were added, the reachability graph construction is complete. All enabled transitions in any vertex of any extended marking in the reachability graph have been considered (unless disregarded in final extended markings), no deadlock was reachable from the initial extended marking tr_0, and $false$ is returned (l. 22).

Due to the first-in-first-out (FIFO) principle of the queue, for an existing node tr' in ll. 13-14 of Procedure 1, all enabled transtions in tr' have either been considered already, tr' was already in the set of final extended markings, or the remaining enabled transitions of tr' will be considered next when $tr = tr'$. These transitions were added along with the transition t with $tr \xrightarrow{t} tr'$. Thus, no new tuple need to be added to the queue. Since each new node is checked against the set of final extended markings, no extended markings not reachable from tr_0 are added. Likewise, since every new node is checked for the deadlock property, no deadlock can be missed. The extended marking tr_i in the input of Procedure 1 is not considered and can be selected arbitrarily for N.

Procedure 1 can terminate either if a deadlock was found or if no more enabled transitions have to be considered, i.e., if the reachability graph was finite. However, it will not terminate if the marked recursive Petri net (N, tr_0, Tr_f, tr_i) is deadlock-free and unbound in the sense of regular Petri nets, where any set of places may gather an infinite number of tokens if the recursive Petri net is free of deadlocks. Further, the recursion of a recursive Petri net may be unbound, so a deadlock-free recursive Petri net may indefinitely call itself and create new child vertices. An unbound recursive Petri net with a deadlock may still lead to termination due to the breadth-first search: before descending into the next recursion, all alternative enabled transitions from the same extended marking were considered first because of the FIFO principle of the queue. We will now apply the analysis to a service-oriented system known to have a deadlock in order to validate our approach by a case study.

4. Case Study on a Service-Oriented System

Weißbach [10] showed for the service-oriented system in Figure 5 that a context-insensitive abstraction to Petri nets is deadlock-free for the call $main(2)$, whereas the deadlock could be detected in a (P,G)-PRS abstraction [10]. In this section, we will apply the analysis using recursive Petri nets (RPN) to the example from [10, p. 77]. The service-oriented system consists of the four services M, A, B, and C. Service M requires the synchronous procedure a and the asynchronous procedures b and c in the interface R_M. Service A provides the synchronous procedure a in interface I_A and requires the asynchronous procedures b and c in R_A. Service B provides the asynchronous procedure b through I_B, and Service C provides the asynchronous procedure c through I_C. The abstraction in Table 1a and Table 1b is applied to each one of the services M, A, B, and C, with the dummy places $init_a$, $init_b$, and $init_c$ used as placeholders for the respective procedures. The result is the four RPN services N_M, N_A, N_B, and N_C in Figure 6. Composing the RPN fragments to one recursive Petri net applies the dummy replacement functions $\mathscr{D}_{(S,S')} : Dummy_S \to P_{S'}, init_p \mapsto i_p$. As such, $\mathscr{D}_{(M,A)}(init_a) = i_a, \mathscr{D}_{(M,B)}(init_b) = \mathscr{D}_{(A,B)}(init_b) = i_b$, and $\mathscr{D}_{(M,C)}(init_c) = \mathscr{D}_{A,C}(init_c) = i_c$ are replaced in the initial markings annotated to each abstract transition in Figure 6.

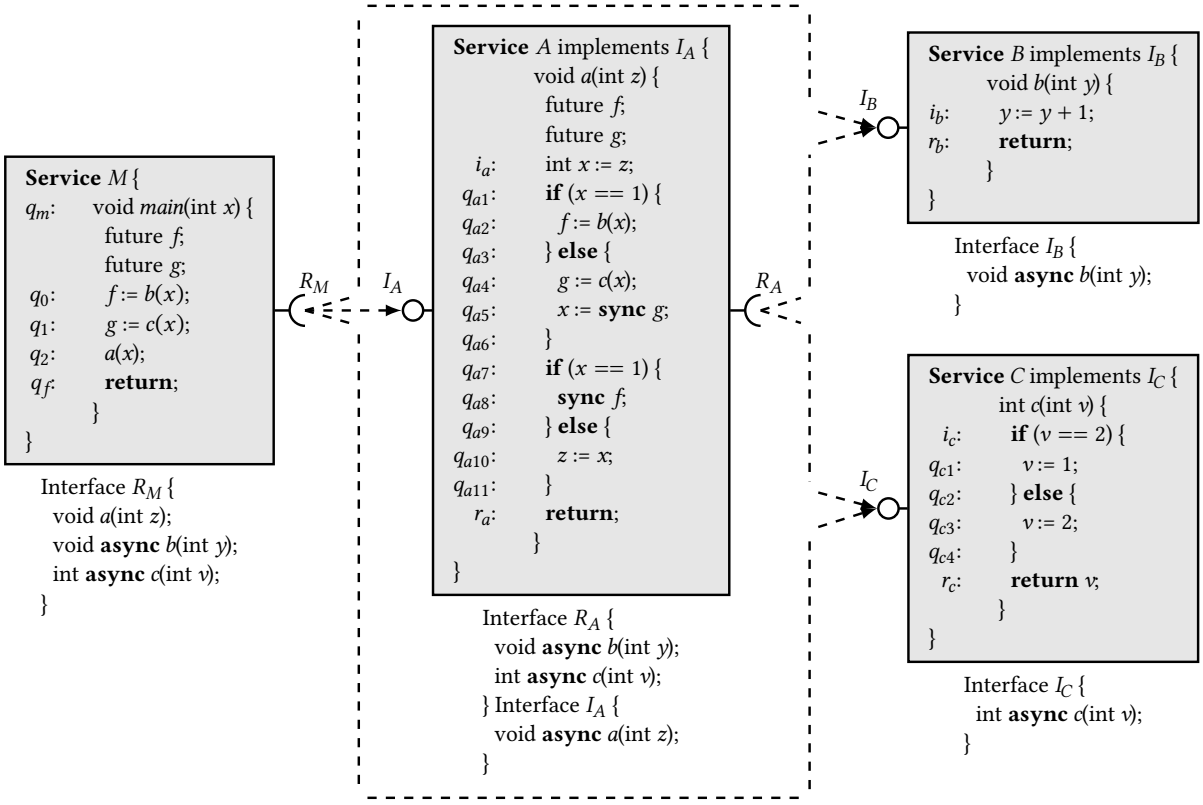


Figure 5: Serivces M, A, B, C with required interfaces R_M and R_A, provided interfaces I_A, I_B, I_C, in the service-oriented language $\mathcal{XYZ}$, adapted from [10, p. 77]

The composition is the recursive Petri net $N_{sos} \triangleq (P_{sos}, T_{sos}, W^-_{sos}, W^+_{sos}, \Omega_{sos})$ with

$$
\begin{aligned}
P_{sos} &\triangleq P_M \uplus P_A \uplus P_B \uplus P_C \\
T_{sos} &\triangleq T_M \uplus T_A \uplus T_B \uplus T_C \\
W^-_{sos} &: P_{sos} \times T_{sos} \to, (p,t) \mapsto n; W^-_M((p,t)) = n \vee W^-_A((p,t)) = n \vee W^-_B((p,t)) = n \vee W^-_C((p,t)) = n \\
W^+_{sos} &: P_{sos} \times T_{sos} \to, (p,t) \mapsto n; W^+_M((p,t)) = n \vee W^+_A((p,t)) = n \vee W^+_B((p,t)) = n \vee W^+_C((p,t)) = n \\
\Omega_{sos} &: T_{sos,ab} \to \mathbb{N}^{|P|}, t_{m3} \mapsto i_a, t_{m6} \mapsto i_b, t_{m7} \mapsto i_c, t_{a17} \mapsto i_b, t_{a18} \mapsto i_c.
\end{aligned}
$$

We define the initial extended marking of N_{sos} to be $tr_0 \triangleq (\{v_0\}, v_0 \mapsto q_m, \emptyset, \emptyset)$ and the set of final extended markings to be $Tr_f \triangleq \{\bot\}$ (the set that only contains empty extended marking). For the arbitrary extended marking required by Procedure 1, we choose tr_0. Procedure 1 is then applied to the marked recursive Petri net $(N_{sos}, tr_0, Tr_f, tr_0)$. Since the full reachability graph generated by Procedure 1 for this net is quite large, we limit the execution to the path that leads to the deadlock in Table 2. The numbering of the steps follows [10, p. 82]. Because step (8) is not needed in the RPN abstraction, it is skipped and marked with $*$.

Weißbach [10] showed that the deadlock in step (24) of Table 2 is present in the semantics of the service-oriented system *sos*, composed of services M, A, B, and C [10, p. 82]. Further, a (P,G)-PRS abstraction can find this deadlock [10, p. 117], whereas Petri net abstractions that do not take the context of procedure calls into account are deadlock-free [10, pp. 97-101]. We have shown in this section that our abstraction was able to find the deadlock as well. However, there are still some limitations that need to be discussed in Sect. 5.

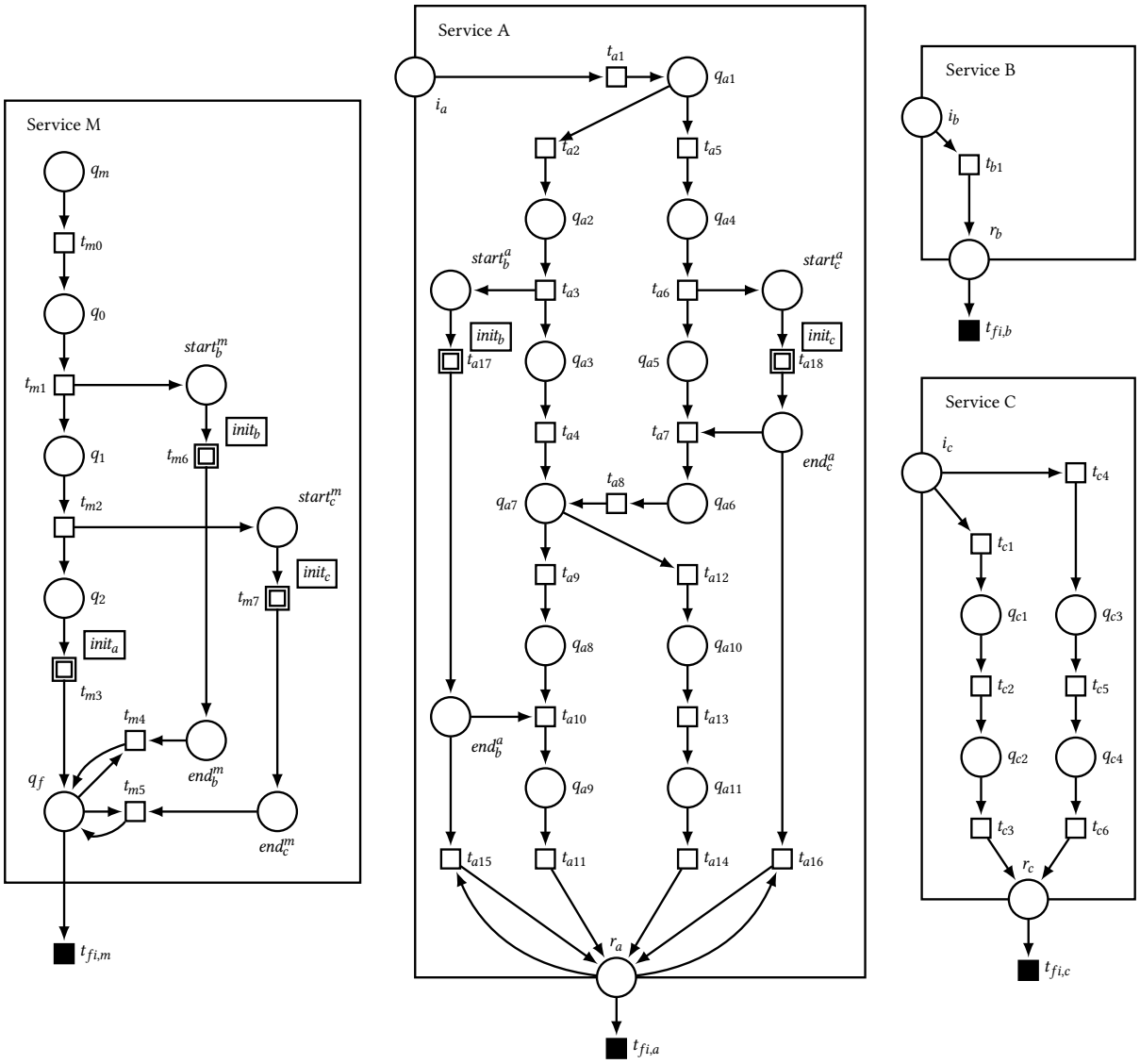


Figure 6: RPN abstractions N_M, N_A, N_B and N_C of services M, A, B and C, using the abstraction function α_{RPN} described by Table 1a and Table 1b

5. Discussion

For the abstraction of the explicit synchronization of asynchronous procedure calls in Table 1b, Table 2 shows that the caller (procedure a) will wait for the return of the callee (procedure b). However, if procedure b were to be synchronized at the return statement of procedure a, then $t_{fi,a}$ would be enabled, even though procedure b may not have returned yet. Since there is inherently no way for the corresponding RPN threads to communicate other than through the firing of a final transition in the callee thread, the structure of the model must be adapted. Otherwise, a threadlock in the callee may be cut out of the tree prematurely, and a deadlock may not be discovered. We plan to modify the abstraction similarly to the proof of the inclusion of the PAN languages in the RPN languages (cpt in [14]), where an additional place will indicate to the caller thread and the final transition if there is currently a child thread representing an asynchronous call from the abstract transition. The absence of a token in this place models the ongoing procedure call, blocking the caller thread from returning prematurely. Further, we plan on showing whether the modeled RPN preserves all deadlocks that the (P,G)-PRS abstraction can preserve.

The limitations for the termination of Procedure 1 were already discussed. Additionally, the reachability graph can grow exponentially in space and time if every transition is enabled in every state. For

Table 2
Execution in the composed recursive Petri net N_{sos}, composed from the abstrations N_M, N_A, N_B and N_C

(0)	$\underline{(q_m)}$	$\xrightarrow{t_{m0}}$	(1)	$\underline{(q_0)}$	$\xrightarrow{t_{m1}}$	(2)	$(\underline{q_1} \parallel start_b^m)$	$\xrightarrow{t_{m2}}$
(3)	$(q_2 \parallel \underline{start_b^m} \parallel start_c^m)$	$\xrightarrow{t_{m6}}$	(4)	$(q_2 \parallel start_c^m) \xrightarrow{t_{m6}} \underline{(i_b)}$	$\xrightarrow{t_{b1}}$	(5)	$(q_2 \parallel start_c^m) \xrightarrow{t_{m6}} \underline{(r_b)}$	$\xrightarrow{t_{fi,b}}$
(6)	$(q_2 \parallel end_b^m \parallel \underline{start_c^m})$	$\xrightarrow{t_{m7}}$	(7)	$(\underline{q_2} \parallel end_b^m) \xrightarrow{t_{m7}} (i_c)$	$\xrightarrow{t_{m3}}$	(8)	*	*
(9)	$(end_b^m) \xrightarrow{t_{m7}} \underline{(i_c)}$, $(end_b^m) \xrightarrow{t_{m3}} (i_a)$	$\xrightarrow{t_{c1}}$	(10)	$(end_b^m) \xrightarrow{t_{m7}} \underline{(q_{c1})}$, $(end_b^m) \xrightarrow{t_{m3}} (i_a)$	$\xrightarrow{t_{c2}}$	(11)	$(end_b^m) \xrightarrow{t_{m7}} \underline{(q_{c2})}$, $(end_b^m) \xrightarrow{t_{m3}} (i_a)$	$\xrightarrow{t_{c3}}$
(12)	$(end_b^m) \xrightarrow{t_{m7}} \underline{(r_c)}$, $(end_b^m) \xrightarrow{t_{m3}} (i_a)$	$\xrightarrow{t_{fi,c}}$	(13)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(i_a)}$	$\xrightarrow{t_{a1}}$	(14)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(q_{a1})}$	$\xrightarrow{t_{a5}}$
(15)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(q_{a4})}$	$\xrightarrow{t_{a6}}$	(16)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a5} \parallel \underline{start_c^a})$	$\xrightarrow{t_{a18}}$	(17)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a5}) \xrightarrow{t_{a18}} \underline{(i_c)}$	$\xrightarrow{t_{c1}}$
(18)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a5}) \xrightarrow{t_{a18}} \underline{(q_{c1})}$	$\xrightarrow{t_{c2}}$	(19)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a5}) \xrightarrow{t_{a18}} \underline{(q_{c2})}$	$\xrightarrow{t_{c3}}$	(20)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a5}) \xrightarrow{t_{a18}} \underline{(r_c)}$	$\xrightarrow{t_{fi,c}}$
(21)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(q_{a5} \parallel end_c^a)}$	$\xrightarrow{t_{a7}}$	(22)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(q_{a6})}$	$\xrightarrow{t_{a8}}$	(23)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} \underline{(q_{a7})}$	$\xrightarrow{t_{a9}}$
			(24)	$(end_b^m \parallel end_c^m) \xrightarrow{t_{m3}} (q_{a8})$ *Deadlock*	$\not\rightarrow$			

Petri nets, further deadlock analyses have been described, such as the analysis for siphons and traps in the Petri net structure and stubborn sets in the reachability graph [20, 21]. We would like to transfer those methods to the analysis of recursive Petri nets.

All steps of our analysis were executed manually, which may be acceptable for small systems such as Figure 5. However, for the analysis of large service-oriented systems with many services, automation of all steps would be preferable. Therefore, we plan on implementing tools for the analysis. These tools should also include optimizations for removing unnecessary places and transitions from the abstraction, such as assignments or if-then-else statements that do not contain procedure calls.

Threats to validity

Henceforth, we consider both threats to validity [22] as well as limitations of the presented approach. Regarding *external validity*, although we did not select a concrete service-oriented system with a deadlock, we specifically selected an established example with a deadlock, undetectable by existing abstractions that do not take concurrency and recursion into account (cp. [10]). For *internal validity*, we concede that the presented abstraction of a service-oriented system to RPN might abstract a deadlock, yet in the future, we intend to proof that the transformation is sound and complete. In case of *construct validity*, we showed that the presented abstraction and deadlock detection procedure correctly identifies a deadlock in an RPN. Moreover, we maintain that in case the RPN contains a deadlock, the procedure will find it. On a more technical level, the presented procedure requires optimizations to be viable for larger reachability graphs and thus for larger service-oriented systems, such optimizations were omitted to ensure the comprehensibility of the presented procedure.

6. Related Work

Abstraction-based deadlock analysis in service-oriented systems has been attempted with several kinds of Petri nets and models. Reisig and Desel [23] provide an overview of modelling systems with Petri nets, including ones modelling deadlocks and distributed systems. However, they do not consider deadlocks in the context of service-oriented systems with unbound recursion.

An extension of Petri nets with recursion was first proposed in [12] to model a multi-agent system. In this refinement-based approach, deadlocks are avoided through a separate coordination algorithm between the agents. Shortly after, Haddad and Poitrenaud [13] introduced recursive Petri nets with the initial definition we utilize in this paper. Haddad and Poitrenaud [14] noted that the set of all deadlocks in a recursive Petri net can be described by a semi-linear set but did neither provide a formal definition of deadlocks in RPNs nor consider deadlock analysis of RPNs. While decidability of reachability in RPNs was discussed in [13], further results on coverability, finiteness, and termination were shown in [16, 24]. These results form the foundation for our deadlock analysis in RPNs, which bears similarity to an approach with reachability Petri nets [25] and the more general approach with a coverability graph for an infinite number of states [24]. Beside these, other Petri net extensions tackled recursion, such as nested Petri nets [26] and Object Petri Nets [27]. However, for the former, reachability is undecidable [28], and for the latter, limited to bound recursion [27]. Thus, reachability-based analyses are not decidable for either extension.

In case of more general abstractions of service-oriented systems, process algebras have been considered. Among other findings, Mayr [29] showed the equivalence of (P, P)-PRS to Petri nets [29], placing Petri's formalism in a hierarchy of process algebras [30, 31]. Within Mayr's hierarchy of process algebras [31], (G, G)-PRS and Petri nets have been considered for deadlock analysis in service-oriented systems [32, 10, 8]. While the deadlock analysis using Petri net-based abstractions might fail to detect a deadlock in a service-oriented system, the (G, G)-PRS abstraction does not, because it is context-sensitive wrt. the recursive service calls [10]. Expanding on these results, Weißbach et al. [33] showed that context-sensitive service composition needs at most a call depth of two to guarantee a correct deadlock analysis when employing abstract state machines. Heike et al. [19] showed that protocol conformance checking in component-based and service-oriented systems requires the most general class of (G, G)-PRS when exception handling has to be taken into account [19]. Protocols for deadlock analysis are not specifically part of [19], but a refinement-based approach to compose deadlock-free systems with LTS is presented in [34]. Refining services is more easily done with new systems, whereas our abstraction-based approach can be applied to compositions of existing service-oriented systems.

In sum, our approach considers recursive Petri nets (RPNs) as a different model with both unbound recursion and unbound concurrency. We argue that RPNs are better suited for an abstraction-based deadlock analysis in service-oriented systems than PRS or abstract state machines, as existing Petri net-based tools must only be extended to support RPNs.

7. Conclusion

To overcome the limitations of previous Petri net-based approaches for deadlock analysis [8, 33], in this paper, we described a novel abstraction-based approach for the deadlock analysis of service-oriented systems using recursive Petri nets (RPNs). Like related approaches, the presented analysis is conducted following three steps: abstracting RPNs from services, composing these RPNs, and deadlock analysis in RPNs. To the best of our knowledge, we introduced the first procedure for deadlock analysis on RPNs, based on a breadth-first search in the reachability graph of the RPN. By applying our analysis to the counterexample introduced by Weißbach [10], we showed that our abstraction and deadlock analysis with RPNs was able to find the deadlock, which would remain undetected employing Petri net-based approaches, such as Workflow Nets [35].

In future work, we aim to determine whether the limitations of our RPN abstraction are the result of the specific abstraction employed or inherent to RPNs. In particular, we intend to show that abstractions of (P,G)-PRS and RPN are equally fit for deadlock analysis by showing that RPN languages and (P,G)-PRS languages are equivalent. Last but not least, we want to extend existing Petri net tools to both support RPNs and our deadlock analysis, enabling the analysis of larger service-oriented systems.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: Abstract drafting and LanguageTool in order to: Grammar and spelling check. After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] F. Bathelt, S. Lorenz, J. Weidner, M. Sedlmayr, I. Reinecke, Application of modular architectures in the medical domain - a scoping review, Journal of Medical Systems 49 (2025). doi:10.1007/s10916-025-02158-3.

[2] D. Pal, T. Triyason, S. Funikul, Smart homes and quality of life for the elderly: A systematic review, in: 2017 IEEE International Symposium on Multimedia (ISM), IEEE, 2017, pp. 413–419. doi:10.1109/ism.2017.83.

[3] S. Van Hoecke, R. Verborgh, D. Van Deursen, R. Van de Walle, Samus: Service-oriented architecture for multisensor surveillance in smart homes, The Scientific World Journal 2014 (2014) 1–9. doi:10.1155/2014/150696.

[4] W. Li, Y.-H. Lee, W.-T. Tsai, J. Xu, Y.-S. Son, J.-H. Park, K.-D. Moon, Service-oriented smart home applications: composition, code generation, deployment, and execution, Service Oriented Computing and Applications 6 (2011) 65–79. doi:10.1007/s11761-011-0086-7.

[5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al., Ros: an open-source robot operating system, in: ICRA workshop on open source software, volume 3, Kobe, 2009, p. 5.

[6] J. S. Grover, C. Liu, K. Sycara, Deadlock Analysis and Resolution for Multi-robot Systems, Springer International Publishing, 2021, pp. 294–312. doi:10.1007/978-3-030-66723-8_18.

[7] N. Gafur, G. Kanagalingam, A. Wagner, M. Ruskowski, Dynamic collision and deadlock avoidance for multiple robotic manipulators, IEEE Access 10 (2022) 55766–55781. doi:10.1109/access.2022.3176626.

[8] M. Weißbach, W. Zimmermann, On Limitations of Abstraction-Based Deadlock-Analysis of Service-Oriented Systems, Springer International Publishing, 2020, pp. 79–90. doi:10.1007/978-3-030-63161-1_6.

[9] R. Mayr, Process rewrite systems, Information and Computation 156 (2000) 264–286. URL: https://www.sciencedirect.com/science/article/pii/S0890540199928262. doi:10.1006/inco.1999.2826.

[10] M. Weißbach, Deadlockanalyse Service-orientierter Softwaresysteme, phdthesis, Martin-Luther-Universität Halle-Wittenberg, Halle (Saale), 2019. URL: https://opendata.uni-halle.de/bitstream/1981185920/32374/1/dissertation_online.pdf.

[11] E. J. Hartnick, M. Weißbach, Abstraction-based deadlock analysis of service-oriented systems with recursive petri nets, in: Joint Proceedings of the Workshops at the 46th International Conference on Application and Theory of Petri Nets and Concurrency: Petri Nets and Software Engineering (PNSE'25), Algorithms & Theories for the Analysis of Event Data (ATAED'25), and Petri Net Games, Examples and Quizzes for Education, Contest and Fun (PeNGE'25), volume 3998, 2025. URL: https://ceur-ws.org/Vol-3998/poster5.pdf.

[12] A. E. F. Seghrouchni, S. Haddad, A recursive model for distributed planning, in: Second International Conference on Multiagent Systems, 1996, pp. 307–314. URL: https://cdn.aaai.org/ICMAS/1996/ICMAS96-039.pdf.

[13] S. Haddad, D. Poitrenaud, Theoretical Aspects of Recursive Petri Nets, Springer Berlin Heidelberg, 1999, pp. 228–247. doi:10.1007/3-540-48745-x_14.

[14] S. Haddad, D. Poitrenaud, Modelling and Analyzing Systems with Recursive Petri Nets, Springer US, 2000, pp. 449–458. doi:10.1007/978-1-4615-4493-7_48.

[15] S. Haddad, D. Poitrenaud, Recursive petri nets, Acta Informatica 44 (2007) 463–508. doi:10.1007/s00236-007-0055-y.

[16] A. Finkel, S. Haddad, I. Khmelnitsky, Coverability and Termination in Recursive Petri Nets, Springer International Publishing, 2019, pp. 429–448. doi:10.1007/978-3-030-21571-2_23.

[17] K. B. Laskey, K. Laskey, Service-oriented architecture, WIREs Computational Statistics 1 (2009) 101–105. doi:https://doi.org/10.1002/wics.8.

[18] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, B. A. Hamilton, Reference model for service oriented architecture 1.0, OASIS standard, 2006.

[19] C. Heike, W. Zimmermann, A. Both, On expanding protocol conformance checking to exception handling, Service Oriented Computing and Applications 8 (2014) 299–322. doi:10.1007/s11761-013-0146-2.

[20] A. Valmari, Stubborn sets for reduced state space generation, Springer Berlin Heidelberg, 1991, pp. 491–515. doi:10.1007/3-540-53863-1_36.

[21] K. Barkaoui, J.-F. Pradat-Peyre, On liveness and controlled siphons in Petri nets, Springer Berlin Heidelberg, 1996, pp. 57–72. doi:10.1007/3-540-61363-3_4.

[22] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. Al-Doori, N. Boltz, M. Ehl, et al., Evaluation methods and replicability of software architecture research objects, in: 2022 IEEE 19th international conference on software architecture (ICSA), IEEE, 2022, pp. 157–168. doi:10.1109/ICSA53651.2022.00023.

[23] W. Reisig, J. Desel, Konzepte der petrinetze, Informatik-Spektrum 37 (2014) 172–190. doi:10.1007/s00287-013-0758-0.

[24] A. Finkel, S. Haddad, I. Khmelnitsky, Coverability, termination, and finiteness in recursive petri nets, Fundamenta Informaticae 183 (2022) 33–66. doi:10.3233/fi-2021-2081.

[25] F. S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, G. Zavattaro, A Petri Net Based Analysis of Deadlocks for Active Objects and Futures, Springer Berlin Heidelberg, 2013, pp. 110–127. doi:10.1007/978-3-642-35861-6_7.

[26] I. A. Lomazova, P. Schnoebelen, Some Decidability Results for Nested Petri Nets, Springer Berlin Heidelberg, 2000, pp. 208–220. doi:10.1007/3-540-46562-6_18.

[27] R. Valk, Petri Nets as Token Objects, Springer Berlin Heidelberg, 1998, pp. 1–24. doi:10.1007/3-540-69108-1_1.

[28] I. A. Lomazova, Nested petri nets: Multi-level and recursive systems, Fundamenta Informaticae (2001) 283–293. doi:https://doi.org/10.3233/FUN-2001-473-409.

[29] R. Mayr, Combining petri nets and pa-processes, in: M. Abadi, T. Ito (Eds.), Theoretical Aspects of Computer Software, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 547–561. doi:10.1007/BFb0014567.

[30] C. A. Petri, Kommunikation mit Automaten, phdthesis, Fakultät für Physik und Mathematik

der Technischen Hochschule Darmstadt, Bonn, 1962. URL: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

[31] R. Mayr, Decidability and Complexity of Model Checking Problems for Infinite-State Systems, phdthesis, Fakultät für Informatik der Technischen Universität München, 1998. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d65555934b3b44d73d3f22b37b123aeaa2c38e4c.

[32] M. Weißbach, Deadlock analysis of service-oriented systems with recursion and concurrency, in: A. Lazovik, S. Schulte (Eds.), Advances in Service-Oriented and Cloud Computing, volume 707, Springer, Cham., 2018, pp. 247–259. doi:10.1007/978-3-319-72125-5_20.

[33] M. Weißbach, W. Zimmermann, T. Kühn, Workflow Net Compositions for the Analysis of Service-Oriented Systems, Springer Nature Switzerland, 2025, pp. 235–250. doi:10.1007/978-3-031-84617-5_18.

[34] M. Bravetti, G. Zavattaro, Contract-Based Discovery and Composition of Web Services, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 261–295. doi:10.1007/978-3-642-01918-0_7.

[35] W. M. Van der Aalst, The application of petri nets to workflow management, Journal of circuits, systems, and computers 8 (1998) 21–66. doi:https://doi.org/10.1142/S0218126698000043.