Instance Isolation Analysis for Service-Oriented Architectures

Gero Decker, Mathias Weske
Hasso-Plattner-Institute, University of Potsdam, Germany
{gero.decker,weske}@hpi.uni-potsdam.de

Abstract

When a service engages in multiple conversations concurrently, incoming messages must be correlated with messages previously sent or received. Languages such as BPEL incorporate correlation as first-class citizen. However, current verification and testing techniques for service implementations largely ignore possible correlation anomalies as they typically focus on isolated conversations. This paper defines the notion of instance isolation and shows how to check this property. For doing so it introduces ν^*-nets, a Petri net extension with name creation and name passing.

1 Introduction

Distributed systems architectures such as service-oriented architectures (SOA) rely on the notion of message exchanges. Services are loosely coupled components described in a uniform way that can be discovered and composed. One realization of a SOA is the web services platform architecture where services are offered as web services ([4]).

Web services offer a set of operations through ports. This structural interface can be described using WSDL [3]. Conversations are often more complex than simple request / response interactions. As several of such conversations might run concurrently and message exchanges belonging to different conversations are carried out through the same ports, message correlation mechanisms have to be in place. Incoming messages need to be correlated to messages previously sent or received.

The Business Process Execution Language (BPEL [8]) is a popular language for implementing services that can engage in complex conversations. Different formal verification techniques for checking control flow and data flow anomalies [10, 15, 20] as well as test frameworks [17] and test metrics are available. However, these techniques and frameworks focus on individual conversations. Effects occurring in the case of multiple instantiation of BPEL processes involved in concurrent conversations, are neglected so far. Therefore, this paper presents a formal verification technique concentrating on *instance isolation* in complex conversations among services. Instance isolation will we defined on the basis of ν^*-nets, a Petri net [22] extension including name creation and name passing as known from modern process algebras such as π-calculus [19].

The remainder of this paper will be structured as follows. The next chapter will provide a motivating example from the enterprise systems domain. Section 3 covers related work, before section 4 introduces ν^*-nets as formal foundation for describing conversations including correlation. Section 5 defines instance isolation and shows how to check it. Section 6 will discuss the results of this paper, before section 7 concludes and gives an outlook to future work.

2 Example

This section will introduce a simple example that will be used throughout this paper. Imagine interacting organizations in a procurement setting: Buyers place orders at sellers who in turn respond with an order acknowledgement. As a third message exchange in the conversation, the buyer sends payment to the seller. This might be done e.g. through the transfer of credit card information. For simplicity reasons we do not focus on exceptions in these conversations.

While individual conversations are always bilateral in our example, there are a number of buyers and sellers in the overall setting. All interaction is carried out through electronic message exchanges between the different organizations' information systems. Each information system has three ports for communicating with the outside world. There is one port for each message type: order, order acknowledgement and payment.

When looking into the internal structure of seller S's system, we see a set of process instance agents. The different process instances refer to the orders that are being processed concurrently. However, as the same

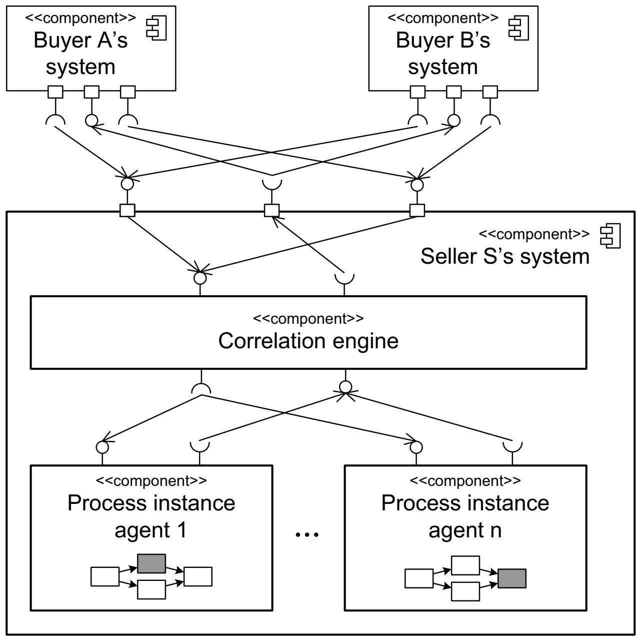

Figure 1. Architectural overview: Correlation engine and process instance agents

ports must be used for different process instances, a correlation engine is necessary as additional component. We modeled that all communication between the process instance agents and the outside world must go through the correlation engine. We introduced message send and receive interfaces for the communication between the correlation engine and the process instance agents. Upon arrival of a message from outside S's system, this message is forwarded to the correlation engine, which in turn checks which process instance agent is subscribed to it. Depending on this, it forwards it to the corresponding process instance agent. Upon arrival of a new order, a new process instance agent is created. Figure 1 depicts these components and their dependencies using the UML notation [1].

Although correlation mechanisms are offered in state-of-the-art information systems, the correlation configuration of implementations is sometimes not properly set. A possible outcome could be that a message is routed to the wrong process instance agent or that subscription is already forbidden in the first place. BPEL defines different exceptions occurring in the context of erroneous correlation configuration. If correlation sets are not properly initiated before being used, a *correlationViolation* exception is thrown. Furthermore, two process instances must not subscribe to the same set of messages. Here, a *conflictingReceive* exception applies. Finally, if an incoming message can be matched for two process instances with different subscriptions, an *ambiguousReceive* exception is thrown. Especially the latter two exceptions only occur when dealing with concurrent conversations.

3 Related Work

Correlation is a well-known concept especially in enterprise systems, where long-running conversations are a frequent phenomenon. Hohpe and Woolf documented a set of architectural patterns for such systems in [11] also including message correlation. Several web services standards introduce correlation identifiers as first-class citizens. BPEL [8] includes *correlation sets* and the Web Services Choreography Description Language (WS-CDL [13]) includes *identity tokens.* WS-Addressing [9] is an extension to the SOAP messaging format introducing a *replyTo* and *faultTo* field into the message header. An overview over recurrent correlation use cases can be found in [2]. The same paper uses these correlation patterns to assess BPEL and WS-CDL.

De Pauw et al. present a technique for mining conversations from message logs in [21]. As part of that, correlation identifiers are identified using heuristics.

The formal model used in this paper will extend a special class of Petri nets, namely ν-nets as presented in [23]. Here, each token carries a name. Name matching is applied upon synchronization, i.e. if a transition has more than one input place. A more complex class of Petri nets supporting the notion of correlation are Colored Petri nets [12] where tokens also carry values and complex guard conditions can be attached to transitions.

BPEL is the most important language for implementing services that engage in complex conversations. There has been quite some work on formal analysis for this language. Mappings to different formalisms are available, e.g. to abstract state machines [7], Petri nets [10] and a π-calculus based formalism [18]. Based on these formal representations compatibility checking between different BPEL processes [15] or consistency checking between a process implementation and a behavioral specification [16] can be carried out. However, these approaches largely ignore data aspects in general and correlation in particular. Moser et al. extended the formal mapping by also considering data flow issues [20]. However, to the best knowledge of the authors there is no work on analyzing correlation issues in the context of concurrent conversations.

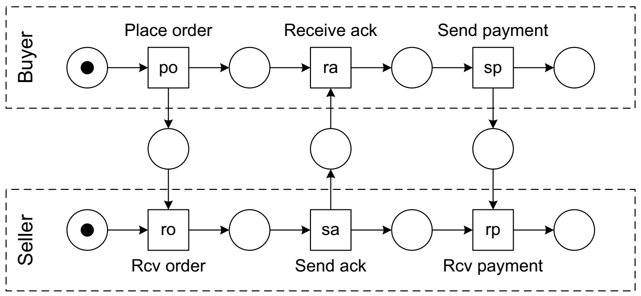

Figure 2. Classical place/transition net

4 Formal Model

Petri nets [22] are a widely used formalism for describing the behavior of distributed and parallel systems. Petri nets consist of *transitions*, *places* and *tokens*. Transitions typically represent activities while places can represent communication channels between different systems. Tokens are located at places and typically represent messages or control flow pointers. The combination of tokens in a Petri net is called the *marking* and represents the state of the overall system.

In classical place/transition nets, two tokens located at the same place cannot be distinguished from each other. Therefore, two messages represented by tokens residing on the same place cannot be distinguished from each other.

Figure 2 describes the sample conversation from section 2. Send and receive activities are modeled using transitions (visualized as rectangles). The three communication channels for orders, order acknowledgements and payments are modeled as places (circles). Furthermore, internal places for the buyer and seller determine the behavioral dependencies between the send and receive activities. The two tokens (filled circles) are the initial marking. There is only one firing sequence allowed: 'Place order', 'Rcv order', 'Send ack', 'Rcv ack', 'Send payment', 'Rcv payment'.

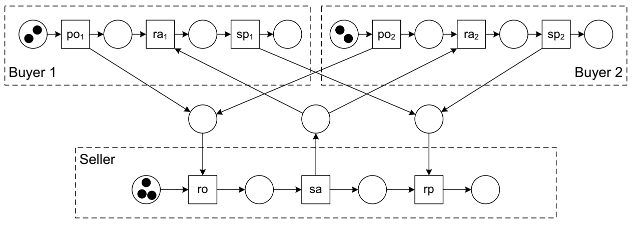

Figure 3. Concurrent conversations

Figure 3 shows a variation of the first Petri net. Now, we introduced a second buyer and more tokens. This enables concurrent conversations between the buyers and the seller. However, we cannot distinguish which buyer will get which acknowledgement. Such correlation information is not included in this representation.

4.1 ν^*-Nets

We introduce ν^*-nets for representing correlation information. They are an extension for place/transition nets with name creation and name passing. Name creation and passing has been extensively studied in process algebras such as π-calculus. However, there is a subtle difference between the usage of names in π-calculus and ν^*-nets: π-calculus works with name substitution. In contrast to this, we distinguish between names and *variables*. Tokens flowing through the net carry names, while the unmarked net is labeled with variables. Upon firing, names are assigned to variables. We denote the (infinite) set of names with Id and the set of variables with Var. Name creation leads to the introduction of fresh names into the marking. A special variable ν is reserved for this purpose ($\nu \in Var$).

The flow connections between places and transitions are labeled with vectors of variables that can have zero or more components. We introduce Var^* as the set of all variable vectors.

We are going to use the symbols $\in$ and $||$ for denoting the containment of a value in a vector and the length of a vector. E.g. $b \in (a, b, c)$ and $|(a, b, c)| = 3$.

1 (ν^*-net) *A ν^*-net N is a tuple $N = (P,\ T,\ F)$ where*

- *P and T are disjoint sets of places and transitions and*
- *$F : (P \times T) \cup (T \times P) \to Var^*$ is a partial function assigning variable vectors to flow connections between places and transitions.*

We introduce the auxiliary functions $pre, post : T \to Var$ for denoting input and output variables of a transition, where $pre(t) := \{v \in Var \mid \exists p \in P\ ((p,t) \in dom(F) \wedge v \in F(p,t))\}$ and $post(t) := \{v \in Var \mid \exists p \in P\ ((t,p) \in dom(F) \wedge v \in F(t,p))\}$. We also introduce $var(F)$ as the set of all variables in the net, i.e. $var(F) := \{v \in Var \mid \exists (o_1, o_2) \in dom(F)\ (v \in F(o_1, o_2))\}$. We assume all ν^*-nets to satisfy the following conditions:

- For every place p all variable vectors assigned to flow connections originating in p or targeting p have the same length, i.e. $\forall p \in P\ [\exists n\ (\forall (o_1, o_2) \in dom(F)\ [p \in \{o_1, o_2\} \Rightarrow |F(o_1, o_2)| = n])]$.

- ν does not occur in variable vectors assigned to flow connections targeting transitions, i.e. $\forall t \in T\ [\nu \notin pre(t)]$.

- All variables occurring in variable vectors assigned to flow connections originating in transitions occur in at least one of the variable vectors assigned to a flow connection targeting that transition, i.e. $\forall t \in T\ [post(t) \setminus \{\nu\} \subseteq pre(t)]$.

- No variable may be contained twice in the same variable vector.

Tokens flowing through ν^*-nets are colored. They represent vectors of names, where Id^* is the set of (potentially empty) name vectors.

2 (Marking) *Let (P, T, F) be a ν^*-net. The marking m of a ν^*-net assigns multi sets of name vectors to places, i.e. $m : P \to \mathcal{MS}(Id^*)$. We denote the set of names in a marking m as $S(m)$, where $S(m) := \bigcup_{p \in P}\{id \in ids \mid ids \in m(p)\}$. We denote the marked ν^*-net as (P, T, F, m).*

We assume that a marked ν^*-net satisfies the following condition: Each name vector contained in a marking has the same number of components as the variable vectors assigned to incoming and outgoing flow connections to that place.

We introduce *modes* as assignment of names to input and output variables of a transition, $\sigma : (pre(t) \cup post(t)) \to Id$. For every mode σ we introduce a corresponding vector mode $\sigma^* : Var^* \to Id^*$ assigning name vectors to variable vectors, where $\sigma^*((v_1, \ldots, v_n)) = (\sigma(v_1), \ldots, \sigma(v_n))$ and $\sigma^*(()) = ()$.

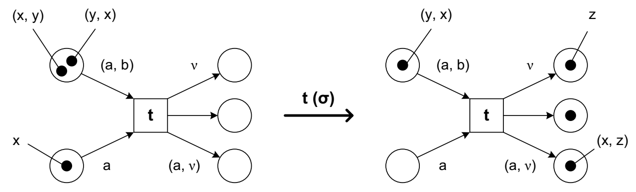

Figure 4. Sample ν^*-net and firing of t

3 (Enablement and Firing) *Let (P, T, F, m) be a marked ν^*-net. A transition $t \in T$ is enabled with mode σ if $\sigma(\nu) \notin S(m)$ and $\forall p \in P\ [(p, t) \in dom(F) \Rightarrow \sigma^*(F(p, t)) \in m(p)]$. The reached marking after firing of t is m', where $m'(p) := m(p) - \{\sigma^*(F(p, t))\} + \{\sigma^*(F(t, p))\}$. We denote this as $(P, T, F, m) \overset{t(\sigma)}{\to} (P, T, F, m')$ or $m \overset{t}{\to} m'$ as abbreviation.*

Figure 4 shows a marked ν^*-net on the left. Here, transition t is enabled with mode σ, where $\sigma(a) = x$, $\sigma(b) = y$ and $\sigma(\nu) = z$. Firing of t with mode σ leads to the marking depicted on the right. As Figure 4 shows, we omit inscriptions of flow connections and token descriptions in the diagram whenever the corresponding variable and name vectors have zero components.

Two Petri nets, as every transition system, can be compared by simulation and bisimulation techniques. If one marked Petri net P_1 simulates another marked Petri net P_2, P_1 can simulate all transitions fired in P_2. In the case of bisimulation both nets can simulate each other. They might have very different structure, but they have equivalent firing behavior.

It is easy to see that a marked ν^*-net (P, T, F, m) can be simulated by the marked place/transition net (P, T, F', m'), where $F' = dom(F)$ and $m'(p) = |m(p)|$ for all $p \in P$. Here, $|m(p)|$ is the number of vectors residing on place p.

4.2 Interconnection Models

The Petri net representations of the example have shown that we actually depict interconnected process models, where each process model belongs to a role. The two roles involved in the example are *buyer* and *seller*. R denotes the set of roles.

4 (Interconnection Model) *An* interconnection model *is a tuple (P, T, F, m, r) where*

- *(P, T, F, m) is a marked ν^*-net and*

- *$r : P \cup T \to R$ is a partial function assigning a role to places and transitions, where $T \subseteq dom(r)$.*

All places $p \in P$ having a role assigned are called internal places *and all other places in P are called* communication places. *An internal place p must only be connected to transitions assigned to the same role. Each transition has at most one communication place as input place and at most one communication place as output place.*

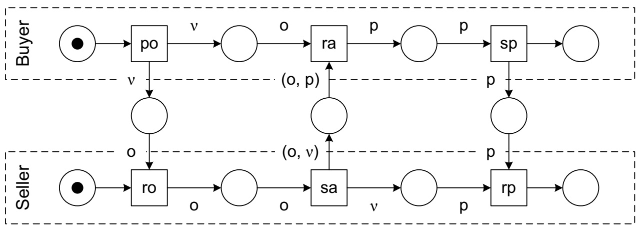

Figure 5. Interconnection model N

Figure 5 depicts the interconnection model for the example of section 2. The dashed lines represent the assignment of places and transitions to roles. Two correlation identifiers were used in the model. A fresh name is created upon firing of transition *po*. Note that the two tokens produced on the output places of *po* will carry the same name. In the following, variable o is used to pass this name through the net. Although such a consistent naming of variables in a ν^*-net is not mandatory, it increases the readability of the model. The most important use of variables o is at transition *ra*: Here, both input tokens have to carry the same name. I.e. synchronization only takes place if the message sent by the seller is awaited for by the buyer.

Another interesting situation already occurs at transition *ro*. While the token produced by *po* carries a name, the second token consumed does not carry any name. The seller accepts any message from the buyer and passes on the received name for later use.

A second fresh name is created upon firing of transition *sa*. For properly correlating the different messages in the conversation the creation of one fresh name would have been enough. However, using different correlation identifiers is quite common in real-world settings. While the first part of the conversation makes use of an order identifier o, a switch towards using a payment identifier p is made during the conversation. In addition to different phases as reason for different correlation identifiers, the different participants typically want to define their own correlation identifier.

5 Instance Isolation

This section is going to introduce the notion of instance isolation for interconnection models. Interconnection models typically represent (all possible paths of) exactly one conversation. As a participant of a particular role might be involved in several conversations at the same time, it must be checked whether the conversations are isolated from each other. We want to ensure that one process instance within a participant's system corresponds to at most one conversation. In the remainder of this section we will present how it this can be checked.

A process instance is involved in at most one conversation if and only if it never competes for the same message with another process instance. By competition (or conflict) we mean that two different process instances are able to consume exactly the same message. This can be checked by analyzing two concurrent conversations. If two process instances do not compete for the same message in this scenario, we can conclude that process instances are always pairwise isolated.

A process instance joins a conversation as *initiator* or *follower*. As initiator it sends a message prior to receiving a message. As follower it is the other way round. In the case of a follower, it often occurs that the message received can be consumed by any process instance. I.e. in this particular case, it should be allowed that different process instances compete for the same message. However, once a message was sent or received, there should be no competition for messages with other process instances any longer. This must be considered in the instance isolation analysis.

Figure 5 from the previous section illustrates the distinction between initiators and followers. Here, the seller acts as follower and the incoming purchase order might be processed by any process instance. More details on initiators and followers can be found in [2], where a set of common correlation scenarios are listed.

For instance isolation analysis, we are going to represent two concurrent conversations by duplicating all transitions and internal places. Figure 6 illustrates this for our example. At the heart of the analysis, we need to find those situations where two receive transitions belonging to the same role are enabled for the same message in different conversations.

The basic idea is to use reachability analysis for detecting these situations. We introduce "competition transitions" that are enabled whenever two competing receive transitions are enabled for the same token on a communication place. In case such a competition transition is reachable, we know that the two conversations are not isolated from each other.

In order to cater for those situations where competition is allowed, competition transitions must only be reached after a message was sent or received. This is realized by the introduction of an additional input place for all competition places. Token are produced onto this place by all message send and receive activities.

Figure 7 shows the previously mentioned additional places and competition transitions. Figure 8 shows the algorithm for generating the duplication model which will serve as input for the instance isolation analysis.

In line 2 of the algorithm, all communication places ($P \setminus dom(r)$) from the original model are reused in the duplicated model. Lines 5 to 15 copy all internal places and all transitions twice into the duplicated model. This includes assigning roles to places and transitions and copying the marking. Lines 17 to 19 take care of the additional places introduced per role. The flow connections between send / receive transitions and these places are added in lines 20-22. Lines 23 to 26 finally introduce the competition transitions for pairs of receive transitions. Special care needs to be taken for setting

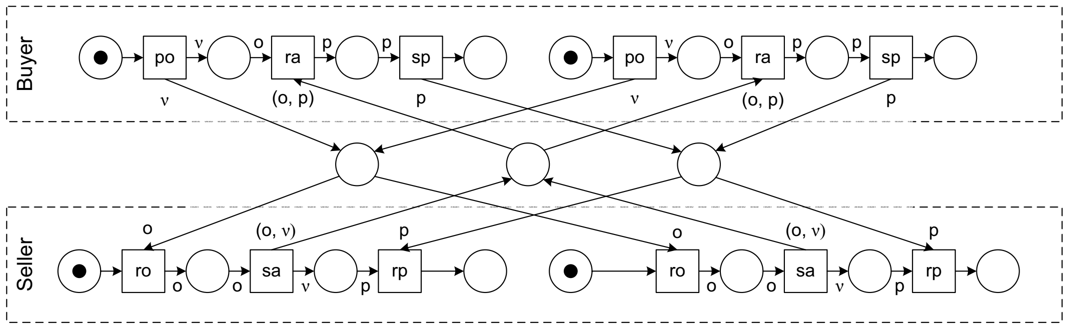

Figure 6. Representing two concurrent conversations

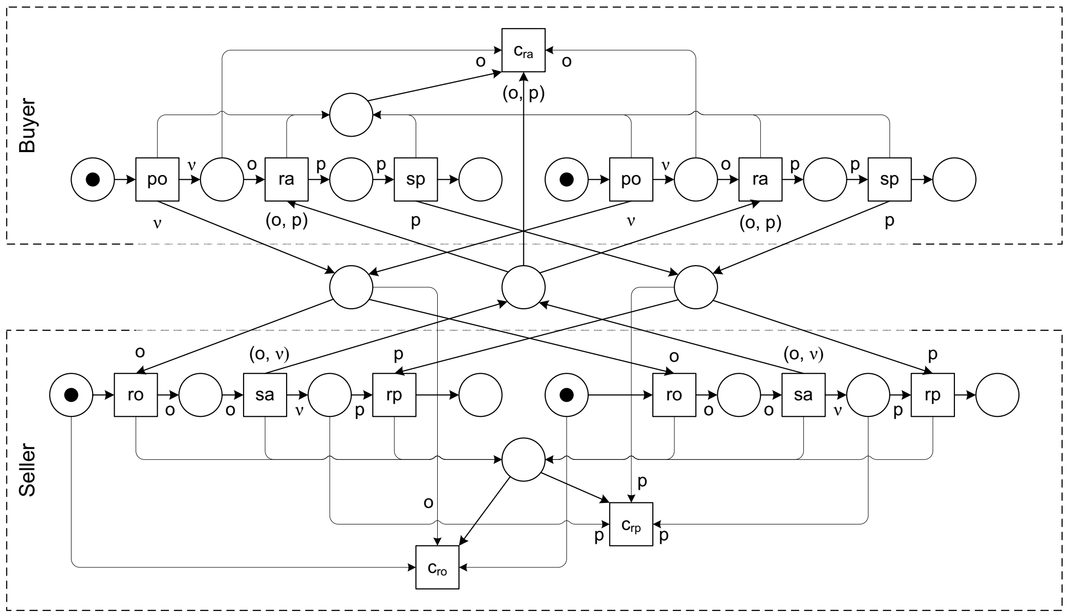

Figure 7. Addition of competition transitions

up proper flow connections. A connection from the additional place to the transition is added (line 26) and the connections targeting t_1 are simply copied (line 27). Renaming of variables might be necessary for copies of the connections targeting t_2 (lines 28-30). Renaming is necessary for avoiding additional synchronization constraints incurred by corresponding variables. The auxiliary function $map_v : Var \times P_D \times T_D \times T_D \rightarrow Var$ is introduced for this purpose. It is defined as follows: $map_v(v, p, t_1, t_2) = v'$ if v' is a component of $F_D(p, t_1)$ and v is a component of $F_D(p, t_2)$ at the same position, i.e. $F_D(p, t_1) = (v'_1, \ldots, v'_n)$, $F_D(p, t_2) = (v_1, \ldots, v_n)$ and $v'_i = v' \wedge v_i = v$ for an i. Otherwise $map_v(v, p, t_1, t_2) = v''$, where v'' is a variable that is not contained in any of the variable vectors assigned to flow connections targeting t_1.

Once the duplicated model is created for an interconnection model with all competition transitions, instance isolation can be checked through reachability analysis.

5 (Instance Isolation) *Let $N = (P, T, F, m, r)$ be an interconnection model. N guarantees* instance isolation *if and only if none of the competition transitions are reachable in the duplicated model $D(N)$.*

6 Discussion

Missing correlation configuration is the typical reason for non-isolation of process instances. However, we need to distinguish between synchronous and asynchronous communication. Often, request/response interactions are carried out synchronously. In this case, there are correlation mechanisms in place in lower level of the protocol stack. E.g. a TCP connection is exclusively

```
1: function D $(P, T, F, m, r)$
2:   $P_D := P \setminus dom(r),\ r_D := \{(p, s) \in r \mid p \in P_D\}$
3:   $T_D := \emptyset,\ F_D := \emptyset,\ m_D := \emptyset$
4:   $T_1 := \emptyset,\ T_2 := \emptyset$
5:   for each $i \in \{1, 2\}$
6:     $\phi := \{(p, p) \mid p \in P \setminus dom(r)\}$
7:     for each $p \in dom(r)$
8:       $p' :=$ new(), $P_D := P_D \cup \{p'\}$
9:       $r_D(p') := r(p),\ m_D(p') := m_D(p)$
10:      $\phi(p') := p$
11:    for each $t \in T$
12:      $t' :=$ new(), $T_D := T_D \cup \{t'\}$
13:      $T_i := T_i \cup \{t'\},\ r_D(t') := r(t)$
14:      $\phi(t') := t$
15:    $F_D := F_D \cup \{(o_1, o_2, v) \mid (\phi(o_1), \phi(o_2), v) \in F\}$
16:  $map_r := \emptyset$
17:  for each $s \in range(r)$
18:    $p_r :=$ new(), $P_D := P_D \cup \{p_r\}$
19:    $r_D(p_r) := s,\ map_r(s) := p_r$
20:  $F_D := F_D \cup \{(t, p_r, ()) \mid$
21:    $\exists p \in P\ (\{(t, p), (p, t)\} \cap dom(F_D) \neq \emptyset) \wedge$
22:    $p_r = map_r(r_D(t))\}$
23:  for each $t_1, \in T_1, t_2 \in T_2, p \in P$ :
24:      $r_D(t_1) = r_D(t_2) \wedge \{(p, t_1), (p, t_2)\} \subseteq dom(F_D)$
25:    $t_c :=$ new(), $T_D := T_D \cup \{t_c\}$
26:    $F_D := F_D \cup \{(map_r(r_D(t_1)), t_c, ())\} \cup$
27:      $\{(p', t_c, F_D(p', t_1)) \mid (p', t_1) \in dom(F_D)\} \cup$
28:      $\{(p', t_c, (map_v(v_1, p, t_1, t_2), \ldots,$
29:        $map_v(v_n, p, t_1, t_2))) \mid p' \neq p \wedge$
30:        $(p', t_2, (v_1, \ldots, v_n)) \in F_D\}$
31:  return $(P_D, T_D, F_D, m_D, r_D)$
```

Figure 8. Algorithm for generating duplicated models with competition transitions

used for exchanging request and response. Synchronous invocations can be properly modeled by creating a fresh name and passing it along with the two messages. A discussion about how name creation and passing can be used to formally model such scenarios was already reported in [6], where π-calculus was employed.

One might argue that a human modeler can easily detect missing or erroneous correlation configuration in interconnection models and that hence automatic checking of this property is not needed. However, it must be considered that interconnection models representing real-world scenarios can easily contain hundreds of communication places, where manual model verification becomes error-prone.

The definition of instance isolation presented in this paper assumes that every process model is executed at most once in a conversation. However, there might be scenarios where the same process model is instantiated multiple times. Imagine a procurement scenario where a shipper has to be selected among a set of shippers. As part of that, prices and shipping conditions need to be requested from all shippers. In this scenario there would be one role representing all shippers and the technique presented in this paper would only check isolation of a set of process instances involved in one conversation of a set involved in another conversation.

In addition to checking instance isolation, interconnection models as presented in this paper could also be used to check other interesting properties of conversations. E.g. it could be derived whether “instance merging” is necessary. Imagine e.g. a stock broker where stock offers and stock requests can be registered and updated. During this initial phase there will be a process instance at the broker’s side handling the offer and one handling the request. Once the broker finds a match between an offer and a request, the two process instances are merged and dealing with this trade is handled in the same process instance from this point on. The overall interconnection model might include both phases. In this case there will be a transition in the model targeted by flow connections having disjoint variable lists attached. Such Instance merging imposes challenges on execution engines and tools monitoring proper interaction behavior of systems. However, a detailed discussion about instance merging would go beyond the scope of this paper and must be left to future work.

In terms of computational complexity, the proposed technique suffers the usual drawbacks of reachability analysis in the presence of concurrency. The duplication of conversations in our technique worsens this situation as it significantly increases the number of reachable markings. However, we were still able to check nets with hundreds of transitions within seconds using our reachability checker.

7 Conclusion

This paper has introduced the notion of instance isolation in service-oriented architectures. ν^*-nets were introduced as formal foundation. ν^*-nets extend ν-nets by allowing a token to carry multiple names. We have implemented the analysis functionality in a prototype.

The first language the proposed technique should be used for is BPEL. However, BPEL processes describe what we called roles in this paper. BPEL cannot be used for specifying full interconnection models. Therefore, we have proposed BPEL extensions, namely BPEL4Chor, for modeling such choreographies in previous work [5].

There is already a tool chain in place for generating place/transition nets from BPEL4Chor [14]. Therefore, including the mapping of correlation configuration into variable vectors is a next step in future work.

References

[1] UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.

[2] A. Barros, G. Decker, M. Dumas, and F. Weber. Correlation Patterns in Service-Oriented Architectures. In *FASE 2007*, Braga, Portugal, 2007.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, Mar 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[4] F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.

[5] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *ICWS 2007*, Salt Lake City, USA, July 2007.

[6] G. Decker, F. Puhlmann, and M. Weske. Formalizing Service Interactions. In *BPM 2006*, Vienna, Austria, Sept 2006. Springer LNCS.

[7] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In D. Beauquier, E. Brger, and A. Slissenko, editors, *ASM 2005*, pages 131–151. Paris XII, Mar. 2005.

[8] D. C. Fallside and P. Walmsley. Web Services Business Process Execution Language Version 2.0. Technical report, Oct 2005. `http://www.oasis-open.org/apps/org/workgroup/wsbpel/`.

[9] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core, W3C Recommendation. Technical report, May 2006. `http://www.w3.org/TR/ws-addr-core/`.

[10] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *BPM 2005*, volume 3649 of *LNCS*, pages 220–235, Nancy, France, Sept. 2005. Springer.

[11] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[12] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer, 1996.

[13] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation. Technical report, November 2005. `http://www.w3.org/TR/ws-cdl-10`.

[14] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. In *WSFM 2007*, Brisbane, Australia, September 2007.

[15] A. Martens. Analyzing Web Service based Business Processes. In *FASE 2005*, volume 3442 of *LNCS*, Edinburgh, Scotland, April 2005. Springer-Verlag.

[16] A. Martens. Consistency between Executable and Abstract Processes. In *EEE 2005*, pages 60–67, Hong Kong, China, 2005. IEEE Computer Society.

[17] P. Mayer and D. Lübke. Towards a BPEL unit testing framework. In *TAV-WEB 2006*, Portland, Maine, USA, July 2006.

[18] M. Mazzara and R. Lucchi. pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70:96–118, 2006.

[19] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100:1–40, 1992.

[20] S. Moser, A. Martens, K. Gorlach, W. Amme, and A. Godlinski. Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. In *SCC 2007*, pages 98–105, Salt Lake City, Utah, USA, 2007. IEEE Computer Society.

[21] W. D. Pauw, R. Hoch, and Y. Huang. Discovering conversations in web services using semantic correlation analysis. In *ICWS 2007*, pages 639–646, Salt Lake City, Utah, USA, July 2007. IEEE Computer Society.

[22] W. Reisig. *Petri nets*. Springer Verlag, 1985.

[23] F. R. Velardo and D. de Frutos-Escrig. Name creation vs. replication in petri net systems. In *Petri Nets 2007*, volume 4546 of *LNCS*, pages 402–422, Siedlce, Poland, June 2007. Springer.