Synthesis of Orchestrators from Service Choreographies

Stephen McIlvenna  BIT (CQU)

2009

A dissertation presented to the
Faculty of Information Technology
Queensland University of Technology
in fulfilment of the requirements for the degree of
Master of Information Technology (Research)
Keywords

service composition, service protocol, choreography, orchestration, Petri nets, BPMN
Abstract

With service interaction modelling, it is customary to distinguish between two types of models: choreographies and orchestrations. A choreography describes interactions within a collection of services from a global perspective, where no service plays a privileged role. Instead, services interact in a peer-to-peer manner. In contrast, an orchestration describes the interactions between one particular service, the orchestrator, and a number of partner services.

The main proposition of this work is an approach to bridge these two modelling viewpoints by synthesising orchestrators from choreographies. To start with, choreographies are defined using a simple behaviour description language based on communicating finite state machines. From such a model, orchestrators are initially synthesised in the form of state machines. It turns out that state machines are not suitable for orchestration modelling, because orchestrators generally need to engage in concurrent interactions. To address this issue, a technique is proposed to transform state machines into process models in the Business Process Modelling Notation (BPMN). Orchestrations represented in BPMN can then be augmented with additional business logic to achieve value-adding mediation. In addition, techniques exist for refining BPMN models into executable process definitions. The transformation from state machines to BPMN relies on Petri nets as an intermediary representation and leverages techniques from theory of regions to identify concurrency in the initial Petri net. Once concurrency has been identified, the resulting Petri net is transformed into a BPMN model.

The original contributions of this work are: an algorithm to synthesise orchestrators from choreographies and a rules-based transformation from Petri nets into BPMN.
Statement of original authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

___________________
Stephen McIlvenna

2 February 2009
My principal supervisor, Marlon Dumas, thanks for having so much patience and explaining concepts so clearly. I very much appreciate your availability to regularly discuss my work, and it was a great novelty working as a team literally spanning the globe. My associate supervisor, Moe Thandar Wynn, thanks for providing support and teaching me the art of algorithm formalisation.

\[ P = \{ \text{(The Business Process Management research group for having me to stay),} \]

\[ \text{(QUT for providing resources needed to conduct this research),} \]

\[ \text{(Meinen Bürokollegen von der Margaret St für eure Gesellschaft, die Ausflüge und den Spaß während meiner Zeit an der Uni. Ich verspreche euch, dass ich es irgendwann mal schaffe endlich ins Ausland zu gehen!)} \]

\[ \text{(Mum, Dad, David, Paul and Tirzah, thanks for all the things along the way to where I am today) } \} \]
# Contents

Keywords ................................................................................................................................. i
Abstract .................................................................................................................................... iii
Statement of original authorship .............................................................................................. v
Acknowledgements ................................................................................................................... vii
Figures ...................................................................................................................................... xi
1 Introduction ............................................................................................................................ 1
   1.1 Problem area and motivation ......................................................................................... 1
   1.2 Problem statement ........................................................................................................ 6
   1.3 Contributions and scope .............................................................................................. 6
   1.4 Summary ...................................................................................................................... 7
   1.5 Outline ......................................................................................................................... 8
2 Literature .............................................................................................................................. 9
   2.1 Service oriented architecture ....................................................................................... 9
      2.1.1 Web services ........................................................................................................ 9
      2.1.2 Service composition ............................................................................................ 10
   2.2 Languages for service behaviour modelling ................................................................. 12
      2.2.1 Business Process Execution Language ............................................................... 13
      2.2.2 BPEL derivatives ............................................................................................... 14
      2.2.3 Choreography Description Language .................................................................. 14
      2.2.4 Let’s Dance ........................................................................................................ 14
      2.2.5 State machines ................................................................................................... 14
   2.3 Petri nets ..................................................................................................................... 17
   2.4 Business Process Modelling Notation ......................................................................... 19
   2.5 Summary ..................................................................................................................... 21
3 Orchestrator synthesis .......................................................................................................... 23
   3.1 Interface definition ..................................................................................................... 24
   3.2 Synthesis .................................................................................................................... 26
   3.3 Deadlock detection ..................................................................................................... 38
   3.4 Observations .............................................................................................................. 39
   3.5 Implementation ......................................................................................................... 41
      3.5.1 Interface design .................................................................................................. 41
      3.5.2 Tooling overview .............................................................................................. 42
   3.6 Validation ................................................................................................................... 45
3.6.1 Two parties ................................................................. 45
3.6.2 Three parties .............................................................. 48
3.6.3 Four parties ................................................................. 49
3.6.4 Deadlock demonstration ............................................... 55
3.7 Summary ............................................................................ 56

4 Readable service behaviour modelling .............................................. 57
  4.1 State machines to Petri nets ................................................... 58
  4.2 Petri nets to BPMN diagrams .................................................. 59
  4.3 Implementation ..................................................................... 70
    4.3.1 State machines to Petri nets ............................................. 70
    4.3.2 Petri nets to BPMN diagrams ............................................ 71
    4.3.3 BPMN readability optimisation ......................................... 72
  4.4 Summary ............................................................................. 76

5 Conclusion ............................................................................... 77
  5.1 Contributions ....................................................................... 77
  5.2 Future work ......................................................................... 78

Bibliography ............................................................................... 79
Figures

Figure 1: Locally-defined orchestrations composed with a global choreography .................................................. 2
Figure 2: Refining a choreography into an orchestration ......................................................................................... 2
Figure 3: Choreographed land development application payment ................................................................. 4
Figure 4: Orchestrationed land development application payment ............................................................ 4
Figure 5: Behaviourally incompatible BPEL processes ....................................................................................... 13
Figure 6: Communicating FSMs labelled with message interactions .............................................................. 15
Figure 7: A non-deterministic FSM derived from B in Figure 6 ............................................................................. 15
Figure 8: Transforming a state machine into a Petri net ....................................................................................... 17
Figure 9: Collapsing a region of concurrency ..................................................................................................... 18
Figure 10: Subset of BPMN elements ............................................................................................................... 19
Figure 11: Message exchange syntax .................................................................................................................. 25
Figure 12: Deterministic choice on message data ................................................................................................. 25
Figure 13: Choreography topology ..................................................................................................................... 27
Figure 14: The buyer interface ............................................................................................................................ 28
Figure 15: The supplier interface .......................................................................................................................... 28
Figure 16: The shipper interface ........................................................................................................................... 28
Figure 17: Synthesis of interface pairs ................................................................................................................ 29
Figure 18: Overview of synthesiseInterfacePair() ............................................................................................ 32
Figure 19: Interfaces prior to synthesis .................................................................................................................. 36
Figure 20: Synthesised orchestrator interface .................................................................................................... 36
Figure 21: Synthesised interface from the buyer and supplier .............................................................................. 37
Figure 22: The shipper interface which causes deadlock ................................................................................... 38
Figure 23: Service topology after introducing the orchestrator .......................................................................... 40
Figure 24: Editor generation with the GMF .......................................................................................................... 41
Figure 25: Interface designer ................................................................................................................................ 42
Figure 26: Tooling overview ................................................................................................................................. 43
Figure 27: UML class diagram of the plain object model ...................................................................................... 44
Figure 28: Example xCBL buyer FSM ................................................................................................................ 46
Figure 29: Example generated xCBL supplier FSM (mirrored from the buyer) .................................................. 46
Figure 30: Example xCBL orchestrator FSM ...................................................................................................... 47
Figure 31: VICS ordering scenario ....................................................................................................................... 49
Figure 32: Four party choreography .................................................................................................................... 50
Figure 33: Buyer FSM ........................................................................................................................................ 50
Figure 34: Supplier FSM ..................................................................................................................................... 51
Figure 35: Shipper FSM .................................................................................................................. 51
Figure 36: Airline FSM ..................................................................................................................... 52
Figure 37: Partially synthesised orchestrator FSM, where $P = \{\text{buyer, supplier, shipper}\}$ ... 53
Figure 38: Shipper FSM which caused deadlock ............................................................................... 55
Figure 39: Transforming state machines into BPMN diagrams ......................................................... 57
Figure 40: BPMN diagram with a logic pool and a black box pool ................................................... 59
Figure 41: Concise message forwarding notation ............................................................................... 60
Figure 42: Rule dependencies ........................................................................................................... 60
Figure 43: Mapping a place to a start event ....................................................................................... 62
Figure 44: Adding an end event ........................................................................................................ 62
Figure 45: Removing a place ............................................................................................................. 63
Figure 46: Mapping a place to an XOR merge gateway ................................................................. 63
Figure 47: Mapping a place to an event-based XOR gateway ......................................................... 64
Figure 48: Adding an AND fork ...................................................................................................... 64
Figure 49: Adding an AND join ....................................................................................................... 65
Figure 50: Mapping a data-based XOR decision with one message type present ......................... 66
Figure 51: Mapping a data-based XOR decision with multiple message types present ............... 66
Figure 52: Mapping transitions to tasks ......................................................................................... 66
Figure 53: Initial Petri net ............................................................................................................... 68
Figure 54: Graph after applying rules 1, 2, 3, 4 and 5................................................................... 68
Figure 55: Graph after applying rules 1, 2, 3, 4, 5, 6 and 7........................................................... 69
Figure 56: Final BPMN diagram after applying all 9 rules ............................................................ 69
Figure 57: State machine to BPMN tool chain ............................................................................... 70
Figure 58: Generated diagram for a non-orchestrator with pools and message flows ............... 73
Figure 59: Petri net derived from the orchestrator, showing structure chosen by Petrify .......... 74
Figure 60: BPMN diagram for Figure 59 ...................................................................................... 74
Figure 61: Modified Petri net optimised for cleaner BPMN generation .................................... 75
Figure 62: BPMN diagram for Figure 61 ....................................................................................... 75
Figure 63: Fully synthesised orchestrator, where $P = \{\text{buyer, supplier, shipper, airline}\}$ .... 83
Figure 64: Petri net generated by Petrify for the working example, arranged with DOT ........ 84
Figure 65: BPMN diagram generated for the working example orchestrator ............................ 84
1 Introduction

A Service-Oriented Architecture (SOA) is an architecture for a software system where the basic elements are services - meaning entities that 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 exchange messages according to certain contracts. A software service is called a Web Service (WS) if it applies open standards such as eXtensible Markup Language (XML), Web Service Description Language (WSDL), and/or SOAP.

1.1 Problem area and motivation

A typical approach to design an SOA is to identify basic services and to then 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.

There are two main viewpoints when defining a composition of services: choreographies and orchestrations. A choreography describes interactions within a collection of services from a global perspective. Conversely, an orchestration is defined from a local perspective and describes interactions between one particular service, the orchestrator, and a number of partner services.
How the two are related

Given that orchestrations define the behaviour of stand-alone services, which can in turn be choreographed with other services as illustrated in Figure 1, both approaches to service composition are complementary, and serve different purposes. Choreographies are suited to defining high level requirements in the early stages of service analysis and design. They provide a global view of service interaction without requiring detailed knowledge of each service. Orchestrations define detailed behaviour of specific services, and are suitable for capturing detailed processes in the later stages of service design. Figure 2 illustrates how interactions between services in a choreography occur in a peer-to-peer manner, but in an orchestration revolve around the centralised orchestrator.

The peer-to-peer nature of choreographies means they can become difficult to maintain with evolving requirements, since new requirements may necessitate changes to multiple services. In many cases, such changes may be resolved by introducing intermediary adapters to maintain compatibility between the original services (B. Benatallah, Casati, Grigori, Nezhad, 2003).

Figure 1: Locally-defined orchestrations composed with a global choreography

Figure 2: Changing a choreography into an orchestration

---

1 (Peltz, 2003)
& Toumani, 2005). In other cases, changes in requirements necessitate the underlying services to be updated. Where the design choice of composition by choreography causes escalated maintenance efforts, compositions are not easily maintained and cannot respond agilely to dynamic requirements.

Take for instance a choreography scenario where an order management service, an invoicing service and a logistics service interact with one another. If the interactions need to be changed, it could result in all three services requiring updating to reflect the changes. Alternatively, adapters could be introduced between each pair of services to maintain compatibility with the existing implementation and the new requirements. Where adapters are continually added to compensate for interface changes of evolving services, proliferation of adapters hampers maintainability.

Orchestrations, on the other hand, revolve around the orchestrator service, and can be more easily updated to reflect new requirements. Coming back to the example, if the order management, invoicing and logistics services interacted through a single orchestrator, it becomes possible to change the overall behaviour of the system by updating just the orchestrator. The introduction of orchestrators eliminates the problem of adapter proliferation, as orchestrators can act as single points of adaptation for multiple parties.

When choreographies become mature, and parties have a relationship of trust between them, it may be beneficial to change the choreographies into orchestrations. For example, consider a system for processing business land development applications. Such land development applications require documents to be exchanged between multiple government agencies and local authorities, like the city council in which the property is located, the Department of Transport for road planning issues, and the Environmental Protection Agency. If a system supporting this business process was implemented by choreography, meaning that the services representing the various agencies interact in a peer-to-peer manner, it would be difficult to subsequently add a single point of payment. Instead, as seen in Figure 3, each agency would require the applicant to make a separate payment for the service it offers. On the other hand, if a single orchestrator were responsible for coordinating all the services, it would become feasible, at least from a technical implementation angle, to introduce a single point of payment at the level of the orchestrator. In other words, the orchestrator could be augmented to add value on top of the existing services by allowing the applicant to make a single payment, which the orchestrator could distribute to the end recipients, as shown in Figure 4. The orchestrator could also maintain logs that could be later used to analyse overall system performance or other non-functional properties of the system—something that is more difficult to achieve while the interactions occur in a peer-to-peer manner.
Such an example assumes the various government departments have to some extent mutually trusting relationships or agreements to allow payment to be centralised. Changing choreographies into orchestrations requires the service owners involved to be agreeable to there being a single point of control, which is possibly outside of their ownership domain. While an important facet of service governance (Malinverno, 2006), ownership does not affect the technical feasibility of refining choreographies into orchestrations.
Another motivation for the desire to turn choreographies into orchestrations is for service provision to third parties. The emergence of web service marketplaces provides a platform where services are dealt with like commodities, where services are provided to external parties. To expose the functionality of a choreography to unknown external parties, it is of interest for integration and security reasons to provide a single endpoint which can be exposed to the external parties. The ability to synthesise such endpoints would expedite the provision of distributed functionality through web service marketplaces.

Presently, there is a lack of methods for bridging the two composition viewpoints, in particular to synthesise orchestrators from choreographies. A synthesised orchestrator, or orchestrator template, achieves the same effect as the originating choreography, but allows a single point of configuration which can in turn be refined and embellished with value-adding functionality.

In order to synthesise orchestrators from choreographies, it is necessary to agree on a language for representing choreographies and orchestrations. Following previous notable work in the field (Yellin & Strom, 1997), a simple language based on Finite State Machines (FSMs) is adopted in this work. Specifically, the protocol, also called the behavioural interface, of each service participating in a choreography is expressed as an FSM and a choreography is implicitly specified as the collection of protocols of the involved services. In a first stage, FSMs are adopted to also represent the synthesised orchestrators. Adopting a simple language to uniformly represent choreographies and orchestrations allows the core aspects of the problem to be focussed on without being distracted by the numerous features and complexities found in more fully-fledged languages.

There is one side effect of using a simple language based on state machines to represent orchestrations where they tend to involve interactions that may occur concurrently. While in principle it is still possible to capture concurrent interactions using state machines, which are essentially sequential, this design choice leads to a combinatorial explosion. Essentially, it is necessary to enumerate in the state machine every possible permutation of each group of interactions that may occur concurrently. This approach is impractical.

Business process modelling languages are designed to represent such concurrent interactions in graphical process models easily comprehended and understood by humans. These provide a cleaner way of representing orchestrators. Ideally process models for orchestrators would be executable, providing a single process definition for the human modeller and the underlying software services.
1.2 Problem statement

The problem addressed by this research is how to refine service choreographies into orchestrator templates by synthesising orchestrators in a readable and executable form.

- How can the external behaviour of services be explicitly defined to capture messaging protocols in a simple manner? Existing approaches allow ambiguous protocol specification, which leads to unpredictable service behaviour.
- How can orchestrator templates be synthesised from choreographies where services are defined with simple behavioural descriptions based on state machines? Service evolution may require changes to multiple services, or the introduction of adapters. Changing choreographies into orchestrations introduces a single point of control.
- How can the behaviour of synthesised orchestrators be represented in readable and possibly executable process models? Expressing concurrency in a simple language based on state machines leads to unnecessary verboseness, especially in the case of orchestrators, which contain the product of all logic in the underlying service compositions. Therefore an approach to represent the synthesised orchestrators as process models is investigated.

1.3 Contributions and scope

The contributions of this research are:

- A simple language to specify service messaging protocols
- An algorithm to synthesise orchestrators from choreographies, using the simple language
- An approach to transform orchestrator services defined in the simple language into readable business process models defined in a standardised process modelling language.

Concretely, the main outcome of this research is a method for taking explicit behavioural descriptions of services from choreographies, and synthesising orchestrator services. To achieve this, the behaviour of services needs to be defined using a simple language where protocols can be described without ambiguity. Such a language is proposed. Orchestrators synthesised in the simple language are not readable if concurrency is expressed, so a transformation to a business process modelling language is presented.

The research is particularly concerned with services that engage in long-running interactions in the context of a business process. This is the case for example in an order management service, that offers the possibility, via other services or via front-end software applications, to submit purchase orders; to monitor the status of these orders; to cancel submitted purchase
orders; or to otherwise affect the course of the underlying business process. Such services are sometimes also called conversational services (B. Benatallah, Casati, & Toumani, 2004). They can be distinguished from functional services, that engage in simple request-response interactions, like for example a service that offers the possibility to other applications or services to consult the weather forecast for a particular location. Such a weather forecast service is said to be functional because it only engages in interactions where it receives as input requests for weather forecasts, and it responds to them, without there being any correlation between this interaction and subsequent or previous ones. While functional services are quite common and have a role to play in an SOA, it is in the context of the composition of conversational services that choreographies and orchestrations play a role in service design, and thus the research naturally focuses on conversational services.

The research focus is on service messaging behaviour, abstracting away from message schemas, which are also important for an end-to-end solution. In all the examples, message schemas, also called message types, are treated as black-boxes and they are simply identified by name. The behaviour of a service is also assumed to involve only point-to-point message exchanges, as opposed to broadcast, multicast, or publish/subscribe interactions (Barros, Dumas, & ter Hofstede, 2005). A communication environment is considered where interactions are synchronous and atomic, meaning that when a message is sent by a party, this party blocks until the recipient receives the message.

1.4 Summary

Choreographies and orchestrations for service composition achieve different goals, and both are needed to support SOA. When choreographies mature and evolve, they may become difficult to maintain, as configuration changes need to be made in more than one place. Orchestrations provide a single point of configuration, and provide a solution to the adapter proliferation problem. The motivation to refine choreographies into orchestrations is apparent where participants have a relationship of trust and will allow functionality to be centralised, such as a single point of payment for land development applications associated with multiple government agencies. Additionally, orchestrators provide convenient endpoints for provision of distributed logic to external parties through web service marketplaces. There is currently no bridge between the two viewpoints of composition, to synthesise orchestrations from choreographies. This feature is the main focus of the research at hand.
1.5 Outline

The following chapter presents existing relevant research, Chapter 3 describes an approach to orchestrator synthesis, and Chapter 4 proposes a transformation from verbose protocol descriptions to business process models. Concluding is Chapter 5, where research findings are summarised.
2 Literature

Current approaches to service composition, service behaviour modelling and representation need to be reviewed before recommending solution requirements for orchestrator synthesis and modelling to ensure the research is aligned with existing knowledge. The following sections introduce the field of work and summarise relevant literature.

2.1 Service oriented architecture

One way of implementing software to fulfil requirements is to divide functionality based on what services are needed and create autonomous software services that communicate via message exchanges. The environment created by communicating services is referred to as a Service Oriented Architecture (SOA). The service-oriented approach to software development promises the ability to swiftly respond to changing business rules (Zimmermann, Doubrovski, Grundler, & Hogg, 2005).

2.1.1 Web services

Standardisation of technologies used for software services in the context of SOA has resulted in services where the messages and interfaces are defined using non-proprietary languages. In the setting of the Web Services (WS) standards stack, services exchange messages encoded in SOAP, and their schemas, also called structural interfaces, are described using the Web Service Description Language (WSDL). These standards build on top of other standards, most notably HyperText Transfer Protocol (HTTP), eXtensible Markup Language (XML) and XML Schema. Services implemented on top of these standards are called web
services. A host of other WS standards have been proposed to support commonly required features such as messaging reliability with WS-ReliableMessaging, secure message exchange with WS-Security, and documentation of business level requirements with WS-Policy (Erl, 2005). The Business Process Execution Language (BPEL) can be used to define executable process models that involve web services (Erl, 2005).

2.1.2 Service composition

A repository of services is not worth much without the ability to compose them together to perform business processes. Two different viewpoints for modelling business processes involving web services are choreography and orchestration (Peltz, 2003).

The modelling viewpoint for choreographies is from a global perspective. Message sequences among the services concerned are tracked, rather than the specific business process of any particular service (Brogi, Canal, Pimentel, & Vallecillo, 2004). Each party involved in a choreography defines its part in the collaboration, and no single party has control over the entire choreography.

The modelling viewpoint for orchestrations is from a local perspective. Message exchanges are modelled from the view of a single party, which controls the composition (Chen, Wassermann, Emmerich, & Foster, 2006).

Choreographies and orchestrations are both useful for implementing SOA. High level requirements are suitably modelled from a global view, and are therefore acceptably modelled by means of a choreography. At this level of abstraction, each service participating in the composition is described, whether that service is in the domain being considered for implementation, or is external. For circumstances where detailed business process must be executed, orchestrations provide a more natural way of modelling service compositions, since they describe business processes from the perspective of one service that coordinates multiple partner services. Assuming all services have been implemented by some other means, the implementation of an orchestration boils down to the implementation of the service that acts as the orchestrator, and the behaviour of this composite service is captured in detail by the orchestration.

Thus, for implementing initial and high level requirements, choreographies provide a convenient starting point to discover how the composite services must interact. More detailed drivers arising later in the software development cycle demand a relatively lower level of specification, to express business logic and capture the business processes carrying out the actual work. Such mature requirements lend themselves to be implemented with
service orchestration, where processes-centric services coordinate a number of partner services.

If choreographies designed in the initial analysis and design phase of a system evolve to support more complex business processes, they will not be as maintainable as equivalent orchestrations. Business logic will be distributed across a number of services, rather than being consolidated in a single place. Additionally, as choreographies mature and service interfaces change, or services are removed or introduced, adapters may be required to maintain compatibility with the remaining parts of the choreographies (Kaminski, Müller, & Litoiu, 2006; Nezhad, Benatallah, Martens, Curbera, & Casati, 2007).

In an orchestration, the single controlling service through which all interactions are channelled is the orchestrator, which manages the interactions and stores the state of the activity being performed (Bravetti, Guidi, Lucchi, & Zavattaro, 2005). Orchestrators provide a convenient single point of configuration for service compositions where changes can be implemented in one place instead of across multiple services. Other uses of orchestrators could be for the provision of services to third parties, where endpoints are façades. Several enterprises have already emerged to provide web service marketplaces where service endpoints are traded as a commodities (Tai, Desai, & Mazzoleni, 2006) demonstrating the significance of single-point service provision. Utilisation of such web services from third parties can have a profound positive impact by enabling business process change not previously feasible (Ray & Ray, 2006).

Currently, there is no existing technique for refining choreographies into orchestrations, by providing a way of synthesising orchestrator interfaces. The type of interface this work is concerned with are behavioural interfaces, as opposed to structural interfaces that capture operations and message schemas or non-functional interfaces that capture aspects such as security, reliability and availability. Behavioural interfaces capture definitions of the supported message sequencing constraints. Behavioural interfaces are also called protocols (Barros, Dumas, & Oaks, 2005; Yellin & Strom, 1997).
2.2 Languages for service behaviour modelling

The first challenge of modelling service behaviour is to decide how protocols should be defined. This includes the structure of the protocol to capture message dependencies, as well as the environment in which protocols are used. Protocols need to be defined to reason about compatibility between services (B Benatallah, Casati, & Toumani, 2006). Services exchange messages using a pre-defined protocol to have conversations, where services commence from a starting state, and either send or receive messages according to the protocols until the services are in a state where the conversation is over. When two services cannot successfully converse without reaching deadlock, which arises when one service awaits a message which is never sent by the other service, they are said to be incompatible (Yellin & Strom, 1997).

Successful conversation, however, relies on the assumption made about the messaging environment. Of particular importance is whether buffering of messages is performed by the service endpoints or not. In a buffered environment, message transmission takes time and may be queued upon arrival if the service is not in a state in which it can process the incoming message. One problem with reasoning about protocols in a buffered environment is that reasoning with unbounded queues leads to an infinite state space which makes compatibility verification unfeasible (Bultan, Su, & Fu, 2006). One way of overcoming this problem is to restrict the queue length (Berardi, Calvanese, De Giacomo, Hull, & Mecella, 2005), or remove queues entirely (B Benatallah et al., 2006).

The approach often taken by researchers in this field appears to be the assumption of a messaging environment where message transmission is instantaneous, and the protocols of the sender and receiver for any given interaction advance in synchrony (B Benatallah et al., 2006; Yellin & Strom, 1997). While this assumption is not entirely in line with the characteristics of many existing communication protocols, which do allow for buffering, such research has shown that results obtained under the synchronous messaging assumption can in many cases still be translated to an asynchronous environment.

To assemble a model for defining service behaviour, existing languages are analysed to assess their suitability for protocol definition.
2.2.1 Business Process Execution Language

One language seemingly capable of expressing protocols is the Business Process Execution Language (BPEL). This language was designed for coordinating the flow of business process services (Acharya et al., 2005) and can specify the business process in a detailed internal form, or a more abstract external form. An example of two communicating services described abstractly in BPEL is shown in Figure 5. The client sends login data and decides whether they are a premium customer. The partner web service on the right hand side of the figure receives the login data and also determines if the customer is classified as premium. In the case where the client on the left expects to be treated as a premium customer, but the other service classifies the customer otherwise, a deadlock will occur (Moser, Martens, Häblich, & Mülle, 2006). This illustrates a fundamental flaw in BPEL for protocol definition. Deadlock can arise in this example due to the presence of internal transitions, where either service can silently move into a different state without exhibiting any externally observable change, and thus irreversibly altering the course of the conversation. It is contended that a protocol must only expose externally visible behaviour. Since BPEL allows internal behaviour to be exposed, even in so-called abstract process definitions, it is thought BPEL is unsuitable as a protocol definition language, regardless of the execution environment. The place of BPEL is more as a language for specifying executable process models involving web services as evidenced by the number of BPEL execution engines available on the market.

Figure 5: Behaviourally incompatible BPEL processes\(^2\)

\(^2\) (Moser et al., 2006)
2.2.2 BPEL derivatives

Another language for modelling service interactions is BPEL4Chor, based on BPEL and designed for defining choreographies (Decker, Kopp, Leymann, & Weske, 2007). This language proposes extensions to BPEL so the flow between participants in a choreography can be defined, as well as the behaviour of each participant. Since BPEL is used unchanged (Decker et al., 2007) the inherent weakness of allowing internal decisions is inherited. Another BPEL-based standard, BPEL Light (Nitzsche, van Lessen, Karastoyanova, & Leymann, 2007), proposes a variation not dependent on WSDL, but also inherits the flaws described above with regards to protocol definition.

2.2.3 Choreography Description Language

The Choreography Description Language for web services (WS-CDL) is a descriptive language and does not allow for executable specifications. This standard provides a global view of interactions between participants (Bravetti et al., 2005), from which participant behaviour can be derived. WS-CDL is an XML-based language and is tightly integrated with WSDL (Barros, Dumas, & Oaks, 2005). Intended for defining global interactions between participants in choreographies, WS-CDL has weaknesses when it comes to capturing orchestrations, and is thus not the most suitable candidate as a single language for defining both choreographies and orchestrations.

2.2.4 Let’s Dance

Let’s Dance (Zaha, Barros, Dumas, & ter Hofstede, 2006) is a visual language aimed at business analysts for modelling choreographies. Models described in this language encompass all choreography participants, and from such a model the participant behavioural interfaces are derivable. This proposed language has not been standardised or widely adopted, and it is based on a quite specific mix of control-flow specification constructs not found in standard languages. Its only use has been as an instrument for illustrating the problems that arise when deriving local behavioural models of participants in a choreography from global models.

2.2.5 State machines

A more fundamental, widely-studied and widely-adopted class of languages for behaviour modelling is the class of languages based on Finite State Machines (FSMs). FSMs are composed of a set of distinct states, connected by labelled transitions. In the case where FSMs are used to model possible sequences of messages, each transition signifies either receiving or sending a message (Yellin & Strom, 1997). Protocols defined as FSMs have an initial state to mark the beginning of the conversation, and a set of final or acceptance states,
indicating where the service has completed a conversation (B Benatallah et al., 2006). At any
given time, a state machine can be in only one state. Example FSMs are presented in
Figure 6, where transitions are labelled with a message name with a polarity, to indicate
whether the message is sent, denoted by ‘!’ , or received, denoted by ‘?’ . These two FSMs
communicate by advancing along matching transitions until both have reached acceptance
states, indicated by concentric circles. Matching pairs of transitions are identical except for
polarity. When A sends ‘request’ to B, A and B both advance to states where B can send
‘ack’ to A.

Figure 6: Communicating FSMs labelled with message interactions

To ensure choices are externalised when defining protocols with FSMs, it must be ensured
any given state does not have the same combination of message and polarity, as this could
cause the same misunderstanding between services as described in Section 2.2.1 if the
identically marked branches lead to different conversations. A protocol represented by an
FSM where changes in state are signalled externally, and there is no ambiguity due to
identically labelled transitions emitting from any given state is said to be deterministic. A
requirement for determinism, of course, is that all transitions are labelled. Otherwise,
changes in state along unlabelled transitions would not be externalised. Timeouts are
essentially unlabelled transitions, seen from the perspective of an external party that does not
have access to the timer, and therefore make any protocol non-deterministic.

Figure 7: A non-deterministic FSM derived from B in Figure 6

The FSMs in Figure 6 represent deterministic protocols. To illustrate non-determinism,
Figure 7 reproduces the FSM of B in Figure 6 with ‘alarm!’ replaced with ‘request?’ . From

3 (Bordeaux, Salaün, Berardi, & Mecella, 2004)
the initial state now, how will any party conversing with this service know which state the service will advance to after receiving the ‘request’ message? It cannot be determined. Clearly, determinism must be enforced to facilitate sustained conversation that does not lead to deadlock.

It is proposed a simple FSM-based language be used to define the behaviour of each participant in choreographies for which orchestrators are to be synthesised. The assumption of instantaneous message transfer without queues appears to be required to ensure the research problem is of an achievable scope. Take for example the communicating FSMs in Figure 6. If these services were to exchange messages in a buffered environment, A could send ‘request’ and B could send ‘alarm’, and deadlock would arise. One solution to overcome problems arising from transmission latency is to introduce a third party to coordinate synchronisation messages to ensure conversation can continue (Bultan et al., 2006). With orchestrator synthesis, such approaches would be expected to cause the solution to escalate in complexity, and will therefore not be pursued, opting instead for the assumption of a messaging environment without buffering.

FSM-based approaches to modelling service behaviour provide the basis for capturing deterministic protocols. Existing FSM languages for service behaviour definition are very similar to the example provided above. Some languages use a different syntax for message polarity, namely ‘+’ for receive, and ‘−’ for send interactions (B Benatallah et al., 2006; Nezhad et al., 2007). Current FSM languages define labels in terms of message type, such as OrderResponse, and do not allow deterministic choice on message data. To deterministically model order acceptance or rejection, two distinct message types are required, like OrderResponseAcceptance and OrderResponseRejection. An approach is needed where multiple transitions emitting from a given state can be labelled with the same message type and uniquely qualified with Boolean condition guards expressed in terms of the message contents, such as OrderResponse[processed=true] and OrderResponse[processed=false].

One recognised drawback, however, is that the inherent explicitness of state machines leads to state explosion (Mach & Plasil, 2004). This is especially true of service modelling where multiple messages can be sent or received in any order. In such a scenario, each possible case must be sequentially defined, leading to verbose FSMs. For the specification of orchestrations, this would be a greater problem, as interactions between the orchestrator and the partner services often do not have to occur sequentially, so if orchestrator behaviour could be defined in an FSM-like language supporting concurrency, the logic would be easier to visualise.
2.3 Petri nets

Petri nets provide a graphical language capable of expressing concurrent interactions. Consisting of places, transitions, and directed arcs, Petri nets have been used for high level workflow management (Adam, Atluri, & Huang, 1998; van der Aalst, 1998) and on a more detailed level for modelling process behaviour (Hinz, Schmidt, & Stahl, 2005; Lohmann, Massuthe, Stahl, & Weinberg, 2008). Places may contain tokens, which can be distributed across the net throughout the execution. Token location determines which transitions can be fired, since a transition can only fire when there are tokens in all of its input places. The initial marking is the arrangement of tokens prior to execution. For modelling orchestrator behaviour, where interactions often occur concurrently, Petri nets can be used to address the problem of state explosion that occurs with state machines.

Taking a verbose model where message interactions are expressed sequentially, and producing a model with concurrency, is possible by identifying the regions of concurrency in the verbose model, and replacing those regions with concurrent equivalents. Research applying the theory of regions (Cortadella, Kishinevsky, Lavagno, & Yakovlev, 1998) has been conducted and applied to the problem of synthesis of logical controller circuits (Josephs, Nowick, & Van Berkel, 1999), and for process mining (van der Aalst, Rubin, van Dongen, Kindler, & Gunther, 2006).

The bridge from state machines to Petri nets is trivial, as depicted in Figure 8. In functional terms from state machine to Petri nets, each state in an input FSM becomes a place node in the target Petri net, and each transition from the FSM a Petri net transition node, connecting the respective input and output places with directed arcs.

![Figure 8: Transforming a state machine into a Petri net](image-url)
The theory of regions applied to Petri net models having concurrent interactions expressed as verbose sequences reduces the number of nodes and simplifies the logic. Figure 9 shows how a region of concurrency can be collapsed while retaining the identical logic, assuming transitions are atomic, and are interleaved throughout regions of concurrent processing. Algorithms for collapsing regions of concurrency are implemented by Petrify (Cortadella, Kishinevsky, Kondratyev, Lavagno, & Yakovlev, 1997), a tool which can be executed with input parameters that influence properties of the resulting Petri net, such as the free choice property. A net is a free choice net if and only if for every two transitions—if they share any input place, they share all input places (Chrzastowski-Wachtel, Benatallah, Hamadi, O’Dell, & Susanto, 2003). This structural restriction ensures choice and synchronisation cannot be mixed (van der Aalst et al., 2006), which would be difficult to represent in a higher level process modelling language. It seems business process modelling languages are free choice by design to promote capturing logic in a form easily understood by human modellers (Dehnert & van der Aalst, 2004).

As shown, Petri nets and the theory of regions can reduce the complexity of behavioural models compared to state-based transition systems in the presence of concurrent behaviour. Petri nets were not designed with the business person in mind, and are better suited for modelling, analysing and simulating dynamic systems (List & Korherr, 2006), rather than business processes. For orchestrators, it would be beneficial to represent service behaviour in a high level language to gain an understanding from a business perspective. This is especially true when orchestrators are destined for value-adding mediation, where the process flows of the synthesised orchestrators need to be changed in response to business factors.
2.4 Business Process Modelling Notation

Translating Petri nets to process models in higher level languages may cause the models to be troublesome to implement, since not all process modelling languages are executable languages. There is a recent standardised modelling language, the Business Process Modelling Notation (BPMN) (OMG, 2006), which although is not natively executable, was designed for compatibility with BPEL for process execution (Rosemann, Recker, Indulska, & Green, 2006). BPMN is a graphical notation for modelling business activities, so it can be used to present orchestrator logic to a business audience without sacrificing execution potential. While techniques exist for generating executable BPEL process definitions from BPMN models (Ouyang, Dumas, ter Hofstede, & van der Aalst, 2008; White, 2005), they do not appear to be infallible due to conceptual mismatches between the languages (Recker & Mendling, 2006). Still, BPMN can be seen as a starting point in the process of implementing service orchestrations on top of BPEL or other technologies.

Figure 10: Subset of BPMN elements

1. Apart from intermediate error events, intermediate message or timer events may also be the source of exception flows.
2. A message flow may link task to task, task to start event, task to end event, and end event to start event.

(Dijkman, Dumas, & Ouyang, 2008)
The main elements of BPMN are summarised in Figure 10. Processes are modelled from the perspective of individual parties, and the logic for each party is contained inside a separate pool. Processes in each pool commence with an event that triggers flow commencement. An individual process model is composed of activities, events and gateways that are linked through sequence flows. An event denotes something that occurs instantaneously. An activity is a piece of work to be performed. In this work, activities correspond to sending or receiving messages. Gateways serve to drive the way in which the flow of control is passed around during execution, and this flow of control ultimately determines which activities are performed and when. The flow of control begins with the start event in a pool, and it flows throughout the activities, events and gateways until the end event of the pool is reached. Simple protocols, such as those expressed in Petri nets, do not require more complex BPMN constructs such as timeouts, errors and cancellation regions. Data-based choice in BPMN will be used since in the simple protocol definition language to be developed, condition guards will externalise choice based on message data.

As shown in Figure 10, there are different types of gateways. For example, AND forks are gateways where the flow of control is split into two threads, while XOR merges are gateways where several mutually exclusive branches come together. Some AND fork gateways may be implicitly captured in a BPMN diagram. Specifically, the BPMN specification stipulates that if two or more sequence edges stem from a single task, this implies concurrent processing (OMG, 2006), and it is thus equivalent to having the activity in question being followed by an AND fork gateway. Conversely, a task with multiple incoming sequence edges means there is a merge from a previous XOR decision. In other words, a task with multiple incoming edges is equivalent to having this task immediately preceded by an XOR merge gateway where these multiple flows converge. Thus multiple outgoing sequence flows from an activity imply an AND fork, whereas multiple incoming sequence flows pointing to a single activity imply an XOR merge. These different semantics of multiple incoming versus outgoing arcs do not seem intuitive, so to ensure process models are as least ambiguous as possible, all gateways should be explicitly represented.

Service oriented processes modelled in BPMN show the relationships between services utilised to achieve the overall goal. Composite services modelled in BPMN would therefore show, at a conceptual level, how the services are structurally composed and the behavioural dependencies between them (Decker & Barros, 2008). Activities in such models would correspond to message sending and receiving.
To visualise orchestrator logic from Petri nets in BPMN, the Petri nets could be translated into BPMN. Not only do the two modelling domains use different visual elements, they also have differing expressive power. BPMN is inherently free-choice (Dijkman et al., 2008) meaning that choice and synchronisation are strictly separated. Translation of a Petri net that is not free choice to BPMN is not always possible. It is proposed that only free-choice Petri nets should be considered for transformation into BPMN.

### 2.5 Summary

Service composition is essential for realising business processes implemented with web services and service oriented architecture. The two customary approaches to composition, choreography and orchestration, are applicable to different stages of system analysis, but there is currently no bridge between the two methods.

Modelling service interactions involves tracking messages received and sent by services, in particular sequences defined by messaging protocols. Existing web service standards do not enforce definition of deterministic protocols, which can lead to incompatibility where deadlock arises. Conversations of services in messaging environments where services have queues, and where message transmission takes time, are more difficult to analyse from the perspective of computational complexity. It is suggested that a simple FSM-based language be proposed to capture deterministic protocols. Protocols defined in this way should only expose external behaviour and thus should allow for choices to be made only based on the type and contents of messages.

The biggest drawback in defining FSM-based protocols containing concurrent interactions is the problem of state explosion. The theory of regions is useful for collapsing regions of concurrency to simplify the representation of logic in protocol definitions. Readability could be further improved by transforming Petri nets into BPMN, so an approach to achieve this is desired.
3 Orchestration synthesis

A service choreography can be defined by specifying the behavioural interfaces of the participating services. To achieve the goal of synthesising a behavioural interface for an orchestrator from the behavioural interfaces composing a choreography, a language for defining behavioural service interfaces based on state machines is outlined in Section 3.1. An algorithm for synthesising orchestrator interfaces is then presented in Section 3.2, which accepts as input a list of behavioural interfaces of services composing a choreography, and produces a single orchestrator interface. Section 3.3 then demonstrates how the synthesis algorithm detects deadlocked message exchanges, and Section 3.4 discusses how orchestrators can be used with the choreographies from which they were derived.

The formal interface specification model and synthesis algorithm was implemented to produce a toolset that allows users to graphically edit behavioural interfaces, to synthesise orchestrator interfaces, and to view the synthesised interfaces. This toolset is detailed in Section 3.5. Validation of the synthesis algorithm is detailed in Section 3.6. This test-driven validation was conducted using choreographies defined by industry reference models, and covers tests with choreographies involving different numbers of services interacting in various topologies.

For the sake of convenience, in this and subsequent chapters, behavioural interfaces will be referred to simply as interfaces.
3.1 Interface definition

Finite State Machines (FSMs) are used to capture service behaviour, where every transition represents a message being exchanged between services. Each transition is labelled with a message exchange, meaning no unlabelled or empty transitions are allowed, in order to avoid non-determinism, which may lead to unpredictable behaviour. In other words, for any protocol, the set of possible message exchanges should not be influenced by any internal action not specified in the protocol.

Each message exchange represents a message either being sent or received. The direction of message movement is the message polarity, and indicates whether the action associated with the exchange is a ‘send’ or ‘receive.’ Communication between parties is assumed to be bilateral, where each message is sent point-to-point from one party to another. Protocols that interact with more than one party must therefore enumerate the parties involved in the message exchanges. There is also an assumption that in every execution of the choreography, only one service will play each role in the choreography. For example, if a choreography involves three roles: customer, supplier and shipper, then in any execution of this choreography, there will be one service playing the role of customer, one playing the role of supplier, and one playing the role of shipper.

The interface of an orchestrator synthesised from a choreography is such that, typically, the orchestrator receives a message from one participating service and it immediately forwards this message to another participant. This is because the orchestrator is introduced into the choreography for the purpose of acting as a single point of control. Looking at the interface of an orchestrator, one will thus observe a proliferation of receive-send actions, whereby the same message is merely received from one party, and forwarded to the next. To simplify the specification of orchestrator protocols, message exchanges are represented as the quadruplet Exchange = (p_from : Party, p_to : Party, msg : MessageType, forwarding : Boolean). For an exchange corresponding to either one send or one receive, where forwarding does not occur, forwarding=false. For an exchange corresponding to receiving and immediately sending the identical message, forwarding=true.

The assumption of a communication environment without buffering and transmission latency, as discussed in the previous chapter, means messages are consumed instantaneously. For this reason, it is also assumed that the sending component of a forwarding action occurs as soon as the receiving component happens. For all exchanges, message polarity is represented by the first two tuple elements, as shown by Figure 11, where the arrow between the party names is used to unambiguously show message flow direction.
To achieve determinism, hidden decisions based on message contents are disallowed. To deterministically model such choice, Boolean condition guards expressed in terms of the message content can be externalised by inclusion into the message component of the exchange. For example, to model the choice where a supplier decides whether to process an order, the message OrderResponse can be coupled with the expressions \([\text{processed}=\text{true}]\) and \([\text{processed}=\text{false}]\), meaning that if the order has been processed then one branch is taken, otherwise the other branch is taken. Here, it is assumed ‘processed’ corresponds to a property that can be directly extracted from the message being exchanged, in this example OrderResponse, and therefore both the sender and the receiver are able to evaluate these two Boolean expressions. In this way, the choice is fully externalised, and both parties are able to determine which branch is taken.

**Figure 12: Deterministic choice on message data**

The complete specification of a protocol, represented by an FSM, is composed of states and transitions, where each transition is labelled with one message exchange. Merging two FSMs, an operation that will be introduced in details later in this chapter, results in a synthesised product where each state is composed of two states—one from each of the original FSMs. Therefore, in the FSM model defined here, each state may be either an elementary state, containing no other states, or a composite state, containing a pair of states of either type. Each FSM has one initial state, and a set of final states. It is assumed all final states are sink states, containing no outgoing transitions.
A protocol is therefore defined as the state automaton $FSM = (S, s_0, E, \delta)$ 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 protocols.
- $s_0 \in S$ 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 FSM and the set of parties that the FSM represents, hence, an interface is a pair $(P : \{\text{Party}\}, sm : FSM)$. 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. However, in the case of an orchestrator service, the behavioural interface essentially represents the aggregated behaviour of multiple services. Thus the convention is adopted that the behavioural interface of an orchestrator service represents all the parties, and accordingly, $P$ will contain multiple parties in the case of an orchestrator.

### 3.2 Synthesis

The goal of orchestrator synthesis is to generate a behaviourally compatible message forwarding service capable of intercepting messages within a given choreography. After introducing an orchestrator to a choreography, the services from the choreography interact only through the orchestrator. To achieve this goal, the interfaces of each service in the choreography must be available. Synthesis is performed by an algorithm consisting of three functions, which merge behavioural interfaces of a choreography and produce the interface of an orchestrator for that choreography.

To illustrate by a working example, a choreography involving three participants will be employed. It involves a buyer ordering goods from a supplier, and a shipper delivering goods to the buyer. As Figure 13 shows, each service interacts with every other service in the choreography. The buyer initiates the interaction by sending an order to the supplier, who responds indicating whether the order was processed successfully. If the order was processed, the supplier sends a request for shipping to the shipper, who in turn can optionally respond with any number of update messages to report on shipping status. Next, the supplier issues an invoice to the buyer, and the shipper sends a delivery notice to the buyer, terminating interaction relating to the original order. An alternative case for the ordering process occurs where the supplier deems the order is not able to be processed, in which case
the order response sent from the supplier to the buyer is the final message for the ordering scenario, and the shipper is never involved.

**Figure 13: Choreography topology**

The interfaces of these parties are described in Figure 14, Figure 15 and Figure 16. In each case, an interface consists of the party being represented, and the associated FSM. Each FSM consists of only elementary states because merging has not yet commenced. Initial states have an identifier of 1. All other state identifiers within the same FSM are unique and defined arbitrarily. In the FSM of the buyer, states 3 and 6 are final states as they have no outgoing transitions.
Figure 14: The buyer interface

Figure 15: The supplier interface

Figure 16: The shipper interface
The approach used for synthesis 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 17. If a deadlock condition is detected while synthesising any interface pair, the synthesise() function immediately returns Deadlock to indicate synthesis is not possible.

**Figure 17: Synthesis of interface pairs**

The algorithm uses 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 clarity of the logic in formal function definitions, the transition function can be characterised as a set of Transition objects, each composed of a message exchange, one source and one target state. Unordered sets are denoted by {} and ordered lists by [].

<table>
<thead>
<tr>
<th>Function: synthesise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: I_all : [Interface]</td>
</tr>
<tr>
<td>Output: Interface Union Deadlock</td>
</tr>
<tr>
<td>Preconditions:</td>
</tr>
<tr>
<td>Variables: synthesised : Interface Union Deadlock</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>synthesised := synthesiseInterfacePair(dequeue(I_all), dequeue(I_all))</td>
</tr>
<tr>
<td>if synthesised is Deadlock</td>
</tr>
<tr>
<td>return synthesised</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>while I_all ≠ []</td>
</tr>
<tr>
<td>synthesised := synthesiseInterfacePair(synthesised, dequeue(I_all))</td>
</tr>
<tr>
<td>if synthesised is Deadlock</td>
</tr>
<tr>
<td>return synthesised</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end while</td>
</tr>
<tr>
<td>return synthesised</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

**Function 1: synthesise()**

The input interfaces must also be ordered according to the role of each interface in the underlying choreography. It is assumed only one interface FSM can send a message from the initial state and the others start their execution by receiving a message. This essentially means that there is a single party in a choreography that is responsible for starting the execution. In the case of the choreography involving a buyer, a supplier and a shipper, it would be the buyer who would start an execution of the choreography. The two interfaces exchanging the first message must be the first two in the input list to ensure orchestrator behaviour is captured from the beginning of the choreography. Subsequent interfaces in the list must appear in the order in which they come into the choreography. The effect of this ordering is shown in Figure 17.

The synthesise() function uses synthesiseInterfacePair(), shown in a flowchart in Figure 18 and detailed in Function 2, to completely merge two interfaces. This function synchronously traverses the FSMs of the input interfaces to build the orchestrator FSM. The function performs a synchronous breadth-first search of the two input FSMs. A breadth-first search begins with a root node, and explores all neighbouring nodes. For each of those nodes, the neighbouring nodes are explored (Broder et al., 2000), and so on, until a goal is found. In the case of FSM synthesis, both input FSMs are searched in synchrony by looking at state pairs,
where a state pair is composed of a state from each input FSM. The goal is the set of all pairs of final states. An algorithm utilising the visitor pattern (Palsberg & Jay, 1998), where a queue contains nodes to visit in the order they are to be explored, is a suitable way of implementing a breadth-first search. Nodes are visited once, and when the queue is empty, the graph has been fully searched.

In synthesiseInterfacePair(), the root node for performing a breadth-first search is the composite state derived from the initial states of the input automata. This state is the initial state of the orchestrator automaton, and is placed in a queue of states to be processed. Only composite states created for the orchestrator 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 only visited once. The input automata pair are completely synthesised 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, a supplementary function attempts to find pairs of message exchanges that can occur between the two input automata from the orchestrator state in question. A match is apparent when both input automata can synchronously exchange a message of the same type 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 included in the orchestrator synthesis. A tuple indicating that a transition could not be matched is added to the set of pairings only if the exchange is orchestrated, or is related to an interface not yet considered for orchestrator synthesis. 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 may indicate a deadlock.
Figure 18: Overview of synthesiseInterfacePair()
Function: synthesiseInterfacePair
Input: $i_a = (P_a, s_m_a = [S_a, s_0_a, E_a, \delta_a])$: Interface, $i_b = (P_b, s_m_b = [S_b, s_0_b, E_b, \delta_b])$: Interface
Output: Interface $\cup$ Deadlock
Preconditions: $\forall s \in S_b$, s is ElementaryState
Variables: $s_{0_m}, s_{current}, s_{new}, s_{toAdd}$: CompositeState
$S_{toVisit} = \{\text{CompositeState}\}$, $S_{visited} = \{\text{CompositeState}\}$
$s_m = [S_m, s_{0_m}, E_m, \delta_m]$: FSM
$t_{a\_mustSynthesise} = \{\text{Transition}\}$, $t_{b\_mustSynthesise} = \{\text{Transition}\}$
$e_{new} = \text{Exchange}$
$TP_{all} = \{(\text{Transition}, \text{Transition})\}$

begin
$s_{0_m} := (s_0_a, s_0_b)$
states($s_m$) := $\{s_{0_m}\}$
enqueue($S_{toVisit}$, $s_{0_m}$)
$P_{known} := P_a \cup P_b$
$t_{a\_mustSynthesise} := \{t \in \delta_a \cup e \in \text{exchange}(t) \mid \text{fromParty}(e) \in P_{known} \wedge \text{toParty}(e) \in P_{known} :$
\text{forwarding}(e) \wedge t\}$
$t_{b\_mustSynthesise} := \{t \in \delta_b \cup e \in \text{exchange}(t) \mid \text{fromParty}(e) \in P_{known} \wedge \text{toParty}(e) \in P_{known} :$
\text{forwarding}(e) \wedge t\} \not=$
while $S_{toVisit} \neq \{\}$
$S_{current} := \text{dequeue}(S_{toVisit})$
$S_{visited} := S_{visited} \cup \{S_{current}\}$
$TP_{all} := \text{transitionPairings}(t_{a\_known}, s_{1}(S_{current}), i_b, s_{2}(S_{current}))$
for each $(t_a$, $t_b)$ in $TP_{all}$
if $t_a \neq \text{NULL} \wedge t_b \neq \text{NULL}$
$s_{new} := \text{target}(t_a), \text{target}(t_b)$
$e_{new} := (\text{fromParty}(\text{exchange}(t_a)), \text{toParty}(\text{exchange}(t_b)),$
\text{msg}(\text{exchange}(t_a)), \text{true})$
else if $t_a = \text{NULL}$
$s_{new} := (s_{1}(S_{current}), \text{target}(t_b))$
$e_{new} := \text{exchange}(t_b)$
else
$s_{new} := (\text{target}(t_a), s_{2}(S_{current}))$
$e_{new} := \text{exchange}(t_a)$
end if
states($s_m$) := states($s_m$) $\cup \{s_{new}\}$
exchanges($s_m$) := exchanges($s_m$) $\cup \{e_{new}\}$
$t_{in} := t_{in} \cup \{(S_{current}, e_{new}) \rightarrow s_{new}\}$
t_{a\_mustSynthesise} := $t_{a\_mustSynthesise} \setminus \{t_a\}$
t_{b\_mustSynthesise} := $t_{b\_mustSynthesise} \setminus \{t_b\}$
if $s_{new} \not\in S_{visited} \wedge s_{new} \not\in S_{toVisit} \wedge$
$s_{new} \in \text{finalStates}(s_m) \wedge s_{2}(s_{new}) \in \text{finalStates}(s_m)$
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($P_a \cup P_b$, $s_m$)
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:** $\{\{\text{Transition}, \text{Transition}\}\}$

**Preconditions:** $s_a \in S_a$; $s_b \in S_b$

**Variables:**
- $T_{\text{pairs}} : \{\{\text{Transition}, \text{Transition}\}\}$
- $T_{\text{out}_a}, T_{\text{out}_b}, T_{\text{done}_a}, T_{\text{done}_b} : \{\text{Transition}\}$
- $P_{\text{known}} : \{\text{Party}\}$

**begin**

\[
\begin{align*}
T_{\text{out}_a} & \coloneqq \{t \in T_a \mid \text{source}(t) = s_a\} \\
T_{\text{out}_b} & \coloneqq \{t \in T_b \mid \text{source}(t) = s_b\} \\
\text{for each } t_a \text{ in } T_{\text{out}_a} & \text{ for each } t_b \text{ in } T_{\text{out}_b} \\
& \quad \text{if } \text{exchange}(t_a) = \text{exchange}(t_b) \\
& \quad \quad T_{\text{pairs}} \coloneqq T_{\text{pairs}} \cup \{(t_a, t_b)\} \\
& \quad \quad T_{\text{done}_a} \coloneqq T_{\text{done}_a} \cup \{t_a\} \\
& \quad \quad T_{\text{done}_b} \coloneqq T_{\text{done}_b} \cup \{t_b\} \\
\end{align*}
\]

\text{end if}

\text{end for}

\text{end for}

\text{P}_{\text{known}} \coloneqq P_a \cup P_b

\text{for each } t_a \text{ in } T_{\text{out}_a} \\
\quad \text{if } t_a \notin T_{\text{done}_a} \land \\
\quad \quad (\text{fromParty}(\text{exchange}(t_a)) \notin \text{P}_{\text{known}} \lor \text{toParty}(\text{exchange}(t_a)) \notin \text{P}_{\text{known}}) \lor \text{forwarding}(\text{exchange}(t_a)) \\
\quad \quad T_{\text{pairs}} \coloneqq T_{\text{pairs}} \cup \{(t_a, \text{NULL})\} \\
\text{end if}

\text{end for}

\text{for each } t_b \text{ in } T_{\text{out}_b} \\
\quad \text{if } t_b \notin T_{\text{done}_b} \land \\
\quad \quad (\text{fromParty}(\text{exchange}(t_b)) \notin \text{P}_{\text{known}} \lor \text{toParty}(\text{exchange}(t_b)) \notin \text{P}_{\text{known}}) \lor \text{forwarding}(\text{exchange}(t_b)) \\
\quad \quad T_{\text{pairs}} \coloneqq T_{\text{pairs}} \cup \{(\text{NULL}, t_b)\} \\
\text{end if}

\text{end for}

\text{return } T_{\text{pairs}}

**end**

**Function 3: transitionPairings()**

With respect to the working example, when an orchestrator is synthesised for the pair of interfaces (buyer, supplier), and the pair of states being processed is (buyer 4, supplier 4) the message exchange (supplier $\rightarrow$ 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 will only return the tuple (NULL, (supplier $\rightarrow$ shipper, ShippingRequest, false)) indicating that one non-orchestrated interaction will be added to the orchestrator FSM.
If it occurs that the set of message exchange pairings is empty, there are no messages that can be synchronously exchanged in the respective states of the orchestrator 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 automaton, 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 orchestrator automaton along with the 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 orchestrator from the two input automata is complete once both have been completely traversed.

To illustrate synthesis of a pair of interfaces with a generic example, Figure 19 shows two interfaces for two different services, ‘a’ and ‘b’. The synthesised interface built from the state (a₁, b₁) is shown in Figure 20. From the initial composite state (a₁, b₁), which is the first state visited, it can be seen there is one matching pair of exchanges (a → b, msg₁, false), one unmatched exchange from ‘a’ (a → c, msg₂, false) along the transition ta₂, and one from ‘b’ (c → b, msg₃, false) along the transition tb₂. This pairing information is discovered by transitionPairings() by returning {(ta₁, tb₁), (ta₂, NULL), (NULL, tb₂)}. Given that ta₁ and tb₁ are a matching pair of message exchanges, one transition with an orchestrated exchange (forwarding=true) is added to the interface being synthesised, and the synchronous traversal of the two automata being merged advances along both of these transitions in the input interfaces. This results in the transition having a target composite state of (a₂, b₂). This state is not placed in the queue of states to visit, because it comprises two final states and thus is also a final state. After adding ta₂ and tb₂ to the interface being synthesised, the states (a₃, b₁) and (a₁, b₃) are placed in the queue for continuing the breadth-first search since these composite states do not represent final states from both input interfaces.

These state pairs are then retrieved from the queue, as there are no other states in line to be processed. For each of these state pairs, a transition is added in the synthesised orchestrator FSM which ends with the state (a₁, b₁) which is final, so is not added to the queue of states to visit. At this point, the queue is empty and all final states have been visited, so synthesis of the interface pair is complete.
With respect to the working example, the orchestrator state (buyer 4, supplier 4) causes the pairing function to return (NULL, (supplier \rightarrow 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 21, 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 merging the shipper, it is not possible to know whether its protocol is compatible with the partner services in the choreography, so interactions involving the shipper remain non-orchestrated.
Finally, synthesise() synthesises the shipper automaton and the interface associated with $P = \{\text{buyer, supplier}\}$ to produce $P = \{\text{buyer, supplier, shipper}\}$ where all interactions are orchestrated.
3.3 Deadlock detection

If the algorithm is unable to synthesise an orchestrator from choreography services, there is behavioural incompatibility in the choreography. More precisely, there is a situation where services can no longer communicate because they cannot exchange any message and have not reached their final states. This occurrence is a deadlock and arises when a protocol must accept a message which is never sent, or can send a message which is never received. The algorithm devised to synthesise orchestrators can also identify deadlock.

Not every orchestrator can be synthesised so easily if the underlying services are not behaviourally compatible. If the shipper protocol mandated the supplier be sent a DeliveryNotice before the buyer, deadlock will occur. The transition highlighted in Figure 22 cannot be synchronously traversed since there is no matching non-orchestrated transition in the interface where P = \{ buyer, supplier \}.

The algorithm detects deadlocks within synthesiseInterfacePair() by identifying transitions in both interfaces received as input which block synchronous advancement and cause any transition not to be synthesised. Before attempting to merge any interface pair, this function creates a set of parties represented by the output interface, which are the known parties at the time of merging. For both input interfaces, transitions involving only known parties are earmarked by including them in sets of ‘must synthesise’ transitions, one for each interface. These ‘must synthesise’ transitions must be added to the orchestrator for a synchronously compatible result. Interactions involving unknown parties, or parties for which interface definitions have not yet been processed, are added to the orchestrator ‘as is’ since without the respective interface definition it is not possible to determine actual service behaviour.
As each interaction is added to the orchestrator, it is removed from the transitions that were previously earmarked. If all orchestrator states in the main queue have been processed and not all interactions involving only known parties were added to the orchestrator, deadlock is apparent.

Continuing the working example, assume the interface for $P = \{\text{buyer, supplier}\}$ is to be synthesised with the shipper interface in Figure 22. The set of known parties for this merging step is $\{\text{buyer, supplier, shipper}\}$ meaning to synthesise a compatible orchestrator, all transitions from $P = \{\text{shipper}\}$ and all non-orchestrated transitions from $P = \{\text{buyer, supplier}\}$ must be added to the orchestrator.

Synthesis is performed as expected until the shipper attempts to send a DeliveryNotice to the supplier. Since the supplier interface is already represented by the other input interface, it is known the supplier is never in a state to accept this message. Since the transition highlighted in Figure 22 is not synchronously traversed, it remains in a set of ‘must synthesise’ transitions, along with any subsequent unreachable transitions. Equivalently, the set of transitions from the previously synthesised orchestrator interface still retains the transition where the buyer receives a DeliveryNotice.

Deadlock detection therefore becomes a simple assertion that all ‘must synthesise’ transitions are synthesised.

### 3.4 Observations

The final orchestrator service is a message forwarding service. As seen in the working example by the topology involving the synthesised orchestrator in Figure 23, every message sent and received by each service in the original choreography is exchanged through the orchestrator. The orchestrator, as generated by the synthesis algorithm, controls the service composition, which then becomes an orchestration rather than a choreography, as a central service dictates what happens. The controlling performed by the generated orchestrator is more passive in nature as it purely replicates existing behaviour. The real benefits of orchestrator synthesis become apparent when the orchestrator is enhanced with additional functionality, whilst retaining backward behavioural compatibility.
Figure 23: Service topology after introducing the orchestrator

Again with respect to the working example, if the supplier made the decision whether an order should be processed by contacting an external credit check agency, how can an orchestrator be synthesised? Clearly, behavioural interfaces may not be available for all services referenced in a choreography. In this scenario, the orchestrator can be synthesised with all available interfaces, with the interactions related to external services remaining non-orchestrated in the resulting orchestrator interface. It can only be assumed that external interactions are compatible with the external services, such as between the supplier and the credit check agency.

Thus far, choreographies having only three or less parties have been considered. Under synchronous bilateral communication with three parties, one party is always left out of each interaction, so it cannot progress. However with four parties, there are two pairs which could each exchange messages independently of the other pair. Therefore the orchestrator must cater for each permutation of exchanges possible. This causes the state space of the orchestrator automaton to grow rapidly, and as a result readability of the automaton is reduced. A detailed example involving four parties with independent message exchanges is presented in Section 3.6.3.
3.5 Implementation

To validate the proposed orchestrator synthesis technique, the algorithm and associated data structures were implemented with the assistance of the Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework (GMF) which are built upon the popular Eclipse development platform. The GMF allowed a behavioural interface data model to be defined, from which object classes could be generated. A graphical designer was developed as an Eclipse plugin using the GMF to facilitate rapid creation of interfaces.

3.5.1 Interface design

The interface design tool was created by first defining the underlying domain model, which was essentially a labelled state machine. Afterwards, the GMF Dashboard tool, seen in Figure 24, was used to derive other definitions, such as the tools to appear in the design palette, and the graphical model which defined the look and feel of the display. Models produced by the GMF required substantial tweaking to remove errors and allow compilation of the graphical editor.

![GMF Dashboard](image)

Figure 24: Editor generation with the GMF

Figure 25 shows the user interface of the interface creation tool used for creating, saving and opening source interfaces, and for viewing interfaces generated by the synthesis algorithm. Interface definitions in an XML format are contained in *.interface files and graphical layout data in *.interface_diagram files, providing separation between the real data and the diagram. New states and transitions can be created by dragging the relevant item in the palette onto...
the drawing canvas. State identifiers and transition labels can be edited directly by clicking on the text.

To hide redundant details when creating interfaces, message exchanges are displayed as a string containing the other party concerned, the polarity, and the message. Polarity is ‘!’ for send and ‘?’ for receive. Each transition label is parsed and converted into a proper Exchange object when the interface is loaded. For the buyer’s protocol, the transition label ‘supplier ! Order’ as depicted in Figure 25 is equivalent to the message exchange (buyer → supplier, Order, false). Party names are entered via the properties tab.

![Interface designer](image)

**Figure 25: Interface designer**

### 3.5.2 Tooling overview

The EMF infrastructure was used to generate code to load interface specifications from files and to create the equivalent EMF representations. The generated EMF classes were not embellished with the algorithm as this would have led to source files not easily manageable, with the algorithm logic tightly integrated with the generated code. Instead, a second set of plain object classes was created, along with procedures to convert to and from objects in the EMF domain. The tooling workflow for orchestrator synthesis is shown in Figure 26, and a UML class diagram of the plain model in Figure 27. Boolean conditions for expressing choice on message content were encoded into the name of the message type. The algorithm was implemented according to its formal specification. Sets defined in the formal model
were implemented as lists to make debugging easier by ensuring predictable intermediate results, since the nature of the set does not consider element order.

Conversion procedures were written to convert between EMF objects and plain objects. For example, in the EMF model, a message exchange is a simple string, but in the plain object model, a message exchange is an object with strongly typed properties. To convert a message exchange from a string to a rich object, the string was parsed and a proper exchange object created with references to the parties and message concerned. When converting in the other direction after orchestrator synthesis, the message exchange object was serialised as a string for display in the graphical environment.

The split EMF and plain object models provided for flexible experimentation with the message exchange structure as only the plain objects and the associated conversion methods needed to be altered. Code generation provided by EMF and GMF was found to be far from flawless, so changes requiring regeneration were kept to a minimum.

Figure 26: Tooling overview
Inheritance is not shown. ElementaryState and CompositeState inherit from State.

Figure 27: UML class diagram of the plain object model\textsuperscript{5}

\textsuperscript{5} Inheritance is not shown. ElementaryState and CompositeState inherit from State.
3.6 Validation

To validate the correctness of the algorithm, choreographies from industry reference models were used as test cases. A variety of choreographies was tested, involving differing numbers of parties interacting in various topologies.

3.6.1 Two parties

The XML Common Business Library (xCBL) standard provides a document framework to facilitate global trading (xCBL.org, 2000). The standard also provides a set of two party choreographies for order management scenarios. The choreographies provide for communication between a buyer and a supplier, to exchange orders, order responses, order changes, order change responses, cancel requests, and cancel request responses. The buyer interfaces for each choreography were designed and the matching supplier interfaces created so orchestrators could be synthesised.

A choreography where two services exchange messages only with each other can be represented by either one of the service interfaces involved, as the other interface can be fully derived. The actions of messages being sent or received in one interface are mirrored in the other interface. The implementation was embellished with a function to generate a mirror interface from an input interface. So, given just one interface of a two party choreography, the orchestrator could be synthesised by first generating the mirror interface, then handing both to synthesise(). This is how the supplier interfaces were generated from the buyer interfaces prior to orchestrator synthesis.

The visual syntax of the interfaces produced by the implementation was slightly different to the syntax used for input to more accurately illustrate the structure of the underlying objects.

States in output FSMs from the implementation were labelled with the details inside the state, for example s_e(supplier, 2) signifies an elementary state from the input interface \( P = \{ \text{supplier} \} \) with an identifier of 2. Consequently, a composite state composed of the elementary states s_e(buyer, 2) and s_e(supplier, 2) is denoted by s_c(s_e(buyer, 2), s_e(supplier, 2)). This syntax proved effective when tracing through test cases to ensure the implementation was functioning as expected.

Transitions in output FSMs were labelled like \((\text{buyer} \rightarrow \text{supplier}, \text{Order}, \text{true})\) to show whether the exchange was orchestrated.
Figure 28: Example xCBL buyer FSM

Figure 29: Example generated xCBL supplier FSM (mirrored from the buyer)
While testing the implementation with the xCBL order management scenarios, it became apparent some requirements could not be described deterministically. For example, the requirement ‘if the supplier rejects the cancellation of the order’ relies on choice involving data inside the message CancelOrderResponse. Since a protocol must be deterministic, such a requirement could not be expressed using only the message type CancelOrderResponse. Two approaches were identified to deterministically express this scenario. The decision could be communicated by two fundamentally different message types, such as CancelOrderReject/CancelOrderAccept, or by externalising the expressions used for choice. The second option was used during testing to adhere as close possible to the requirements. In this example, the choice was captured in the protocol as CancelOrderResponse[cancelled=true] and CancelOrderResponse[cancelled=false]. In actuality when testing, the combination of message type and expression was used as the message name, producing the same net effect as multiple message types, yet a solution which clearly exposes logic when designing the protocol automaton.

A similar issue arose when trying to test cases where the protocol as defined in xCBL did not have a clear point of termination. For example, in one scenario, communication between the buyer and the supplier is assumed complete when the supplier does not want to send any more order responses. When defining the data structures to be used for orchestrator synthesis, it was assumed all final states have no outgoing transitions to ensure each automaton has a clear point of termination. The example here of endless order responses was
overcome by introducing a new message type, Terminate, to be exchanged after order responses are no longer required. Equivalently, this could be achieved by having an attribute in the OrderResponse message to indicate whether or not more order responses are to be expected.

Another issue which surfaced was the large number of states required in an FSM to express certain requirements, such as ‘once an order has been sent to the supplier it can be cancelled at any time.’ When designing the protocol automaton for this, extra states and transitions relating to order cancellation are required, since after every possible step involving other messages, cancellation is possible. Hence, the automata become quite unreadable and overly complex.

### 3.6.2 Three parties

Another industry reference model that was used to evaluate orchestrator synthesis in real scenarios was the Voluntary Inter-industry Commerce Standard (VICS) EDI Framework (GS1 US, 2007). This model describes message interactions in a choreography arrangement between a buyer, supplier, and carrier, to perform ordering, logistics, and payment. Various subsets of the choreography were implemented, and the respective orchestrators were synthesised.

Synthesis was successful, meaning the output was as expected, providing interfaces were specified as defined in Section 3.1. The VICS model does not describe a deterministic protocol, due to the presence of timeout messages. As shown in Figure 31, there is a point in the process where a timeout occurs when the order is not confirmed, yet no message is exchanged to communicate this event. In such a case, it is possible the buyer will not know that the supplier decided to timeout and expects the supplier to accept any message as specified in the protocol. Such an internal decision on behalf of the supplier represents a silent transition, because the state of the supplier changed in such a way that the course of future events is altered without exhibiting any externally visible behaviour. In order to deterministically express this part of the VICS model, a Timeout message was introduced to alert the buyer of a timeout decision.
3.6.3 Four parties

To explore the effects of merging more than three parties, a choreography inspired by the VICS was designed with four parties, a buyer, supplier, shipper, and airline, composed in the topology shown in Figure 32. Additionally, the airline interacts with an airport, whose interface is not described, to demonstrate how interactions external to the choreography are treated when synthesising the orchestrator.

The interfaces specified for orchestrator synthesis are shown in detail from Figure 33 to Figure 36. Briefly, the buyer initiates the choreography by sending an order for goods to the supplier, who responds with the status of the order. If the order was not processed, the choreography terminates, otherwise the supplier requests shipping from the shipper. After exchanging optional messages about the shipping status, the shipper must book the freight with the airline and optionally arrange a suitable level of freight priority with the airline. After arranging shipping, the supplier can send an invoice to the buyer. During this time, the shipper and airline must exchange confidentiality agreements, which may cause the airline to interact with the airport to request security clearance. Finally, information relating to the delivery of the goods is sent to the buyer by both the shipper and the airline.

\(^6\) (GS1 US, 2007) bs = buyer → supplier, sb = supplier → buyer
Figure 32: Four party choreography

Figure 33: Buyer FSM
Figure 34: Supplier FSM

Figure 35: Shipper FSM
Figure 36: Airline FSM
Figure 37: Partially synthesised orchestrator FSM, where P = {buyer, supplier, shipper}
The order of the interfaces in the input list was not determined by the implementation, so
was specified manually as outlined in Section 3.2. The first two interfaces to interact were
the buyer and supplier, so these two appeared first in the list. The next interface involved
was the shipper, which in turn interacted with the airline. This resulted in an input list with
interfaces represented by, in order, [buyer, supplier, shipper, airline].

In this choreography, there are interactions which can occur independently of each other
because there are more than three parties. Under synchronous communication with three
parties, only one party is left out of every message exchange, so no independent interaction
is possible. Under four, there are potentially two pairs which can interact independently, or
in parallel.

One inherent weakness of state machines is the problem of state explosion when expressing
parallelism. A state machine must provide a separate branch for every possible sequence,
leading to state proliferation and reduced readability. For example as seen in Figure 37, after
the supplier requests shipping from the shipper, either the supplier can send the invoice to
the buyer, or the shipper can book freight with the airline. These two interactions are not
dependent on each other. The state machine representation must therefore contain one branch
where the invoice is sent first before the freight booking, and another branch for the
converse. This effect increasingly impacts readability of the orchestrator logic as the number
of independent interactions rises, as evidenced in Figure 63 on page 83, where the final state
automaton is quite large considering the choreography from which it was derived.

As shown in Figure 36, there are interactions with an airport, but the interface for this party
was not available for synthesis. The final orchestrator demonstrates the algorithm can handle
such external interactions, such that the final orchestrator interface shows these interactions
as non-orchestrated, meaning the interactions are related to an unknown party. It must be
assumed such interactions are behaviourally compatible with the external party, as this
cannot be verified.
3.6.4 Deadlock demonstration

To evaluate deadlock detection in the implementation, various tests were carried out by adding, moving, or removing interactions from interfaces to intentionally create deadlock scenarios. One such evaluation was conducted by altering the protocol of the shipper so that if delivery details were not confidential, the shipper not only sent the delivery notice to the buyer as intended, but also to the supplier, as illustrated in Figure 38. This alteration caused deadlock because the supplier can never be in a state to accept a delivery note from the shipper. This means when the shipper interface is in state 8, communication cannot continue and is deadlocked.

Since the input list to synthesise() contains the parties in the order [buyer, supplier, shipper, airline], deadlock was not detected until the partially synthesised orchestrator interface, where P = {buyer, supplier} was synthesised with the shipper interface. Before attempting to merge these two interfaces, the function synthesisePair() identified the ‘must synthesise’ transitions, which included all transitions related to message exchanges between the buyer, supplier, and shipper. Although all transitions from the partially synthesised orchestrator could be synthesised, this was not the case of the transition in the shipper interface where the shipper attempts to send the delivery notice to the supplier, as highlighted in Figure 38, plus

![Figure 38 Shipper FSM which caused deadlock](image-url)
all subsequent unreachable transitions. Therefore, after this function attempted to merge the interfaces, not all ‘must synthesise’ transitions were added to the orchestrator interface, so deadlock was evident. Upon discovering the deadlock, the implementation threw a custom exception and terminated synthesis, saving the orchestrator interface as is, which assisted in confirming the algorithm performed deadlock detection as expected.

### 3.7 Summary

Orchestrator synthesis from choreographies requires behavioural interfaces for participant services to be defined. A simple language is proposed to define behavioural interfaces where FSMs represent messaging protocols. The language allows for choice based on message contents to be externalised ensuring protocols cannot be defined non-deterministically. A synthesis algorithm was presented to generate orchestrator interfaces in the simple language, and explained with a working example of a three-party choreography. The algorithm is intended for use in the context of a messaging environment without buffering where message transfer is instantaneous. The algorithm can detect deadlock. An example was provided to illustrate how deadlocks can be found.

Validation of the algorithm and associated data structures was carried out by testing industry reference models with tools implemented with the Eclipse development platform. Testing revealed some issues with protocol specification, such as timeouts, and how such business requirements could be expressed with the simple language for defining protocols to ensure predictable service behaviour. Although the highest number of participants in any given choreography was four, no limitations were identified that would prevent the synthesis of orchestrators from an unlimited number of participants.
4 Readable service behaviour modelling

State machines are capable of modelling service behaviour, but with one major drawback: state explosion may render state machines useless for fast and accurate human interpretation. This is especially true for orchestrator services as they contain the product of logic from all parties, which may lead to combinatorially complex state diagrams. To address this problem, state machines could be visualised using a reader-friendly modelling notation having a larger vocabulary to express service behaviour succinctly. As discussed in Section 2.4, BPMN diagrams offer a user-friendly way of modelling business processes in general, and orchestrations in particular. Since BPMN also supports concurrent activities explicitly, it does not suffer from the state explosion issues associated with FSMs.

A transformation pathway is presented here whereby finite state machines can be depicted as BPMN diagrams. The path is shown in Figure 39 and involves transforming the state machines into Petri nets, then into BPMN diagrams.

Figure 39: Transforming state machines into BPMN diagrams
4.1 State machines to Petri nets

The inherent state explosion problem of state machines arises from the fact that state machines are sequential in nature. A state machine must contain an explicit path for every allowed interleaving of interactions, even for interactions which are independent of each other and can occur in any order. As already explained in Figure 8, a state machine can be mapped trivially to a Petri net. Essentially, every state in the state machine becomes a place in the Petri net, and every transition in the state machine becomes a transition having one incoming and one outgoing arc in the Petri net. The resulting Petri net is not more readable than the original state machine, but since Petri nets support concurrency, it then becomes possible to transform the Petri net into an equivalent, more compact one, by identifying regions of concurrency and collapsing them, as shown in Figure 9.

Collapsing regions of concurrency requires elaborate techniques not part of the research project. Fundamentally though, regions in sequential models representing concurrent interactions are discovered by finding a complete set of permutations/interleavings of a given set of interactions, and reducing each region to transitions in Petri nets which can be executed in parallel. A body of work known as theory of regions has addressed the issue of identifying concurrency regions in Petri nets and exploiting them to produce more compact Petri nets. This thesis does not endeavour to delve into the workings of the theory, and treats concurrency identification as a black box method. This is achieved by using the Petrify tool which encapsulates various techniques derived from theory of regions. In this way, a Petri net with concurrent interactions describing the orchestrator corresponding to the working example can be created, and this is evidenced in Figure 53.
4.2 Petri nets to BPMN diagrams

Transformation from Petri nets to BPMN diagrams is achieved with rules-based pattern matching. A set of rules is presented where each rule identifies a pattern in the graph to be replaced by the output as specified in the rule. The rules do not include particulars of how to generate pools and message flows, as such details are not affected by the business logic of the service to be modelled.

The logic for the Petri net being transformed is contained inside one pool, namely the logic pool, and the pools of all other parties are treated as black boxes. The reason for this is to show the behaviour of the service being modelled and abstract away from the internal behaviour of the other parties. Therefore message flows are attached to tasks in the logic pool, and to edges of black box pools.

![Figure 40: BPMN diagram with a logic pool and a black box pool](image)

The transformation presented here does not use the complete vocabulary of the BPMN specification. Exceptions, cancellation regions, timeouts and sub processes are not supported. The transformation is intended to yield simple and concise BPMN diagrams. To this end, the message forwarding action, being abundant in orchestrators, is captured as a single task in BPMN, rather than the more verbose juxtaposition of a start or intermediate message event followed by a send task, as in Figure 41. To maintain consistency and simplicity, BPMN diagrams for non-orchestrators are generated without intermediate message events, having a stand-alone task to capture message receipt.
The nine transformation rules presented below produce a BPMN process definition for the logic pool. A rule defines a pattern, which is a region of a directed graph bound by nodes, and an output, which specifies how the pattern should be transformed. A rule has been fully applied when all instances of its pattern have been replaced with its output. Some rules cannot be applied until other rules have been fully applied, leading to the rule dependencies shown in Figure 42. Rules with no dependencies, indicated by the absence of incoming arrows in the rule dependency diagram, do not need to be fully applied before other rules can be applied. For example, instances of the rule 4 pattern can be transformed before all instances of the rule 3 pattern are transformed. Transformation is complete when no more rules can be applied.

The goal of devising the rule set is to communicate clearly the requirements for performing the transformation. Dependencies are present to reduce the complexity of pattern definitions. Their need could be suppressed by making the patterns disjoint from one another, so that at any point in time, only one rule can be applied to a given element or fragment of the input diagram. However, this would make the rules more complex. The approach taken here aims to minimise dependencies to give flexibility to an implementation of the rules, whilst...
promoting succinct rule definitions. Rule numbering is for discussion purposes only and does not imply order—ordering is entirely determined by the dependency diagram.

The moment a Petri net is modified by any rule, it is no longer a valid Petri net definition. The graph is changed into a partially transformed diagram, containing a mix of Petri net and BPMN objects. In the rule definitions, directed links between nodes are referred to as arcs, and it is implicit that arcs become BPMN sequence edges. This conversion is not captured in any rule; it occurs implicitly when the transformation is applied.

The rules are explained using snippets derived from the working example, and demonstrate how patterns are transformed into BPMN for the logic pool. It is assumed other parties referred to in the interactions are represented by separate black box pools and linked to the logic pool via message flows. In the figures explaining each rule, each alpha label and arc pair represents one or more connected objects from the Petri net or BPMN domains, to show how the output of the rule replaces the identified pattern.
Rule 1: Add start events
A place having no incoming arcs indicates a potential starting point for the process. When modelling service behaviour for a well-designed protocol, it is expected there will be exactly one place to indicate where the process commences. However this rule can be applied to graphs having more than one such place.

Dependencies  None.
Pattern  A place with no incoming arcs.
Output  A BPMN start event.

Figure 43: Mapping a place to a start event

Rule 2: Add end events
Termination points in the graph are present where a transition has no outgoing arcs along which flow can continue, or has one outgoing arc leading to a place with no outgoing arcs.

Dependencies  None.
Pattern  A terminating transition, with an optional output place.
Output  A BPMN end event with an arc going from the transition to the end event.

Figure 44: Adding an end event
Rule 3: Remove places
Petri net transitions must be connected through arcs to places, but in BPMN, tasks can be connected directly, so some places can be removed altogether.

**Dependencies** None.

**Pattern** A place with one incoming and one outgoing arc.

**Output** A single arc connecting the two bounding nodes.

![Diagram](image1.png)

**Figure 45: Removing a place**

Rule 4: Map places to XOR merges
Places having multiple incoming arcs, but only one outgoing arc, indicate exclusive joins, where processing occurs along any given incoming arc.

**Dependencies** None.

**Pattern** A place having more than one incoming arc, and one outgoing arc.

**Output** An XOR merge gateway.

![Diagram](image2.png)

**Figure 46: Mapping a place to an XOR merge gateway**
Rule 5: Map places to event-based XOR decisions
A place having multiple outgoing arcs indicates exclusive choice, which can be either data-driven or event driven. This rule is concerned with event-based XOR gateways where the process can receive one of multiple messages of differing types. The data-driven XOR decision variant is defined in rule 8.

Dependencies None.
Pattern A place having multiple outgoing arcs for different message types.
Output An event-based XOR gateway.

Figure 47: Mapping a place to an event-based XOR gateway

Rule 6: Add AND forks
Parallel processing in the graph commences where a transition has more than one outgoing arc. Each arc can be processed as a concurrent thread.

Dependencies None.
Pattern A transition with multiple outgoing arcs.
Output An AND fork gateway as the direct successor of the transition, with the transition’s original outgoing arcs emitting from the fork.

Figure 48: Adding an AND fork
**Rule 7: Add AND joins**

The conclusion of a region of concurrency is indicated by a transition having more than one incoming arc.

**Dependencies** None.

**Pattern** A transition with multiple incoming arcs.

**Output** An AND join gateway as the direct predecessor of the transition, with the transition’s original incoming arcs entering the join.

![](image)

**Figure 49: Adding an AND join**

**Rule 8: Map data-based XOR decisions**

The Petri net contains separate transitions for each possible path of execution and this can lead to a scenario where the receiving or sending of a single message is represented across several transitions, each with a different encoded Boolean condition. Such multiple representations must be combined into a single task to perform the message action, with each condition forming a different branch of exclusive choice when the condition is applied to the message data. So for each message, a single task is produced, followed by the complete set of conditional branches. It is assumed conditions are mutually exclusive, as priority is not assigned to the order of condition evaluation.

**Dependencies** Rules 2, 6 and 7, to ensure each transition has exactly one incoming and one outgoing arc.

**Pattern** Transitions having the same message type being direct successors of a place or event-based XOR gateway. Note that after fully applying rule 5 to the graph, the place in Figure 50 would remain, but the place in Figure 51 would be transformed into an event-based XOR gateway.

**Output** A single task to represent the message action (receive/send/forward) followed by a data-based XOR gateway, with arcs labelled with the message conditions. If all transitions refer to the same message type, the preceding place is removed.
Rule 9: Map transitions to tasks

A transition transforms directly to a task, but not if there is data-based choice as specified in rule 8, where multiple transitions can collapse to one task. After rule 8 has been applied, all remaining transitions can be mapped one-to-one to tasks. If the service being modelled is an orchestrator, all tasks will perform forwarding, otherwise they will perform sending or receiving.

Dependencies  Rule 8, to ensure transitions map one-to-one to tasks.

Pattern  Any transition.

Output  If the transition is empty, an arc, otherwise a task.

---

7 All transitions have one outgoing arc after applying rule 5.
To demonstrate these rules in more depth, the full working example is shown at various stages of transformation. For brevity, rules are fully applied in the order defined. Figure 53 shows the Petri net as created by Petrify, without any BPMN elements. Figure 54 shows the partially transformed graph after mapping the start event, adding the end events, removing unnecessary places, and mapping places to XOR gateways. Figure 55 displays the effect of applying rules 6 and 7, where AND forks and joins are explicitly shown. Figure 56 reveals the final BPMN diagram, after all rules have been fully applied. Message conditions are externalised to conditional branches of data-based choices, and all transitions are forwarding tasks.
Figure 53: Initial Petri net

Figure 54: Graph after applying rules 1, 2, 3, 4 and 5
Figure 55: Graph after applying rules 1, 2, 3, 4, 5, 6 and 7

Figure 56: Final BPMN diagram after applying all 9 rules
4.3 Implementation

Generation of readable process models was put into action with an implementation to transform finite state machines to BPMN diagrams. This was achieved by chaining two distinct software tools together. The first one, called Petrify, was used to identify concurrency in state machines and produce Petri nets with concurrent regions. The second one was an implementation of the rules-based transformation presented in Section 4.2. Following the pathway in Figure 57, a service described using state machines as presented in Section 3.1 could be visualised as a BPMN diagram.

![Figure 57: State machine to BPMN tool chain](image)

4.3.1 State machines to Petri nets

Petrify enables a transition system labelled with message interactions to be converted to a Petri net where transitions are labelled with the same message interactions. Although developed as a command line tool, Petrify is invoked via libraries from an open source project, ProM, which is aimed at process mining and analysis. ProM provides API for invoking Petrify, where an object representation of a state machine is converted to a Petri net object as determined by Petrify, given a set of operational switches. This eliminates the task of writing input files for the command line tool and parsing the output files into objects as required for further processing.

Section 2.4 outlined the requirement that a Petri net to be transformed into BPMN must be a free choice net. This requirement is implemented by ensuring the relevant switch is set prior to invoking Petrify. Once a Petri net is obtained, it is exported via ProM in DOT, a plain-text graph description format (Gansner & North, 2000). The DOT file is displayed within the Eclipse environment to verify the output from Petrify. The Petri net for the working example orchestrator arranged with DOT is shown in Figure 64 on page 84. Petri nets from Petrify contain transitions with no outgoing arcs, but this is semantically identical to each such transition having a single terminating output place.
4.3.2 Petri nets to BPMN diagrams

A second open source initiative, the SOA tools platform project, aids building extensible tools to support software architectures based on services. The BPMN subproject of the platform provides an eclipse editor and full API support for manipulating business process models in BPMN. Process models created from the provided API are saved to disk and opened with the BPMN editor for display. Disappointingly, the editor’s support for automatic layout of the process diagrams is rudimentary, but nonetheless still saves some time manually arranging diagram elements. The BPMN diagram generated by the implementation for the working example is shown in Figure 65 on page 84.

The transformation from a Petri net to a BPMN diagram is achieved with an algorithm implemented with a visitor pattern, which traverses the Petri net breadth-first and incrementally builds up the BPMN diagram in a separate container. The visitor pattern is realised with a queue which allows each Petri net place to be visited once. For each place visited, the outgoing transitions are grouped based on message type to identify data-driven choice. For each group, and therefore message type, a task is added to the BPMN model to capture the message interaction, and each transition in the group is traversed to find subsequent places for visiting. Mapping dictionaries of places to XOR gateways, and of transitions to AND gateways enables BPMN nodes to be found for attaching further elements when a Petri net node is visited.

The transformation was specified in a rule-based manner in Section 4.2, and the implementation honours the rules, demonstrating that an implementation was achievable. Since the implementation deals with two separate graphs, rather than a single graph transformed incrementally, XOR merge gateways cannot be determined without investigating the outgoing arcs of a place before adding it to the queue to be visited. To avoid looking another degree ahead into the graph, each place is mapped to an XOR gateway, even a place preceding a data-based XOR decision where only one message type is present. The XOR gateways not performing decisions or merges are simply removed by a post-processing cleanup routine after all rules have been applied.

The cleanup procedure also removes empty tasks, which may enter the BPMN diagram due to empty transitions in the Petri net, or which are immediately preceding AND join gateways. To save looking ahead in the graph during the transformation while visiting a transition group, tasks are created as soon as grouping has been performed. When a transition with more than one incoming arc, representing an AND join, is encountered in the
traversal for the second or subsequent time, an associated AND gateway already exists in the
BPMN diagram, so only at that point in time can it be determined the transition should not
translate to a task. The task is therefore flagged for removal by making it empty, and the
outgoing sequence edge is attached to the existing AND gateway.

Tooling developed generates pools, and optionally, message flows. Logic pools are created
for all diagrams to provide a container for process logic, but message flows are found to be
practical only when generating diagrams for non-orchestrator services, as the mass of
message flows produced by a multitude of forwarding tasks produces messy layouts. The
behaviour of the buyer service from the working example is shown in Figure 58.

### 4.3.3 BPMN readability optimisation

BPMN readability was found to be affected by the underlying Petri net structure, as the same
service behaviour can be represented by different Petri nets. One example where readability
was impacted was when two transitions, each having two outgoing arcs, were attached to the
same two places, as shown in Figure 59. Although semantically correct, the resulting BPMN
diagram produced by the implementation, seen in Figure 60, is not concise as possible. This
phenomenon is caused by Petrify opting for a minimal-place Petri net (Cortadella et al.,
1997), which reflects the original intention of the tool to be used for optimising circuitry.
The two AND fork gateways are followed immediately by two XOR merge gateways, so at
first glance without tracing the sequence edges, the two subsequent tasks do not look as
though they can occur in parallel. A second example where readability is reduced is the
presence of two end events instead of one, as seen in Figure 60, which may not be very
intuitive for some users. These figures show how Petri nets are displayed with DOT, and the
process models with the STP BPMN editor.

To improve BPMN readability, either the transformation rules could be embellished with
logic to optimise the BPMN layout, or the input Petri nets modified to alter the
transformation result. The former option was not pursued as the rules would have become
inflated with logic not concerned with the semantic transformation. The latter option was
trialed by altering the structure of Petri nets without altering the semantics. To experiment
with solving the problem of murky logic surrounding the parallel region in Figure 60, the
source Petri net was manually altered so empty transitions explicitly marked where the
region commenced and terminated. Figure 61 illustrates the Petri net optimised for improved
readability. While this Petri net is more verbose, the resulting BPMN in Figure 62 is more
readable and concise.
Figure 58: Generated diagram for a non-orchestrator with pools and message flows
Figure 59: Petri net derived from the orchestrator, showing structure chosen by Petrify

Figure 60: BPMN diagram for Figure 59
Figure 61: Modified Petri net optimised for cleaner BPMN generation

Figure 62: BPMN diagram for Figure 61
4.4 Summary

FSMs are suitable for modelling service behaviour, but suffer from state explosion when dealing with concurrent interactions. Readability can be improved by drawing on the theory of regions to produce Petri nets with concurrency represented more compactly. To increase readability for a business audience, a rules-based transformation was devised to translate Petri nets into business process models in BPMN. Mostly needed for visualising logic for orchestrator services, the transformation presented performs translation for non-orchestrator interfaces, such as for participants in choreographies.

Tooling was developed to support the rules-based approach with the help of open source tools, to control the Petrify tool for concurrency identification, and to create BPMN diagrams for viewing in a graphical editor. The Petrify tool created minimal-place Petri nets which do not always result in clear logic in BPMN process models. Logic in generated BPMN models could be made more intuitive if the Petri nets prior to transforming to BPMN are augmented with empty transitions to explicitly mark boundaries of concurrent regions.
5  Conclusion

Orchestrator synthesis provides a bridge between the two service composition viewpoints of choreography and orchestration. Synthesis is achieved by defining the behaviour of the participant services in choreographies and synthesising the behaviour of the orchestrators from those descriptions. Synthesised orchestrators coordinate the participants of a choreography, so that rather than interacting in a peer-to-peer manner, they interact through a single point of control, which provides a place for introducing additional features such as value-adding mediation into the service composition, or to change the behaviour of the service composition in light of evolving requirements.

5.1 Contributions

The contributions of this research are:

- A simple language to specify service messaging protocols. Existing standards allow ambiguous non-deterministic protocols to be defined. The proposed language is fully deterministic and supports control flow choice based on message contents.
- An algorithm to synthesise orchestrators from choreographies, using the simple language. Services from choreographies defined in the simple language can be synthesised to form behavioural interface definitions for orchestrators. The algorithm synthesises pairs of input interfaces, and can detect deadlocked protocols.
- A rules-based approach to transform services defined in the simple language into BPMN business process models, where value-adding mediation could be configured, and where processes could be translated into an executable language such as BPEL. Concurrency expressed in the simple language gives rise to verbose representations not easily readable. Rules were devised where patterns in Petri nets are identified and mapped to BPMN, for a business audience. While effective for visualising logic
for orchestrator services, the proposed transformation is applicable to non-orchestrator services, such as participants in choreographies.

5.2 Future work

The absence of a current web service standard for defining deterministic protocols is a hurdle to service compatibility. Future developments in standards for service behaviour description should consider the importance of determinism and how interfaces should be crafted by developers to achieve behavioural interface compatibility by design.

The orchestrator synthesis algorithm could be extended to expand its application to deal with messaging environments with queues and transmission latency. Consideration could also be made for multilateral message exchanges, such as broadcast, multicast, and publish/subscribe message exchange patterns. Specifically, a strong assumption is made in this work that only one service can be bound to a role in a choreography. With respect to the working example, there is only one buyer, one supplier, one shipper, and so forth. In realistic scenarios, it is possible that more than one service can be bound to a role in the context of one execution of a choreography. An example of this would be where a buyer contacts multiple suppliers to request quotes, or a supplier has a choice between multiple shippers.

Logic in BPMN diagrams produced by direct translation from Petri nets could be improved by ensuring Petri nets contain explicit boundaries for regions of concurrency. A formalisation could be developed for automating the pre-processing of Petri nets to optimise BPMN logic clarity, being a crucial factor for business users.
Bibliography


Figure 63: Fully synthesised orchestrator, where $P = \{\text{buyer, supplier, shipper, airline}\}$
Figure 64: Petri net generated by Petrify for the working example, arranged with DOT

Figure 65: BPMN diagram generated for the working example orchestrator