Synthesis of Orchestrators from Service Choreographies

Stephen McIlvenna[1], Marlon Dumas[2,1], Moe Thandar Wynn[1]

[1]Faculty of Science and Technology, Queensland University of Technology
GPO Box 2434, Brisbane, Queensland 4001, Australia
{s.mcilvenna, m.wynn}@qut.edu.au

[2]Institute of Computer Science, University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract

Interaction topologies in service-oriented systems are usually classified into two styles: choreographies and orchestrations. In a choreography, services interact in a peer-to-peer manner and no service plays a privileged role. In contrast, interactions in an orchestration occur between one particular service, the orchestrator, and a number of subordinated services. Each of these topologies has its trade-offs. This paper considers the problem of migrating a service-oriented system from a choreography style to an orchestration style. Specifically, the paper presents a tool chain for synthesising orchestrators from choreographies. Choreographies are initially represented as communicating state machines. Based on this representation, an algorithm is presented that synthesises the behaviour of an orchestrator, which is also represented as a state machine. Concurrent regions are then identified in the synthesised state machine to obtain a more compact representation in the form of a Petri net. Finally, it is shown how the resulting Petri nets can be transformed into notations supported by commercial tools, such as the Business Process Modelling Notation (BPMN).

Keywords: service composition, choreography, orchestration, Petri nets, BPMN.

1 Introduction

A Service-Oriented Architecture (SOA) is a software architecture where the basic elements are services, meaning entities offer some functionality to other entities, which themselves can be services. At the implementation level, an SOA manifests itself in the form of a collection of software services that exist at certain endpoints and exchange messages according to certain contracts. A software service is called a Web Service (WS) if it applies Web standards such as eXtensible Markup Language (XML), Web Service Description Language (WSDL), and/or SOAP.

A typical approach to design an SOA is to identify basic services and then to compose them into larger services, or conversely, to identify larger services and to then decompose them into smaller services. In either case, the cornerstone for SOA design is the definition of compositions of services. This work is concerned with how these compositions of services are modelled, and specifically, how different perspectives for modelling such compositions of services can be reconciled.

Depending on their interaction topology, service compositions are usually classified into two styles: choreographies and orchestrations (Peltz 2003). In a choreography, no service plays a privileged role (peer-to-peer topology), whereas in an orchestration, interactions occur between one particular service, the orchestrator, and a number of other subordinated services (hub-and-spoke topology). For example, Figure 1 illustrates a business-to-business choreography involving a buyer, a supplier and a shipper, while Figure 2 shows the corresponding orchestration.

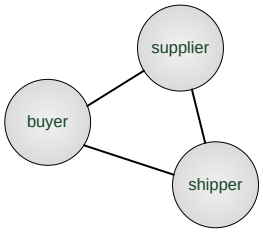

Figure 1: Choreographed composition

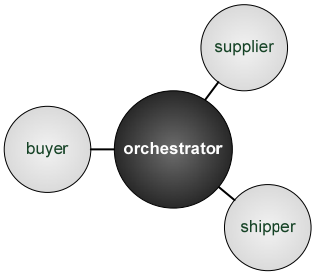

Figure 2: Orchestrated composition

The choice between a choreographed and an orchestrated service composition approach may be driven by a number of factors, often of an organisational nature. With reference to the above example, it may be that initially, the supplier deals with a single shipper for all orders. The buyer interacts with the supplier in order to agree on the purchase order and to pay for the goods, but when it comes to delivery issues, the buyer needs to interact directly with the shipper. Subsequently, the supplier may decide that in order to provide a more uniform customer experience and to closely monitor the performance of its shippers, it is desirable to have a single

point of interaction with the customer (the 'orchestrator' in Figure 2). This orchestrator would handle interactions related to the purchase order, payment and also delivery. Having done that, it becomes possible to introduce additional value-added services in the orchestrator, such as providing the buyer with the possibility to choose between making a single payment for the goods and for the delivery, or making separate payments, or choosing between different delivery modes.

From the technical perspective, this change in topology requires that an orchestrator is developed and deployed and that all the interactions between the services in the composition be channelled through the orchestrator. This paper provides a technique to automate the development of such an orchestrator from a given choreography.

Choreographies are initially represented using Finite State Machines (FSMs). Based on this representation, an algorithm is presented that synthesises the behaviour of an orchestrator, which is also represented as a state machine. The synthesised state machine may be rather large and unreadable, because the interactions that an orchestrator needs to manage tend to occur in any order or in parallel, and this parallelism leads to state explosion.

Accordingly, concurrent regions are identified in the synthesised state machine in order to obtain a more compact representation in the form of a Petri net. This is achieved using existing results from the field of theory of regions (Cortadella, Kishinevsky et al. 1998). The paper then shows how the resulting Petri nets can be transformed into notations supported by commercial tools, such as the Business Process Modelling Notation (BPMN) (Object Management Group 2008). The result is a skeleton of an orchestrator. This skeleton can be extended and refined using existing business process modelling tools and used as a basis to generate code in executable languages such as the Business Process Execution Language (BPEL).

The entire tool chain for orchestrator synthesis is depicted in Figure 3. The specific contributions of the paper are: (i) a technique for synthesising orchestrators from choreography specifications using state machines as a specification language, and (ii) a technique for transforming Petri nets into BPMN diagrams. The tool chain, starting from a choreography specified as FSMs, has been implemented and tested on a number of scenarios with varying degrees of complexity.

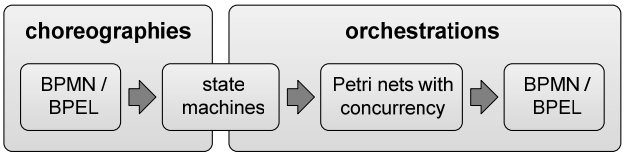

Figure 3: Composition viewpoints bridged by orchestrator synthesis with state machines

The rest of the paper is structured as follows. Section 2 provides some background on service behaviour modelling. Section 3 presents the proposed orchestrator synthesis algorithm. Section 4 shows how Petri nets representing the resulting orchestrator can be transformed into BPMN diagrams. Section 5 describes the validation of the approach and Section 6 discusses related work. Conclusions are drawn in Section 7.

2 Background: Modelling service behaviour

Choreographies can be described as a set of *interface FSMs*, where an interface FSM defines both the message exchanges in which a given participant can engage, and their message control-flow dependencies.

In an environment where messages can be buffered and transmission is not instantaneous, unbounded queues can be problematic for compatibility verification (Bultan, Su et al. 2006). Reasoning with protocols can be simplified by either bounding the queue length (Berardi, Calvanese et al. 2005), or removing queues entirely (Benatallah, Casati et al. 2006).

An assumption sometimes taken (Yellin and Strom 1997; Benatallah, Casati et al. 2006) is an environment where message transmission is instantaneous, meaning the FSMs of the sender and receiver for any given interaction advance in synchrony. While this assumption is not in line with some communication protocols which support asynchronous message transfer, it appears solutions developed under the assumption of synchronous messaging may be transposable to asynchronous environments as referred to in Section 6. Thus, we make this assumption of synchronicity to simplify orchestrator synthesis.

Having made this assumption about the communication medium, we also need to adopt a language for capturing service behaviour. Languages such as BPMN and BPEL could be used to for this purpose. However, these languages are complex in terms of the number of constructs they support, hindering their suitability as a basis for reasoning about service behaviour. Also, these languages are meant for capturing service-oriented business processes rather than capturing the behaviour that one service exposes to other services. For example, both languages allow one to capture internal actions and decisions that a service-oriented process makes during its execution. However, when capturing service behaviour for orchestrator synthesis purposes, we are only interested in capturing the externally visible behaviour that each service exposes.

In light of this, we adopt FSMs as the language for capturing service behaviour. This choice is in line with previous work on component and service behaviour specification (Yellin and Strom 1997; Benatallah, Casati et al. 2006; Berardi, Calvanese et al. 2005). Accordingly, a choreography is captured as a collection of communicating state machines. This design choice is further justified in Section 6.

Specifically, we rely on the notion of *interface FSM*, which is essentially an FSM where the transitions are labelled with communication actions – either sending or receiving a message. To ensure protocols describe only external behaviour, the FSMs we deal with are deterministic, meaning that every state is labelled, and for any given state there are no two outgoing transitions with exactly the same label. In order to deterministically model choices based on message content, we use Boolean guards expressed in terms of message content. For example to capture the requirement that depending on the content of a message of type OrderResponse, the FSM

should follow one transition or another, we append expressions like [processed=true] and [processed=false] to the message type. Hence, one transition could be labelled with 'OrderResponse[processed=true]' and another with 'OrderResponse[processed=false]'.

The orchestrators synthesised by the algorithm presented later in the paper, typically perform message forwarding, and we found that representing a message forwarding action as separate receive and send transitions leads to cumbersome models. To simplify the specification of orchestrators, message exchanges are represented as a quadruplet Exchange = (p_{from} : Party, p_{to} : Party, msg : MessageType, forwarding : Boolean). This tuple specifies the initial sender of the message, the final recipient of the message, the type of the message, and whether the message is directly exchanged between the two parties or it is received from one party (by the orchestrator) and forwarded to the other.

Formally, an interface FSM is a tuple (S, s_0, E, δ) where:

- S is a set of states. A state is labelled with an identifier in the case of an elementary state, or a tuple, possibly with other nested tuples, in the case of a composite state derived during the merging of two or more other interface FSMs.
- $s_0 \in S$ is the initial state.
- E is a set of message exchanges specified as quadruplets, as previously discussed.
- $\delta : S \times E \rightarrow S$ is a transition function to connect states via message exchanges.

A *behavioural interface* is defined as a combination of an interface FSM and the set of parties the FSM represents, the pair (P : {Party}, sm : FSM). Normally, the set of parties P of a behavioural interface will contain only one element, because a behavioural interface represents the behaviour that one party exposes to one or several parties. But in the case of an orchestrator service, the behavioural interface represents the aggregated behaviour of multiple subordinated services and P will contain multiple parties. For convenience, we will use the term *orchestrator interface*, as shorthand to refer to the behavioural interface of an orchestrator service.

Behavioural interfaces of the choreography participants in our working example are shown in Figures 4 to 6.. These interfaces are based on the Voluntary Inter-industry Commerce Standard (VICS) for order management (GS1 US 2007).

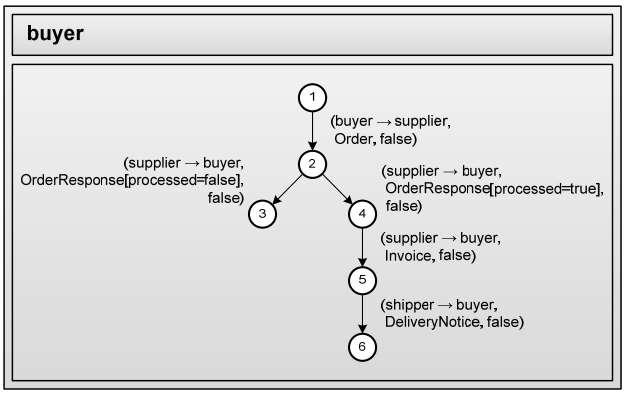

Figure 4: Behavioural interface of the buyer

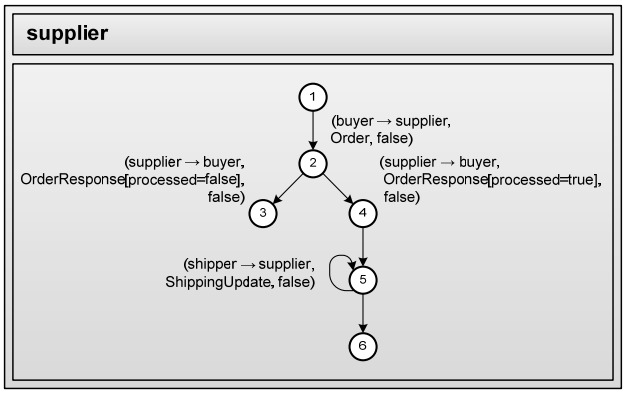

Figure 5: Behavioural interface of the supplier

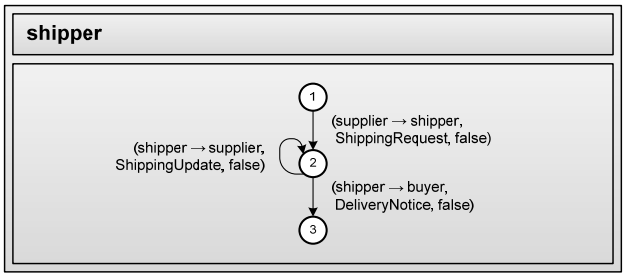

Figure 6: Behavioural interface of the shipper

3 Orchestrator synthesis

The goal of orchestrator synthesis is to generate a behaviourally compatible message forwarding service capable of intercepting messages within a given choreography. In this paper, we propose a synthesis algorithm which merges interface FSMs from any number of parties in a choreography to produce an orchestrator. The algorithm comprises three main functions and makes use of the following auxiliary functions:

- For any ordered list L, enqueue(L, n) adds n to the end of L, and dequeue(L) removes the first element from the front of L. If n is a list, each element is added in order to the end of L.
- For a message exchange e, fromParty(e), toParty(e), msg(e), forwarding(e), retrieve the corresponding components.
- For a state machine sm, states(sm) returns all states, initialState(sm) the initial state, and finalStates(sm) the set of final states, which is derivable by finding all states having no outgoing transitions.
- For a composite state c, s1(c) and s2(c) return the two contained states.
- For a transition t, exchange(t) returns the message exchange, and source(t) and target(t) the source and target states respectively.

For better logic clarity, δ is also characterised as a set of Transition objects, each composed of a message exchange, one source and one target state. Also, unordered sets are denoted by {} and ordered lists by [].

3.1 Synthesis of multiple interfaces

Our algorithm is capable of synthesising any number of behavioural interfaces forming a choreography into a single orchestrator interface. The approach used is to fully merge two of the input interfaces, then merge the result with a third interface, and so on in pairs, until all input interfaces have been synthesised into the orchestrator. This high level processing of taking all input interfaces and synthesising the orchestrator is performed by Function 1, synthesise(), and is visualised in Figure 7. If a deadlock condition is detected while synthesising any interface pair, synthesise() immediately indicates synthesis is not possible.

```
Function: synthesise
Input: I_all : [Interface]
Output: Interface ∪ Deadlock
Preconditions: |I_all| ≥ 2
Variables: synthesised : Interface ∪ Deadlock
begin
    synthesised := synthesiseInterfacePair(
                                dequeue(I_all), dequeue(I_all))
    if synthesised is Deadlock
        return synthesised
    end if
    while I_all ≠ []
        synthesised := synthesiseInterfacePair(
                                synthesised, dequeue(I_all))
        if synthesised is Deadlock
          return synthesised
        end if
    end while
    return synthesised
end
```

Function 1: synthesise()

Input interfaces must be ordered according to the role of each interface in the choreography. It is assumed only one interface FSM can send a message from the initial state – the others start their execution by receiving a message. In the case of the choreography involving a buyer, a supplier and a shipper, the buyer would start the choreography. The two interfaces exchanging the first message must be the first two in the input list to ensure the synthesised orchestrator interface captures the choreography from start to end. Subsequent interfaces in the list must appear in the order they come into the choreography. The effect of this ordering on the working example is shown in Figure 7.

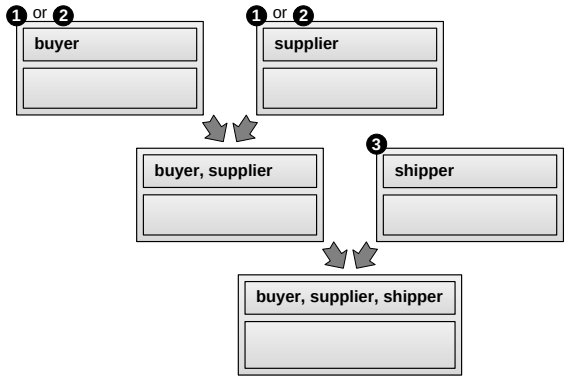

Figure 7: Synthesis of interface pairs showing the ordering of the input list

3.2 Synthesis of an interface pair

A pair of interfaces can be completely merged by Function 2, synthesiseInterfacePair(), depicted in Figure 8. This function synchronously traverses the two input interfaces to build the orchestrator interface by performing a breadth-first search of the two input FSMs, and combining the lists of parties represented by both interfaces. The input FSMs are searched in synchrony by looking at state pairs where a state pair is composed of a state from each FSM. Merging a pair of interfaces is realised with a visitor pattern (Palsberg and Jay 1998) to perform the breadth-first search.

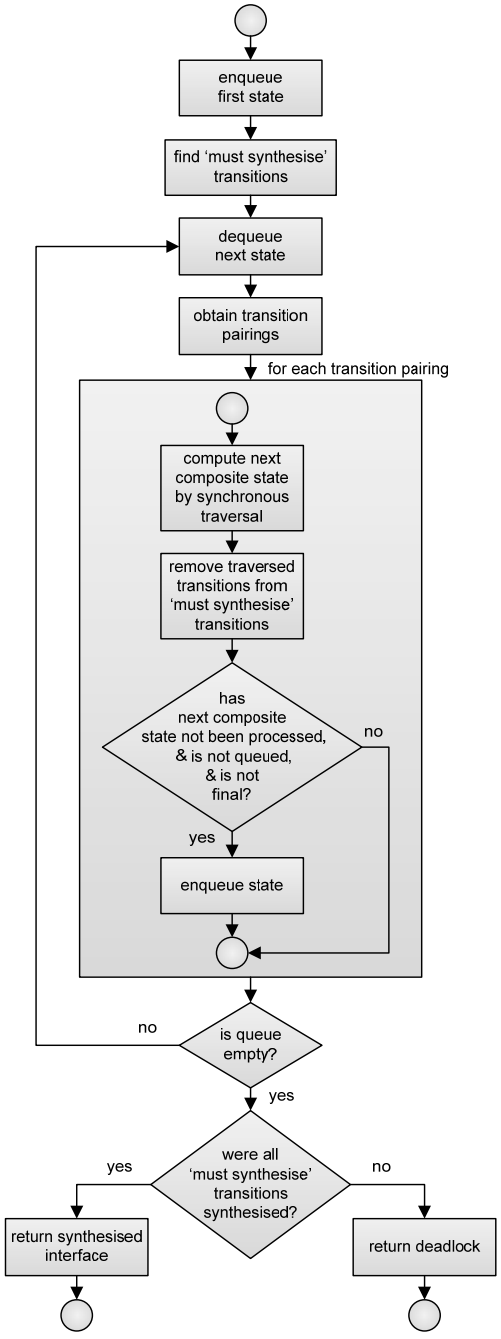

Figure 8: Overview of synthesiseInterfacePair()

The root node for performing a breadth first search is the composite state derived from the two initial states of input FSMs. This state is the initial state of the synthesised FSM, and is placed in a queue of states to be processed. Only composite states are placed in this queue,

and each time one is dealt with, it is placed into a pool of states which have already been processed. Each state is visited only once. Synthesis of the input pair is complete when all states have been visited and the queue of states to visit is empty.

After the initial state has been removed from the queue, another function attempts to find pairs of message exchanges that can occur between the two input FSMs. A match is apparent when both FSMs can synchronously exchange a message and advance to their next respective states.

As seen in Function 3, transitionPairings() explores which messages can be sent or received from the states currently being processed from each input FSM, and attempts to find matches. If a match is found, the pair of transitions is added as a tuple to a set of pairings. If a match for a transition is not found, it is added to a tuple with the other element empty, representing that the interaction cannot yet be orchestrated, which is a normal situation if the party with which this transition should be paired has not yet been dealt with. A tuple indicating that a transition could not be matched is only added to the set of pairings if the exchange is orchestrated, or is related to an interface not yet considered for orchestrator synthesis. Otherwise, the inability to match a given transition with at least one other transition means that one party can send a message while the other is not in a state to receive it, or vice-versa. In some cases, if there is no other way to progress to a new pair of states from the current pair of states, this situation indicates a deadlock.

Function: synthesiseInterfacePair
Input: $i_a = (P_a, sm_a = \{S_a, s_{0\_a}, E_a, \delta_a\})$: Interface, $i_b = (P_b, sm_b = \{S_b, s_{0\_b}, E_b, \delta_b\})$: Interface
Output: Interface ∪ Deadlock
Preconditions: ∀s ∈ S_b , s is ElementaryState
Variables: s_{0_m} , $s_{current}$, s_{new} , s_{toAdd} : CompositeState
$S_{toVisit}$: [CompositeState], $S_{visited}$: {CompositeState}
$sm_m = \{S_m, s_{0\_m}, E_m, \delta_m\}$: FSM
$t_{a\_mustSynthesise}$: {Transition}, $t_{b\_mustSynthesise}$: {Transition}
e_{new} : Exchange
TP_{all} : {(Transition, Transition)}
begin
$s_{0\_m} := (s_{0\_a}, s_{0\_b})$
$states(sm_m) := \{s_{0\_m}\}$
$enqueue(S_{toVisit}, s_{0\_m})$
$P_{known} := P_a \cup P_b$
$t_{a\_mustSynthesise} := \{t \in \delta_a ; e \in exchange(t) \mid fromParty(e) \in P_{known} ; toParty(e) \in P_{known} ; \neg forwarding(e) \bullet t\}$
$t_{b\_mustSynthesise} := \{t \in \delta_b ; e \in exchange(t) \mid fromParty(e) \in P_{known} ; toParty(e) \in P_{known} ; \neg forwarding(e) \bullet t\}$
while $S_{toVisit} \neq \{\}$
$s_{current} := dequeue(S_{toVisit})$
$S_{visited} := S_{visited} \cup \{s_{current}\}$
$TP_{all} = transitionPairings(i_a, s1(s_{current}), i_b, s2(s_{current}))$
for each (t_a, t_b) in TP_{all}
if $t_a \neq NULL \wedge t_b \neq NULL$
$s_{new} := (target(t_a), target(t_b))$
$e_{new} := (fromParty(exchange(t_a)), toParty(exchange(t_a)), msg(exchange(t_a)), true)$
else if $t_a = NULL$
$s_{new} := (s1(s_{current}), target(t_b))$
$e_{new} := exchange(t_b)$
else
$s_{new} := (target(t_a), s2(s_{current}))$
$e_{new} := exchange(t_a)$
end if
$states(sm_m) := states(sm_m) \cup \{s_{new}\}$
$exchanges(sm_m) := exchanges(sm_m) \cup \{e_{new}\}$
$t_m := t_m \cup \{(s_{current}, e_{new}) \rightarrow s_{new}\}$
$t_{a\_mustSynthesise} := t_{a\_mustSynthesise} \setminus \{t_a\}$
$t_{b\_mustSynthesise} := t_{b\_mustSynthesise} \setminus \{t_b\}$
if $s_{new} \notin S_{visited} \wedge s_{new} \notin S_{toVisit} \wedge \neg(s1(s_{new}) \in finalStates(sm_a) \wedge s2(s_{new}) \in finalStates(sm_b))$
$enqueue(S_{toVisit}, s_{new})$
end if
end for
end while
if $t_{a\_mustSynthesise} \neq \{\} \vee t_{b\_mustSynthesise} \neq \{\}$
return Deadlock
end if
return $Interface(P_a \cup P_b, sm_m)$
end

Function 2: synthesiseInterfacePair()

Function: transitionPairings
Input: $i_a = (P_a, sm_a = (S_a, s_{0\_a}, E_a, \delta_a))$: Interface,
s_a : State, $i_b = (P_b, sm_b = (S_b, s_{0\_b}, E_b, \delta_b))$: Interface,
s_b : ElementaryState
Output: {(Transition, Transition)}
Preconditions: $s_a \in S_a$; $s_b \in S_b$
Variables: T_{pairs} : {(Transition, Transition)}
T_{out_a} , T_{out_b} , T_{done_a} , T_{done_b} : {Transition}
P_{known} : {Party}
begin
  $T_{out\_a} \coloneqq \{t \in T_a \mid source(t) = s_a\}$
  $T_{out\_b} \coloneqq \{t \in T_b \mid source(t) = s_b\}$
  for each t_a in T_{out_a}
    for each t_b in T_{out_b}
      if $exchange(t_a) = exchange(t_b)$
        $T_{pairs} \coloneqq T_{pairs} \cup \{(t_a, t_b)\}$
        $T_{done\_a} \coloneqq T_{done\_a} \cup \{t_a\}$
        $T_{done\_b} \coloneqq T_{done\_b} \cup \{t_b\}$
      end if
    end for
  end for
  $P_{known} \coloneqq P_a \cup P_b$
  for each t_a in T_{out_a}
    if $t_a \notin T_{done\_a} \wedge (fromParty(exchange(t_a)) \notin P_{known} \vee toParty(exchange(t_a)) \notin P_{known}) \vee forwarding(exchange(t_a))$
      $T_{pairs} \coloneqq T_{pairs} \cup \{(t_a, NULL)\}$
    end if
  end for
  for each t_b in T_{out_b}
    if $t_b \notin T_{done\_b} \wedge (fromParty(exchange(t_b)) \notin P_{known} \vee toParty(exchange(t_b)) \notin P_{known}) \vee forwarding(exchange(t_b))$
      $T_{pairs} \coloneqq T_{pairs} \cup \{(NULL, t_b)\}$
    end if
  end for
  return T_{pairs}
end

Function 3: transitionPairings()

With respect to the working example, when merging the pair of behavioural interfaces (buyer, supplier), and when the pair of states being processed is (buyer 4, supplier 4) the message exchange (supplier → buyer, Invoice, false) is not added to the pairings as it is known this exchange cannot yet occur with the supplier, since the supplier first needs to send a ShippingRequest to the shipper. Therefore, for the state (buyer 4, supplier 4) the pairing function only returns the tuple (NULL, (supplier → shipper, ShippingRequest, false)) indicating one non-orchestrated interaction be added to the orchestrator interface.

If it occurs that the set of message exchange pairings is empty, then there are no messages that can be synchronously exchanged in the respective states of the composite state being processed. Therefore, deadlock is declared and orchestrator synthesis is terminated. The partially synthesised orchestrator could also be preserved if desired. If message exchange pairings are found, they are added to the orchestrator interface, such that matching exchange pairs are added as orchestrated exchanges, where forwarding=true, and others as non-orchestrated exchanges, where forwarding=false. Based on the result of pairing message exchanges, a new message transition is added to the synthesised FSM along with a composite state representative of the synchronous advancement performed. This new composite state is then queued for processing, but only if the state has not already been processed or queued for processing, and is not final. Synthesis of the two input FSMs is complete once both have been completely traversed.

With respect to the working example, the composite state (buyer 4, supplier 4) causes the pairing function to return (NULL, (supplier → shipper, ShippingRequest, false)). The non-orchestrated interaction is added to the orchestrator and the next state leading on from (buyer 4, supplier 4) is computed. This next state will involve the same buyer state (buyer 4) since the first element of the pairing tuple is NULL, and will involve the next supplier state (supplier 5). The state (buyer 4, supplier 5) is then added to the queue of states to visit.

The product of synthesising the buyer and supplier interfaces is shown in Figure 9, where each state (buyer x, supplier y) is shortened to (b_x, s_y). All interactions involving the buyer and supplier are orchestrated since these parties are fully represented in the orchestrator. Before the shipper is processed, it is not possible to know whether its interface FSM is compatible with that of the other services in the choreography, so interactions involving the shipper remain non-orchestrated.

Finally, synthesise() processes the shipper interface and the interface where P = {buyer, supplier} to produce the complete orchestrator P = {buyer, supplier, shipper}.

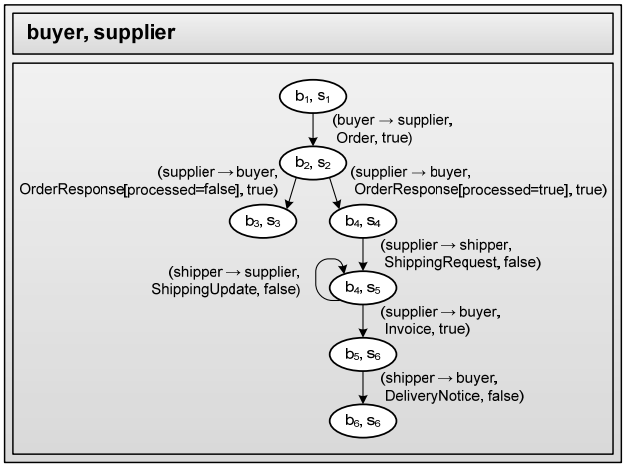

Figure 9: Result of merging the buyer and supplier interfaces

The synthesiseInterfacePair function performs a depth-first search over a graph whose nodes represent pairs of state – one state from each interface being synthesised. Each of these 'state pairs' is visited at most once. Similarly, every pair of transitions (one transition from each of the original state machines) is visited at most once. Thus the wost-complexity of the algorithm is $O(N_1*N_2 + E_1*E_2)$ where N_1 and N_2 are the number of states in each of the two interfaces, and E_1 and E_2 are the number of transitions.

3.3 Deadlock detection

The synthesis algorithm detects deadlock within synthesiseInterfacePair() by identifying non-synthesised transitions in both input interfaces which block synchronous advancement and cause any transition to not be added to the synthesis product. With respect to our working example, let's try to synthesise the interface pair from Figure 9 and Figure 10. The set of known parties for this synthesis is {buyer, supplier, shipper} meaning the synthesised product captures the combined behaviour of these three parties, and therefore all interactions between these three parties must be captured as orchestrated interactions in the synthesised interface.

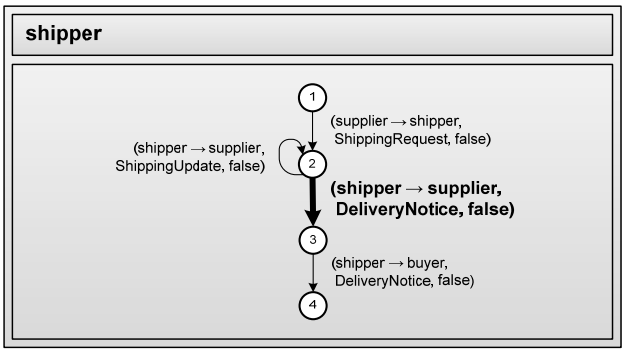

Figure 10: Shipper interface with an interaction causing deadlock

The synthesis algorithm proceeds normally until the shipper attempts to send a DeliveryNotice to the supplier (the interaction highlighted in Figure 10). At this stage, the supplier's behaviour is already fully captured in the interface with which the shipper's interface is being merged (the interface shown in Figure 9), and it is known that the supplier can never be in a state where it can receive a message of type DeliveryNotice. Prior to attempting synthesis of the interfaces in Figure 9 and Figure 10, this troublesome transition in the shipper's interface is earmarked as being a 'must synthesise' transition because it is non-orchestrated and involves only synthesised parties. Since the transition cannot be traversed during the synthesis, it remains marked as 'must synthesise', along with any subsequent, unreachable transitions and other untraversed transitions from the interface in Figure 9. Deadlock is therefore detected if any 'must synthesise' transition cannot be synthesised.

4 Orchestrator modelling

Synthesised orchestrators provide the grounding for migrating choreographies to orchestrations, which can then be augmented with additional functionality, such as lower-level multi-party message adaptation, or higher-level value-adding.

While state machines provide a practical bridge between the two service composition viewpoints, they are not very readable or maintainable due to state explosion, a drawback encountered where multiple messages may be exchanged in any order. In such a scenario, each possible sequence of message exchanges must be explicitly represented, leading to verbose FSMs, which although executable, are not easily maintainable.

Petri nets provide a graphical language capable of expressing concurrent interactions, and consist of place nodes, transition nodes, and directed arcs. They have been used for high level workflow management (van der Aalst 1998) and on a more detailed level for modelling process behaviour (Hinz, Schmidt et al. 2005).

4.1 State machines to Petri nets

We leverage the theory of regions (Cortadella, Kishinevsky et al. 1998) to identify regions of concurrency in FSMs, and replace these regions with their concurrent equivalents. The techniques derived from this theory are implemented in the Petrify tool (Cortadella, Kishinevsky et al. 1997). By reusing this tool, we can transform interface FSMs into free-choice Petri nets with concurrency.

A Petri net is said to be free-choice if and only if for every two transitions, if they share any input place, they share all input places (Chrzastowski-Wachtel, Benatallah et al. 2003). We enforce this restriction to prevent mixing of choice and synchronisation, which is difficult to represent in a higher level modelling language. Once orchestrators are represented as Petri nets, logic is easier to read, but the model clarity can be further improved for business users or developers if displayed in a recognised modelling language such as BPMN or BPEL. We identified value-adding augmentation as a potential use of orchestrators, so we developed a technique to translate Petri nets into BPMN diagrams, so value can be added at a business level by altering the logic in the diagrams.

4.2 Petri nets to BPMN

We developed a set of rules to transform Petri nets into BPMN diagrams. These rules are summarised in Figure 11, where each rule identifies a pattern in the graph to be replaced by the output specified in the rule. Dependencies between rules are introduced in order to reduce the complexity of pattern definitions. An overview of the rule patterns and outputs is presented in Figure 12, taking snippets from an extended version of our working example.

The rules are only concerned with the logic inside the BPMN pool artefact representing the orchestrator. Other pools and connecting message flows are not generated by the rules as they do not affect the control-flow logic.

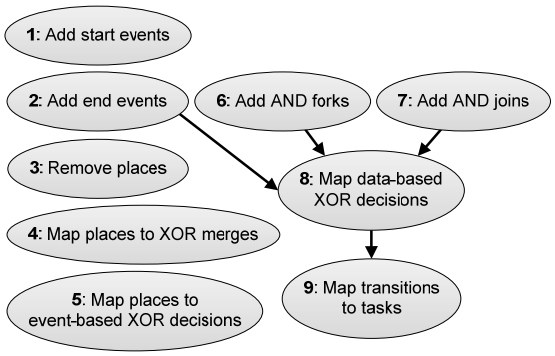

Figure 11: Rules and their dependencies

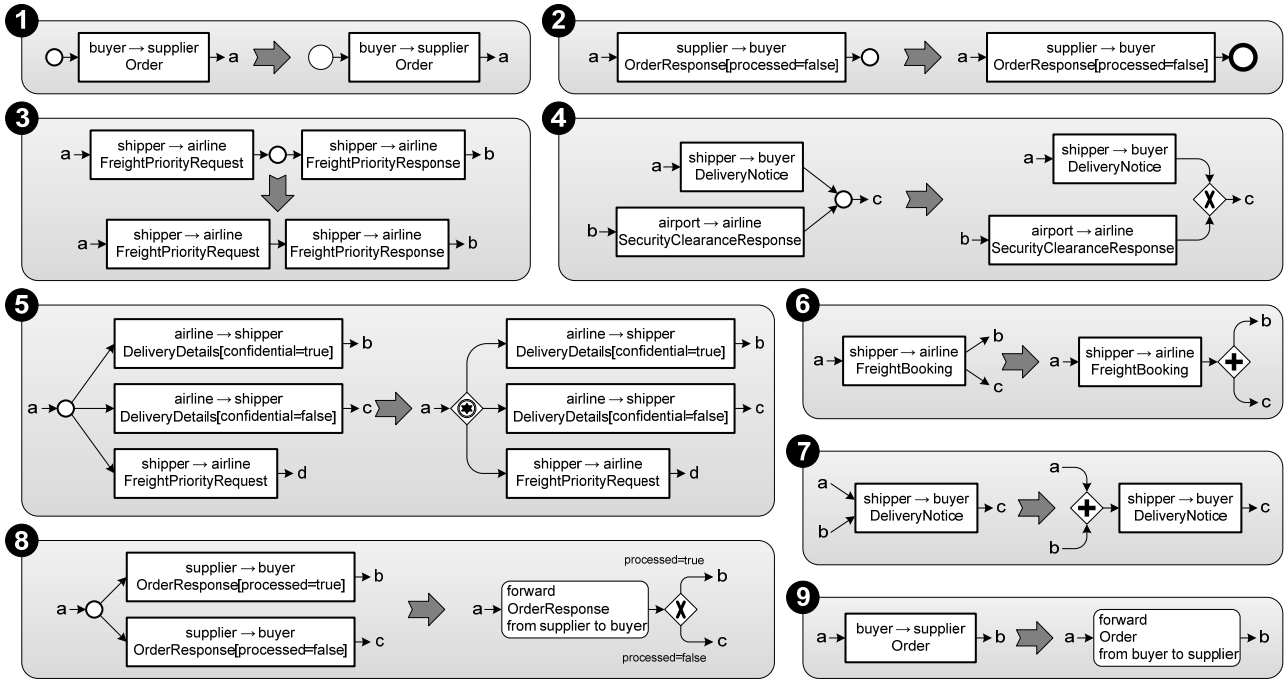

Figure 12: Petri net to BPMN transformation rules

We chose to model all message interactions with BPMN tasks without any BPMN message event elements to reduce the repetition that is prevalent with message forwarding, where a message is sent immediately after it is received. Tasks are therefore used for modelling message receiving, sending, and forwarding. The example cases in Figure 12 consider only orchestrated interactions which forward messages, but the rules are equally applicable to Petri nets describing interactions of individual parties, such as the buyer or the supplier. In such instances, tasks are for sending and receiving only.

5 Validation

We developed tooling to validate the synthesis algorithm and rule-based transformations. Utilising the Eclipse Modelling Framework (EMF) we created a graphical editor to design interface FSMs, which were used by an implementation of the algorithm to produce models readable in the same editor. We extracted around two dozen sample choreographies from two industry standards for business-to-business interactions, namely the XML Common business Library (xCBL) (xCBL.org 2000) and the Voluntary Inter-industry Commerce Standard (VICS) (GS1 US 2007). The examples had between two and four participants, arranged in different topologies and with varying numbers of states and transitions per interface FSM. By altering the FSMs of choreography participants, we also generated sample choreographies with deadlocks that the tool was able to detect.

We noticed state machine orchestrators involving four parties became very verbose, as pairs of services may interact independently, thereby confirming the value of representing orchestrators in a higher-level language such as BPMN.

The tool for transforming Petri nets into BPMN diagrams utilises ProM (van Dongen, de Medeiros et al. 2005), a framework for process mining and analysis, which provides a Petri net object model and an API for invoking the Petrify tool. BPMN diagrams are created through API and then imported into the BPMN modeller included in the SOA Tools Platform Project[1].

We tested an implementation of the transformation rules using the collection of choreographies mentioned above, and successfully automated the generation of correct BPMN diagrams for orchestrators, and for individual choreography participants.

6 Related work

The language for service behaviour modelling proposed in this paper is directly inspired from work in the area of behaviour specification for software components and services (Yellin and Strom 1997; Benatallah, Casati et al. 2006). Our assumption of synchronous communication is also inspired from this prior work. Although this assumption may be seen as inapplicable in some cases, it has been shown that under some conditions, it is possible to transpose results obtained under the synchronous communication assumption to an asynchronous communication medium (Yellin and Strom 1997; Bultan, Su et al. 2006).

Standard notations for specifying orchestrations and choreographies include BPMN and BPEL. BPMN is intended for modelling business processes involving human tasks and/or automated tasks. Automated tasks in BPMN are typically delegated to external services. A BPMN model that includes only service tasks is essentially an orchestration. BPMN has also been shown to be suitable for modelling choreographies (Decker and Barros 2008). BPEL on the other hand, is primarily intended to model orchestrations. However, it is also possible to use BPEL for specifying business protocols (an alternative term for designating behavioural interfaces), and extensions to BPEL have been proposed

[1] http://www.eclipse.org/stp/

to make it usable for capturing choreographies (Decker, Kopp et al. 2007).

Both BPMN and BPEL can be translated to Petri nets using existing techniques. A transformation from BPEL to state machines is implemented by the WS-Engineer toolset (Howard, Emmerich et al. 2007). We are not aware of direct transformations from BPMN to state machines, but transformations exist from BPMN to Petri nets (Dijkman, Dumas et al. 2008), which under certain assumptions can then be expanded into state machines.

A transformation from FSMs to BPEL has also been proposed (Zhao, Bryant et al. 2005), however this transformation does not attempt to identify concurrent regions in the FSM in order to obtain a simpler BPEL process definition. Instead, the generated BPEL process definitions are fully sequential (no 'parallel flow' activities). Thus, if the original FSM is complex due to concurrent message exchanges being represented as interleaved sequences of message exchanges, the resulting BPEL process definitions will mirror this complexity.

In this paper, we use techniques from the theory of regions (Cortadella, Kishinevsky et al. 1998) to transform state machines to Petri nets. We then show how these Petri nets can be transformed to BPMN diagrams. Once a BPMN diagram is obtained, other existing techniques can be applied to transform these diagrams into BPEL for implementation purposes (Ouyang, Dumas et al. 2008).

The work presented in this paper can also be related to work on controllability analysis of service protocols (Lohmann et al. 2008). Controllability analysis is concerned with the following question: given a service protocol P, is there a partner protocol P' such that the choreography consisting of P and P' possesses certain characteristics such as proper termination? Meanwhile, in our work we take a choreography as a starting point, and we derive an orchestrator that is able to interact with all the existing parties in the choreography in order to mediate between all their interactions.

7 Conclusion

We have presented a tool chain for synthesising the behaviour of an orchestrator from a service choreography. This tool chain effectively provides a basis for altering the topology of a service-oriented system composed of services that engage in long-running conversations with one another.

The tool chain starts with the assumption that choreographies are represented as communicating FSMs, and that the communication medium is synchronous. We argued that state machines provide a suitable starting point for this tool chain, pointing out that, under reasonable assumptions, it is possible to transform choreographies specified using standard languages such as BPMN and BPEL into FSMs. We also argued that the assumption of synchronous communication provides a suitable basis for studying the problem of synthesising orchestrators from choreographies. Nonetheless, it would be interesting in future work to study the implications of relaxing this assumption.

At the core of the proposed tool chain lies an algorithm that takes as input a choreography captured as a collection of inter-connected FSMs, and synthesises an orchestrator, also captured as an FSM. Acknowledging that FSMs do not provide a suitable basis for capturing orchestrator interfaces, the tool chain reuses an existing technique to transform the synthesised FSM into a Petri net. We then provide a rules-based transformation from Petri nets to BPMN, thus enabling orchestrator synthesis from choreographies using standard notations.

The proposed tool chain, including the algorithm for orchestrator synthesis and the transformation from Petri nets to BPMN, has been implemented and tested against choreographies extracted from industry standards.

As discussed in Section 1, orchestrators provide a single entry point into a service composition, and as such, they can facilitate the introduction of added value into a service composition. In particular, orchestrators can act as single points of payment or as entry points for tracking a service composition. A direction for future work is to study the organisational implications and the opportunities opened by the possibility of changing the topology of a service composition from a choreographed style to an orchestrated style.

Acknowledgment: This work was partly funded by an ARC Linkage Project (LP0669244) co-sponsored by SAP and Queensland Government.

8 References

Benatallah, B., Casati, F., et al. (2006): Representing, analysing and managing web service protocols. *Data Knowledge Engineering* **58**(3):327-357.

Berardi, D., Calvanese, D., De Giacomo, G., Hull, R. and Mecella, M. (2005): Automatic composition of transition-based semantic web services with messaging. *Proc. 31st VLDB Conference*, Trondheim, Norway, 613-624, VLDB Endowment.

Bultan, T., Su, J., et al. (2006): Analyzing conversations of web services. *IEEE Internet Computing* **10**(1): 18-25.

Chrzastowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M. and Susanto, A. (2003): A top-down Petri net-based approach for dynamic workflow modeling. *Proc. Business Process Management 2003*, Eindhoven, The Netherlands, 336-353, Springer-Verlag.

Cortadella, J., Kishinevsky, M., Kondratyev, A. Lavagno, L. and Yakovlev, A. (1997): Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*: **3**(E80-D):315-325.

Cortadella, J., Kishinevsky, M., et al. (1998): Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers* **47**(8): 859-882.

Decker, G. and Barros, A. (2008): Interaction Modeling using BPMN. In *Business Process Management Workshops*, Lecture Notes in Computer Science, 4928:208-219, Springer-Verlag, Germany.

Decker, G., Kopp, O., Leymann, F. and Weske, M. (2007): BPEL4Chor: Extending BPEL for modeling choreographies. *Proc. International Conference on Web Services*, Salt Lake City, Utah, USA, 296-303, IEEE.

Dijkman, R. M., Dumas, M. and Ouyang, C. (2008): Semantics and analysis of business process models in BPMN. *Information and Software Technology*, To appear.

GS1 US: Voluntary Interindustry Commerce Standard (VICS), http://www.uc-council.org/ean_ucc_system/stnds_and_tech/vics_edi.html. Accessed 18 October 2007.

Hinz, S., Schmidt, K. and Stahl, C. (2005): Transforming BPEL to Petri nets. *Proc. International Conference on Business Process Management*, Nancy, France, 220–235, Springer-Verlag.

Howard, F., Emmerich, W., Kramer, J., Magee, J., Rosenbalum, D. and Uchitel, S. (2007): Model checking service compositions under resource constraints. *Proc. ESEC/FSE'07*. Cavtat near Dubrovnik, Croatia, 225-234, ACM.

Lohmann, N., Massuthe, P., Stahl, C. and Weinberg, D. (2008): Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowledge Engineering* **64**(1):38–54.

OASIS (2007): Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. Accessed 4 February 2008.

Object Management Group (2008): Business Process Modeling Notation, v1.1, http://www.omg.org/docs/formal/08-01-17.pdf. Accessed 24 March 2008.

Ouyang, C., Dumas, M., ter Hofstede, A. H. M., van der Aalst, W. M. P. (2008): Pattern-based translation of BPMN process models to BPEL web services. *International Journal of Web Services Research* **5**(1):42-61.

Palsberg, J. and Jay, C. (1998): The essence of the visitor pattern. *Proc. 22nd IEEE International Computer Software and Applications Conference*, Vienna, Austria, 9-15, IEEE.

Peltz, C. (2003): Web services orchestration and choreography. *IEEE Computer* **36**(10):46-52.

van der Aalst, W. M. P. (1998): The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers* **8**(1):21-66.

van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., and van der Aalst, W. (2005): The ProM framework: A New Era in Process Mining Tool Support. *Proc. 26th International Conference on Application and Theory of Petri Nets (ATPN)*, Miami, Florida, 444-454, Springer-Verlag.

xCBL.org (2003): XML Common Business Library, http://www.xcbl.org/xcbl40/documentation.shtml. Accessed 12 September 2007.

Yellin, D. M. and Strom, R. E. (1997): Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems* **19**(2):292-333.

Zhao, W., Bryant, B.R., Cao, F., Bhattacharya, K. and Hauser, R. (2005): Transforming Business Process Models: Enabling Programming at a Higher Level. *Proc. IEEE International Conference on Services Computing (SCC)*, Orlando, FL, USA, 173-180. IEEE.