An Approach to Tackle Livelock-freedom in SOA

Christian Stahl[1,2,⋆] and Karsten Wolf[3,⋆⋆]

[1] Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
stahl@informatik.hu-berlin.de

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[3] Universität Rostock, Institut für Informatik
18051 Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. We calculate a fixed finite set of *state space fragments* for a service P, where each fragment carries a part of the whole behavior of P. By composing these fragments according to the behavior of a service R we build the state space of their composition $P \oplus R$ which can be checked for deadlocks and livelocks. We show that this approach is applicable to realize a "find" request by a service R with a provided service P in SOA.

1 Introduction

In the paradigm of service-oriented computing (SOC) a *service* serves as a building block for designing flexible business processes by composing multiple services. Service-oriented architectures (SOA) serve as an enabler for publishing services via the Internet such that these services can be automatically found. By dynamically binding published services with other services, a composed service that achieves certain business goals can be designed.

In SOA, we would like to answer a "find" request by a service R with a provided service P such that $P \oplus R$ forms a sound, i.e. a deadlock-free and livelock-free system. The apparent approach is to have P (or a public view of P) stored in the repository and to construct $P \oplus R$ for checking the absence of deadlocks and livelocks upon "find". This approach is, however, not feasible due to state space explosion, and the necessity to have P (or a formally equivalent public view) stored in the repository. State space reduction during the construction of $P \oplus R$ might help a lot but needs to be performed for each "find" request.

Another approach is to publish an operating guideline [1] for each service P; that is, an operational description of all services R such that $P \oplus R$ is sound. To answer a "find" request by R, one has to check whether R matches with the operating guidelines of P [1]. However, in this approach soundness is restricted to deadlock-freedom so far and hence livelocks in $P \oplus R$ are possible.

⋆ Funded by the DFG project "Substitutability of Services" (RE 834/16-1).
⋆⋆ Supported by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

In this paper, we propose a novel approach. We still build the state space $P \oplus R$, but not from operational descriptions of P and R. Instead, we calculate a finite set of *state space fragments* for a service P. Each fragment carries a *part* of the whole behavior of P. These fragments are published in a repository. Upon a "find" request by a service R, the state space of $P \oplus R$ is calculated by composing fragments of P according to the behavior of R. The resulting state space can then be checked for deadlocks and livelocks using a model checker. The approach has two advantages:

1. The construction of fragments and their internal state space reduction is done once for each published service at the "publish" phase. That way, computational efforts are shifted from "find" to "publish". This is a clear advantage as we expect the number of "find" to be much higher than the number of "publish".
2. When reducing the size of fragments, we can apply reduction techniques which are different from standard state space reduction techniques used in model checking. In fact, we may reduce the transition system *after* having computed it.

Section 2 formalizes fragments, shows how fragments for a given service P can be computed, and it presents how the state space $P \oplus R$ can be built from the fragments of P. Section 3 sketches several abstractions to condense fragments while preserving deadlocks and livelocks and, finally, Sect. 4 concludes the paper.

2 Calculating State Spaces From Fragments

2.1 Formalizing Fragments

In this section, we define fragments and connections between these fragments.

A *(state space) fragment* $Frag = (V, E, F)$ is a graph that consists of a set V of *nodes*, a set $E \subseteq V \times V$ of (directed) *edges*, and a set $F \subseteq V$ of *final nodes*. We assume that different fragments have disjoint sets of nodes.

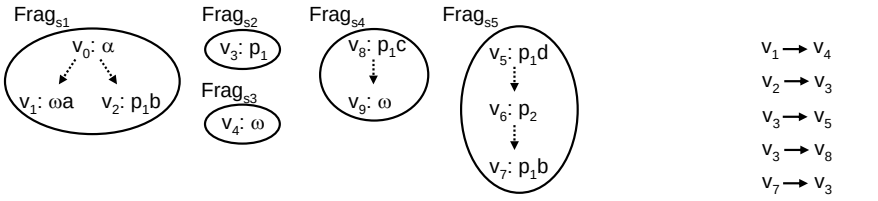
Let x be an element of some fixed set M. An *instance* $Frag(x)$ of a fragment $Frag$ is built by renaming the constituents as follows: $v \mapsto [v, x]$, $e = [v_1, v_2] \mapsto [[v_1, x], [v_2, x]]$ for all $v \in V$ and $e \in E$. That way, the structure is preserved but the nodes get previously unused names.

To plug different fragments yielding again a state space, we define connections which link states of one fragment to states of another fragment. A *connection* $C_{Frag_1, Frag_2}$ between fragments $Frag_1$ and $Frag_2$ is a subset of $V_{Frag_1} \times V_{Frag_2}$.

If $C_{Frag_1, Frag_2}$ is a connection between fragments $Frag_1$ and $Frag_2$, then $C_{Frag_1, Frag_2}(x, y) = \{([v_1, x], [v_2, y]) \mid (v_1, v_2) \in C\}$ is a connection between $Frag_1(x)$ and $Frag_2(y)$.

Consider the fragments and the connections depicted in Fig. 1. For instance, we have fragment $\mathsf{Frag_{s1}} = (\{\mathsf{v_0}, \mathsf{v_1}, \mathsf{v_2}\}, \{(\mathsf{v_0}, \mathsf{v_1}), (\mathsf{v_0}, \mathsf{v_2}), \emptyset\})$ and connection $C_{\mathsf{Frag_{s1}}, \mathsf{Frag_{s3}}} = \{(\mathsf{v_1}, \mathsf{v_4})\}$. Thereby, $\mathsf{v_0}$ relabels α, $\mathsf{v_1}$ relabels $\omega\mathsf{a}$, etc.

Given a set of fragments $Frag_1, \ldots, Frag_n$ and connections $C_1, \ldots, C_m$, a transitions system $TS = (V, E)$ is defined by $V = \bigcup_{k=1}^{n} V_{Frag_k}$ and $E =$



(a) Fragments (b) Connections

Fig. 1. Fragments and connections. Dotted (solid) lines denote fragment internal transitions (connections).

$\bigcup_{i=1}^{n} E_{Frag_i} \cup \bigcup_{j=1}^{m} E_{C_j}$. Thereby, several instances of one and the same fragment or connection may be used to build TS.

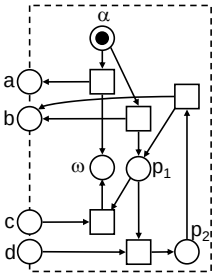
In the following, we introduce our service model open nets and show how fragments and connections of an open net can be calculated.

2.2 Open Nets and Most Permissive Strategy

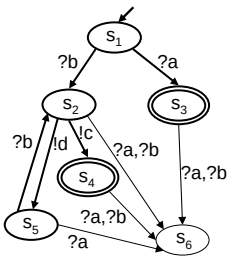
We use *open nets* as a service model. An open net N consists of a Petri net together with an *interface*. The interface is divided into a set of *input places* and *output places*. Input places have an empty preset, output places have an empty postset. Furthermore, N has a distinguished *initial marking* m_0, and a set Ω of *final markings* such that no transition of N is enabled at any $m \in \Omega$. We further require that in the initial and the final markings the interface places are not marked.

The behavior of an open net is defined using the standard Petri net semantics [2]. With $R_N(m_0)$ we denote the set of reachable markings of N.

For example, the open net P depicted in Fig. 2(a) has an initial marking $m_{0P} = [\alpha]$ and the set of final markings is defined by $\Omega_P = \{[\omega]\}$. P has two input places c and d and two output places a and b that are depicted on the dashed frame.



(a) Open net P



(b) Most permissive strategy R^* for P

Fig. 2. Open net P and its most permissive strategy R^*. $?x$ ($!x$) denotes a sending (receiving) message x that produces a token on input place x (consumes a token from output place x) in P.

As a correctness criterion for an open net N we require the absence of deadlocks and livelocks in N. N is *deadlock-free and livelock-free* if for all reachable markings $m \in R(m_0)$, $R_N(m) \cap \Omega_N \neq \emptyset$.

For the *composition* of two open nets M and N, we require that the input places of M are the output places of N and vice versa. M and N can be composed by merging input places of M with equally labeled output places of N and vice versa.

As we are interested in composing open nets such that the composition is deadlock-free and livelock-free we define the notion of a strategy. An open net M is a *strategy* for an open net N if $M \oplus N$ is deadlock-free and livelock-free.

In [1] it has been proven that there always exists a *most permissive strategy* R^* for an open net N that has richer behavior than every other strategy for N.

The most permissive strategy for P (of Fig. 2(a)) is depicted in Fig. 2(b). It is an automaton (which can easily be transformed in a state machine and by adding an input (output) place for each $?x$ ($!x$) to an open net) with initial state $\mathsf{s_1}$ and two final states $\mathsf{s_3}$ and $\mathsf{s_4}$. State $\mathsf{s_6}$ is depicted for technical purposes only. Every edge to $\mathsf{s_6}$ shows a possible set of messages R^* can receive but that will never occur because P cannot send them.

For R^* we can prove the following useful property.

Lemma 1. *If R is a strategy of P, then R^* weakly simulates R.*

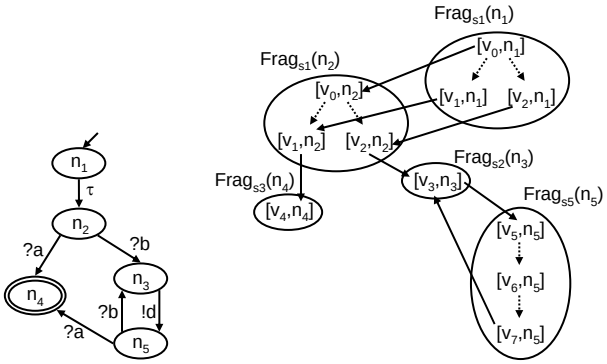
The converse does not hold in general. The automaton R^* (see Fig. 2(b)) weakly simulates R (see Fig. 3(a)) but $P \oplus R$ has a livelock (as we will see later on). In fact, R is an example why the operating guideline approach in [1] is not applicable to tackle livelock-freedom.

When computing R^* we have the information needed to calculate the fragments and connections of P. Each state s of R^* is a fragment. In each state s, R^* has knowledge about the possible markings of P in s. These markings (together with their transitions) are the nodes and the edges of the fragment. Figure 1 shows the fragments and the connections of P. $\mathsf{Frag_{s1}}$ is the fragment derived from $\mathsf{s_1}$, $\mathsf{Frag_{s2}}$ from $\mathsf{s_2}$ and so on. For $\mathsf{s_6}$ there is no fragment. We have relabeled the markings of all fragments by $\mathsf{v_0}, \ldots, \mathsf{v_9}$ to make the internals of P anonymous. For each edge of R^*, we define a connection. The connection is calculated from the edges of R^* and the markings.

Given a service R, the most permissive strategy R^* for P and the fragments and connections of P, we show in the following how a transition system $P \oplus R$ can be constructed.

2.3 Fragments and Connections for P

From the construction of fragments we know that for each state s of R^*, its state space is defined by the fragment $Frag_s$. Furthermore, for each pair of fragments $Frag_1 \neq Frag_2$, the set of edges with source in $Frag_1$ and sink in $Frag_2$ is defined by connection $C_{Frag_1, Frag_2}$. For each fragment $Frag$, let id_{Frag} denote connection $C_{Frag, Frag}$. Then, by Lemma 1, R can only be a strategy for P if R^* (weakly) simulates R.



(a) Requester R (b) TS_R

Fig. 3. Constructing the state space $P \oplus R$ from the fragments for a given service R. Note that the ?b edge in state $\mathsf{n_5}$ yields $[\mathsf{n_4}, \mathsf{s_6}] \in \rho$. However, as $\mathsf{s_6}$ is not reachable, there is no fragment for state $\mathsf{s_6}$ and hence there is no transition from $\mathsf{Frag_5}$ to $\mathsf{Frag_3}$.

Definition 1 (Construction of TS_R). *Let ϱ be simulation relation between R and R^*. Compose transition system TS_R from the following fragments and connections:*

- $FRAG = \{Frag_s(n) \mid [n, s] \in \varrho\}$,
- $CONN = \{C_{Frag_s, Frag_{s'}}(n, n') \mid [n, s] \in \varrho, [n, x, n'] \in \delta_R \ (x \neq \tau), [s, x, s'] \in \delta_{R^*} (\textit{which implies } [n', s'] \in \varrho)\} \cup \{id_{Frag_s}(n, n') \mid [n, s] \in \varrho, [n, \tau, n'] \in \delta_R\}$

The fragment that corresponds to the initial state of R and R^* is unique and it contains the initial state of the resulting transition system TS_R.

In our example, $(\mathsf{n_1}, \mathsf{s_1}), (\mathsf{n_2}, \mathsf{s_1}) \in \varrho$. Thus we add two instances of $\mathsf{Frag_{s1}}$, i.e. $\mathsf{Frag_{s1}(n_1)}$, $\mathsf{Frag_{s1}(n_2)}$. As we have transition $[\mathsf{n_1}, \tau, \mathsf{n_2}] \in \delta_R$ in R we add connection $id_{\mathsf{Frag_{s1}}}(\mathsf{n_1}, \mathsf{n_2})$. Furthermore, we add fragment $\mathsf{Frag_{s3}(n_4)}$ because $(\mathsf{n_4}, \mathsf{s_3}) \in \varrho$. From transition $[\mathsf{n_2}, \mathsf{?a}, \mathsf{n_4}] \in \delta_R$ in R and $[\mathsf{s_1}, \mathsf{?a}, \mathsf{s_3}] \in \delta_{R^*}$ we conclude that connection $C_{\mathsf{Frag_{s1}}, \mathsf{Frag_{s3}}}(\mathsf{n_2}, \mathsf{n_4})$ for ?a has to be added. Figure 3(b) shows the resulting state space $P \oplus R$. TS_R contains a livelock (the nodes of $\mathsf{Frag_{s2}(n_3)}$ and $\mathsf{Frag_{s5}(n_5)}$ have no final node), thus R is no strategy for P.

The resulting transitions system TS_R can be verified for deadlocks and livelocks. Our main result of this paper guarantees that each deadlock and livelock in $P \oplus R$ is preserved in TS_R and vice versa.

Theorem 1. *Let R be a strategy for P and let TS_R be as defined above. Then TS_R is bisimilar to the state space $P \oplus R$.*

3 Deadlock and Livelock-preserving Abstraction

To speed up the model checking run when checking TS_R, we can apply state-of-the-art reduction techniques such as partial-order and symmetry reduction. Besides this, we statically reduce the fragments. This may lead to a smaller TS_R and thus increasing the performance of our approach and, in addition, we need to store smaller fragments in the repository (when publishing P). All abstractions we sketch in the following preserve both deadlocks and livelocks.

For each fragment we compute its strongly connected components (SCCs). It is sufficient to store only SCCs instead of nodes.

To condense the state space of each fragment, we adapt state space condensation rules from [3]. These rules can be applied to each fragment. For example, we can condense the three SCCs in $\mathsf{Frag}_{\mathsf{s5}}$ (see Fig. 1(a)) to a single SCC.

Finally, we can also minimize R, in particular, its τ transitions. For instance, we can apply minimization rules that preserve branching bisimulation. That way, states n_1 and n_2 in Fig. 3(a) could be merged.

4 Conclusion

We have proposed a technique to realize the "find" operation in SOA in case the composed system is required to be free of deadlocks *and* livelocks. We suggest that a service provider publishes a set of state space fragments such that each fragment carries a part of the whole behavior of P. Given a requester R, "find" means to construct the state space $P \oplus R$ from the fragments of P guided by the behavior of R. The resulting state space is checked for deadlocks and livelocks.

Although the space complexity is the product of the state spaces of R and P (in worst case), we assume that applying abstraction techniques results in much smaller states spaces.

We are currently implementing the proposed approach in our analysis tool Fiona [4]. The computed state space $P \oplus R$ can then be checked for deadlocks and livelocks using the model checker LoLA [5]. Future work also includes a case study to validate the strength of the abstraction techniques.

References

1. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings. Volume 4546 of Lecture Notes in Computer Science., Springer-Verlag (2007) 321–341
2. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
3. Juan, E.Y.T., Tsai, J.J.P., Murata, T.: Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules. ACM Trans. on Programming Languages and Systems **20**(5) (1998) 917–979
4. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In Dustdar, S., Fiadeiro, J., Sheth, A., eds.: Fourth International Conference on Business Process Management, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings. Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag (2006) 17–32
5. Schmidt, K.: LoLA: A low level analyser. In Nielsen, M., Simpson, D., eds.: 21st International Conference on Application and Theory of Petri Nets, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding. Number 1825 in Lecture Notes in Computer Science, Aarhus, Denmark, Springer-Verlag (2000) 465–474