Computing (2013) 95:567–610
DOI 10.1007/s00607-012-0237-5

A design methodology for verified web-service mediators

Jing Cao · Albert Nymeyer

Received: 24 March 2012 / Accepted: 6 November 2012 / Published online: 23 November 2012

Abstract A new web service can be built by combining existing web services and coordinating their actions by a 'mediator'. This work presents a formal design methodology based on model checking that generates all the mediator designs that meet requirements specified by a user. The methodology allows the user to explore the designs in search of the mediator that offers the best theoretical performance. Markov models of various measures of performance are considered. Each measure may result in a different ranking of the mediator designs (from best to worst for example). From these rankings the user can select the design that at least theoretically best fits the purpose. The novelty of this work is the use of model checking and Markov analysis in a single formal framework to generate, explore, and select from a set of provably-correct mediators—while still in the design stage of development.

Mathematics Subject Classification 68N30 · 68U07 · 65C05

1 Introduction

Web services are distributed software components that solve specific tasks varying from responding to simple requests to complex business processes [25,20]. They are typically designed to interact with other web services. Large applications can be built by *composing* existing component web services. This component-based approach to software development can significantly impact the way commercial applications are

J. Cao (✉) · A. Nymeyer
School of Computer Science and Engineering, The University of New South Wales,
Sydney, Australia
e-mail: jcao@cse.unsw.edu.au

A. Nymeyer
e-mail: anymeyer@cse.unsw.edu.au

developed [40]. Just as in all software development, when composing web services, design (or architectural) decisions often need to be made early in the development process. The consequences of these decisions in functional terms (i.e. whether the requirements will be satisfied) and non-functional terms (e.g. Will the performance be adequate?, Will the system be usable?) are often unknown at the time the decisions are made. The design must first be implemented and tested. If bad decisions are made, correcting the design after the implementation stage has been reached can be very expensive, or even impossible. Clements and Northrop [17] put it succinctly: "whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen".

The analysis of a high-level specification of a web service is an attempt to anticipate at an early stage in the software-development cycle the impact that design decisions will have on the non-functional and functional requirements. Non-functional requirements may include factors such as speed, response time, size and power consumption. The functional requirements define behavioural correctness, where 'correctness' not only refers to the behaviour, but also refers to the system's robustness, to whether the system is deadlock-free and to more specific properties such as whether the system is guaranteed to respond to every query. An even more fundamental question is, given a set of component web services, does a mediator exist that meets the behavioural and performance criteria set by the user.

Verifying behavioural correctness at the design stage requires the system to be formal modelled [40]. This involves specifying the components mathematically, and composing these specifications. If model checking is used, functional properties may also be specified and these can be verified. However, if additionally the user wishes to have some confidence that the design has the best possible performance, then there is no technology available. Formal methods such as model checking allows qualitative, i.e. functional, statements about the behaviour of a system to be made, but in general not quantitative, i.e. non-functional, statements. These same issues face most developers of complex, large-scale software systems that are built using component reuse. In particular they are faced by computer engineers and hardware designers [34], where the demands for both functional correctness and high performance can be 'mission critical': a mistake can lead to financial disaster, and may cost lives. For example, the aviation industry, the space industry and medical-equipment manufacturing use technologies where there is simply no margin for error.

The underlying theory of the approach that we present here was originally developed in the field of hardware design [8–10,12]. In hardware design (of VLSI circuits for example), a *protocol interface* is *synthesised* from a formal specification. The role of the interface is to match different, autonomous, off-the-shelf component protocols. The interface that is synthesised interacts with the components to produce behaviour that is specified by the user. In web services, a *mediator* is synthesised instead of a protocol interface, and the components are existing or standard web services. While the overall approach to synthesising a protocol interface and a mediator is similar, there are significant differences. For example, a model of a mediator takes into account both *local* web services, which are connected directly to the mediator, and *remote* web services, which are connected to the local services but not the mediator itself. There is no analogy to this in hardware design. The definition of performance is also

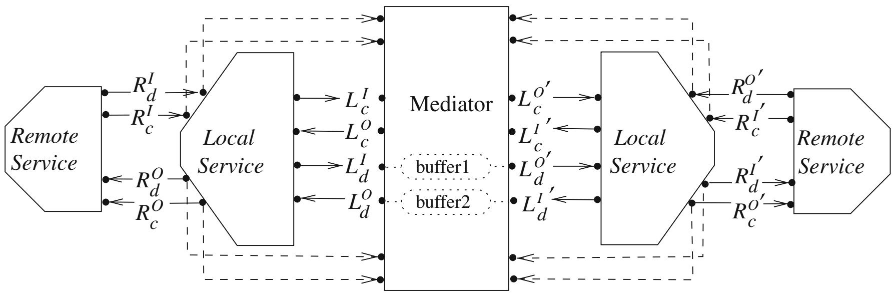


Fig. 1 A model of a mediator for local and remote web services

different: we have chosen two commonly-used web-service performance measures [42], response time and data throughput, but also a less-traditional measure, the energy cost (it abstracts the electrical cost of providing a web service), which is a performance consideration most often associated with portable devices and remote equipment. Each measure of performance is analysed using Markov chains, and it is this analysis that drives the process of selecting the best-possible theoretical mediator design.

In Fig. 1 we illustrate our model of a mediator for web services. The mediator controls the communication between services that are directly connected to it. These are called *local services*. For simplicity we show just two in this figure. Other services that are not connected to the mediator but are connected to the local services are referred to as *remote services*. Again, we show just two for simplicity. All interactions are bi-directional. The solid arrows in the figure are called *transactions* and involve the transfer of data and control signals. The dashed arrows do not involve data transfers: they are a mechanism that enables the mediator to observe the communication between the local and remote services. They play an important role in verification (only).

We differentiate in the figure between local input control transactions, denoted L^{I_c} and $L^{I'_c}$; local input data transactions, denoted L^{I_d} and $L^{I'_d}$; local output control transactions, denoted L^{O_c} and $L^{O'_c}$; and local output data transactions, denoted L^{O_d} and $L^{O'_d}$. Similarly for the transactions between the local and remote services. Data transactions are simply message transfers. The contents of these messages do not change the flow of control in the mediator. Control transactions in contrast generally cause the mediator and services to take actions. Examples are sending a request to a travel agent for a flight or to a hotel for a room. The important issue here is that the behaviour of the mediator is driven by control signals (or messages if you prefer), and the data transfers play a subservient role. Note that the input/output direction of the transaction is relative to the mediator: all input transactions are in the direction towards the mediator, and all output transactions are in the direction away from the mediator.

There are two buffers in the mediator in Fig. 1 that are connected to the data channels. Intuitively, a buffer is just memory that may be used in a data channel to store data. For example, a service may wish to send a request of length ten characters to another service that is able to accept requests of length only five characters (the control signals of the two services hence do not match). The mediator that acts as intermediary between the services will send the first half of the request immediately, and store the

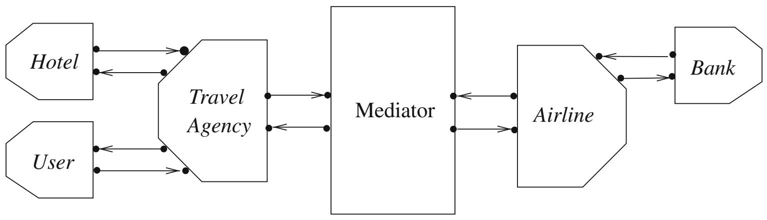


Fig. 2 A mediator for local services *Travel Agency* and *Airline*, which are in turn connected to remote services *Hotel*, *User* and *Bank*

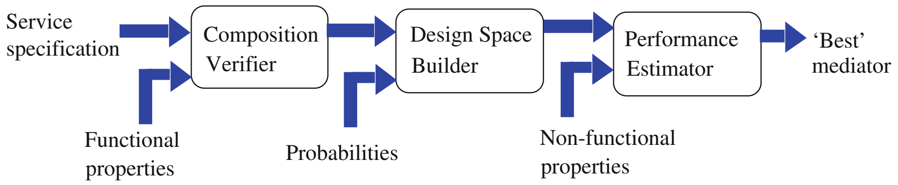


Fig. 3 The architecture of the EVERESTdesign methodology for web-service mediators

second half in the buffer to be sent later. In general, every channel that involves the transfer of data would be expected to have a buffer. In this work we formally describe the transfer of data to and from the mediator through buffers, and we also formally model the cost of these transfers in terms of performance. For example, we compute formally the cost in energy of transferring data, which requires the amount of data that is transferred to be modelled.

Example 1 In Fig. 2, we show an example of a mediator for local services *Travel Agency* and *Airline* [25]. The *Travel Agency* is in turn connected to remote services *User* and *Hotel*, and the *Airline* to remote service *Bank*. We have simplified the transactions in the figure by not showing control and data separately and we have omitted the connections that allow the mediator to observe the remote actions (the dashed arrows in the previous figure) to keep the picture simple. It is understood that all the interactions between the local and remote services can be observed by the mediator. The purpose of the system is straightforward: the *Travel Agency* books airline flights and hotels for users. The *Airline* is connected to a bank that must confirm that payment has been received.

In essence, the aim of the mediator is to translate all control and data actions from one local service into control and data actions that are understood by some other local service. In this work, we present a design methodology for mediators, which we call EVEREST.

The architecture of this design methodology is shown in Fig. 3.

The methodology consists of three phases:

Verifier This phase constructs a finite-state machine that represents all the behaviours. It does this using an 'on-the-fly' model checker. The resulting state machine is not only provably correct, it is guaranteed to satisfy any functional properties specified by the user.

Builder In the second phase, a so-called *design space* of mediators is built from the state machine generated by the first phase. *Markov probabilities* are introduced at this stage. These probabilities reflect the range of behaviours in which the mediator is expected to perform.

Estimator In the third phase, the non-functional properties that specify the desired minimum performance and the Markov probabilities are used to select the theoretically best mediator in the design space. This may require the user to trade-off competing performance measures: for example, the response time and the size of the system.

1.1 On-the-fly model checking

In the Verifier phase, a state machine that is a composition of the component services and the functional properties is constructed. This state machine describes all the possible behaviours, all of which are guaranteed correct. The functional properties that the user requires are specified using computational tree logic (CTL) by the user. This CTL specification defines the ‘correctness’ properties for the model checker. The model checker is based on the *tableau method*, which decomposes the given CTL formulas into smaller formulae [5,16] and used to label the (verified) state machine. In the process of generating the state machine, any state and transition that violates a correctness condition is removed. If no state remains then no composition exists. Otherwise, a formally-verified state machine is constructed.

1.2 A Markovian design space of mediators

The single, verified, state machine generated by the model checker is a superset of all possible behaviours. For example, from some state, different transitions to new states may be possible, which, while all correct, may result in different performance. To generate a particular mediator, we need to split states. This process of splitting leads to a set of all the possible individual mediators, which we call the *design space* of mediators. Each design is modelled using *Markov chains*.

Markov chains have been widely used for about a century for performance and dependability evaluation of computer systems. Characteristic of a Markov chain is, given the current state, the future evolution of the Markov chain is totally described by the current state, and independent of past states. This is called the *Markov property*. To use Markov chains, so-called single-step probabilities are required. These are used to compute the *steady-state probability* of each state, which is the probability that the system will be in a particular state in the infinite limit of time (see [22] for details). A classical method of computing the steady-state probabilities is to solve a set of equations, called the *Chapman-Kolmogorov equations*. Based on these steady-state probabilities, we can then compute the *steady-transition probabilities* (see [11] for calculation details). These steady-state and steady-transition probabilities are computed for every design.

1.3 Selecting the 'optimal' design

Traditionally, design decisions are made in the early stages of a development. Towards the end of a development, when the implementation is undergoing testing, the consequences of these decisions may become evident. There is only one implementation of course; the engineer will never know whether implementing another design would have resulted in a better system. In testing, an engineer executes the implementation to determine its performance. The test cases chosen ideally mimic real-life scenarios, and large numbers of test cases would typically be used. Often this stage of the development is the most time-consuming, error-prone, and mundane of the whole software development, and of course, as we are dealing with the actual implementation, it is also a stage that comes very late in the development: generally too late to correct earlier design decisions without substantial cost.

In our work, instead of an implementation we have a set of mediator designs. We analyse the theoretical performance of each of these designs using a Markov analysis. In essence, we make a prediction which designs would perform the best. We cannot claim the predictions will be accurate in an absolute sense: the actual performance of the web service will be affected by many implementation details that we cannot anticipate. What we do claim is that there is likely a correlation between the ranking of the designs in terms of theoretical performance and the rankings of the implemented systems *if* each of the designs were to be implemented. One basis for this claim is the evidence from the field of hardware design. The Markov analysis models typical user behaviour, which is something rare in software design: user behaviour is generally reflected in the test cases that are applied after the implementation have been completed. User behaviour is one of the inputs of the design process in this paper, and is used to select the best theoretical design. There is no cost to abandon one design in favour of another in our approach. In fact, the trade-off in the various measures of performance between all the designs plays an important role in motivating our methodology.

The Markov model, while theoretical, is based on probabilities that reflect real-world behaviour. Determining these probabilities is similar to determining which simulations or which test cases to use, but it has significant advantages:

– Once the probabilities are set, the ranking of the designs is completely determined.
– The Markov analysis occurs at high level, before any development has taken place. So it is trivial to change the design.
– Probabilities can be standardised to reflect 'accepted practice'.
– If user behaviour is observed or predicted to change, it is straightforward to re-analyse all the designs to see which offers the best theoretical performance.

1.4 Optimal performance

The functional properties are 'hard and fast' behavioural requirements that the mediator design should satisfy: they determine correctness. There is no notion of correctness in the non-functional, performance requirements of the mediator. These requirements are used to compare and rank the mediator designs. There are many ways to measure the performance; in this work we use the following:

Energy consumption This is the amount of energy required to carry out the set of transactions necessary to carry out the service. For example, the service may be the travel agent booking both a flight and a hotel for a user. The result of the service will be the flight ticket, or an indication that no flight, or no hotel, is available. Given a Markov model from the Builder phase, we can model the power q^{pow} consumed by the system as [11,12]:

$$q^{pow} \propto \sum_{s_i, s_j} \mathcal{W}_{i,j} \mathcal{H}_{i,j} \tag{1}$$

where the sum is over all adjacent pairs of states s_i and s_j of the Markov model, $\mathcal{W}_{i,j}$ is the weight of the transition between states s_i and s_j, and $\mathcal{H}_{i,j}$ is the Hamming distance between this pair of states. The state-encoding algorithm that we applied here can be found in [11]. Using this definition of power, the energy consumed by the service can be computed.

Data throughput Examples of data that must flow through the buffers are user data (such as personal details), hotel details, flight details and dollar amounts. The data throughput is the average amount of data that is transferred in a single time unit. To compute the Data Transfer Rate (DTR), we use the formula [11,13]:

$$\text{DTR} \propto \sum_{s_i, s_j} \mathcal{T}_{i,j} D_{i,j} \tag{2}$$

where $\mathcal{T}_{i,j}$ is the steady-transition probability of a transition from state s_i to s_j, and $D_{i,j}$ is the amount of data that has been transferred in this transition.

Response time Services generally consist of some number of request-response transaction pairs. Typically, the mediator receives a request from some local service, which will require it to send requests to other local services, each of which take time to respond to. Eventually the mediator will respond to the original request. In this work we formally define the sum-total of transactions necessary to respond to a request by a *dialogue*. The times it takes for the mediator to execute the dialogue is called the *response time* (denoted as $\mathcal{R}$). The response time of a request is calculated in this work using the formula:

$$\mathcal{R} \propto \sum_{s_i, s_j} \mathcal{T}_{i,j} n \tag{3}$$

where s_i and s_j are states in the dialogue, $\mathcal{T}_{i,j}$ is the steady-transition probability of the transition from state s_i to s_j, and n is the total number of transitions.
Any given request may lead to different responses (depending on circumstances such as availability and cost for example) so to determine the response time the average of all possible dialogues is calculated, as well as the standard deviation to see the scattering. There are clearly many ways of computing the average that could include for example the likelihood of particular responses.

1.5 Outline

In the next section we describe related work. In Sect. 3, we explain the Verifier phase; in Sect. 4 the Builder phase, and in Sect. 5, the Estimator phase. Fragments of the travel-agency example shown in Fig. 2 will serve as a running example in this work. This case study is considered in its entirety in Sect. 6. In Sects. 7 and 8 the contribution of this work, future work and a perspective are provided.

2 Related work

Web-service composition has been studied extensively over the last decade [3,4,6, 18,25,28]. A recent survey can be found in [20]. Let us consider the specification of the functional and non-functional requirements. Kumaran and Nandi [26] specify the behaviour of a web service by defining policies for data formats, and sequence and timing constraints. Specific languages such as the web services description language (WSDL), which is XML-based, have been developed to express these policies. Our service model also has time and sequence constraints, but unlike policies, we do not need to define the format of the data because this is low-level detail, and our approach takes a high-level view of service interaction. Instead, we use formal finite-state machines and Markov models and web-service languages such as WSDL would be inappropriate. Other work [4,27] also use finite-state machines to define web services. Although there is other work [14] that uses a Markov model for performance measurement as we do, in that work probabilities are used to model the rate of jobs leaving a server and joining a queue in a network, which is different from our probabilities that model the signal traffic between services.

In general, service composition can be carried out in two ways: referred to as *orchestration* and *choreography*. In orchestration, (component) web services are under the control of a central web service generally called a *coordinator*. This service coordinates the operations on the component services participating in the composition [31]. The component services do not 'know' that they are involved in a composition. The coordinator is the only service that has full knowledge of the composition, and orchestration is hence considered a top-down approach. Choreography, in contrast, is not centralised, and does not have a coordinator. Each web service that participates in the choreography has to know when to execute each of its actions and with whom it is communicating. In essence, choreography is a model of distributed knowledge where services work cooperatively to realise particular functionality. Full functionality is attained only by involving all services. This approach is hence considered to be bottom-up.

Orchestration is considered to be state-of-the-art because it is more flexible. Off-the-shelf component services can easily be incorporate into a composition, without the need for modification or reconfiguration: they do not need to know the identity of the service with which they interact. This is a facility that is required by most current service products [33], and it is indeed a facility that we take advantage of in our work.

A coordinator that is often encountered in daily life is a stock broker. It can be said that one of the functions of a broker is to *mediate* between the stock market and investors. Paolucci et al. [30] noted that 'brokering' involves both mediation and discovery, and that it is difficult to generate a broker automatically. If component services need to communicate, and they do not share a common language, then mediation involves providing a translation facility. Because a broker often connects a requester to any one of a number of service providers, the broker often also has the task of discovering the best (or most suitable) provider. So in principle a broker first must discover a suitable provider, and then mediate between the provider and the requester. We briefly survey research on these aspects below.

In [30,37], after discovering a suitable provider for a requester, a broker that mediates between them is generated. In that work, the broker cannot deal with information from one service that is not understood by the other service, which is one of the focuses of our work. At the operational level, these 'mismatches' are called *control mismatches*; at the data level, they are called *data mismatches*.

In [32], the broker translates user requests, expressed in an XML-formatted language, to pre-defined web-service messages. Unlike the language translation that our mediator carries out (which is happening at the operational level), mediators in other work [30,32,37] often just convert between formats. In [7], the mediator provides data flows and can resolve certain types of mismatches. Unfortunately, the method used to generate the mediator is not revealed in that work. In [7,15], a broker is connected to a so-called *engine*, which is a provider that connects with other services. These other services can be compared to our remote services.

The majority of research on orchestration focuses on discovery [1,35,37,42]. We do not separate the tasks of discovery and mediation: indeed, in our approach these tasks are tightly integrated. Given a requester and some number of providers, we build a mediator that composes the providers. Techniques generally used in discovery, such as verification and performance estimation, are part of our mediator design method. The non-functional requirements that are generally used to select the best provider in 'discovery' research are used in our research to identify the optimal mediator in the design space.

For example, Paolucci et al. [37] employ the technique of semantic matching of web services to find the most appropriate provider. Semantic matching focuses on ensuring that all functional requirements are satisfied, but the provider is not guaranteed to be correct. In discovery, verifying a form of correctness is studied in [35]. Unlike the temporally-based formal verification used in our approach, however, [35] verify the Quality-of-Service by running a set of test cases that check whether given QoS properties are satisfied by each provider. The QoS properties in [35] include response time (i.e. the time a provider takes to respond to a request), service cost and service availability. Their response time is equivalent to our response time. We do not consider service cost or availability because of the small number of services that are involved. QoS properties are also used in [2,21,29,39,42] for service discovery. Ai-Masri and Mahmoud [1] consider other performance measurements such as data throughput, which is equivalent to our data throughput. Yu and Lin [41] measure *system utility*, which is the total number of active requesters at each time, and *reconfiguration*, which involves the reallocation of resources among existing and incoming requesters. Very recently, Ivanovic et al. [23] formulate the computational cost of service networks by considering not only the internal logic of service composition, but also the number and behaviour of invoked services. They make assumptions about the services however, such as the number of the external services that will be involved in transactions. Our performance model goes further: transition probabilities in the Markov analysis account for the different individual response times by the external (remote) services and result in expected performance. However, we do assume in the energy performance model that the external services in each mediator in the design space consume the same energy. This is clearly an over-simplification, and removing this assumption is future work.

In research in mediator synthesis, formal methods have been used more extensively [19,40]. For example, [19] specify correctness properties using Linear Temporal Logic, and [40] compare various formal methods (namely automata, Petri nets and process algebras) for service composition. The work that is most closely related to our work is [28], who develop a mediator synthesis method in an orchestration framework (although in that work the mediator is called a choreographer, which conflicts with our nomenclature). Just as in our model, they represent services using I/O automata, but they go further and also define functional requirements using automata (instead of the temporal logic that we use). Given these automata, they developed local on-the-fly algorithms to explore all possible behaviours in so-called universal-service automata (which can be compared to our parallel composition automaton). If a correct coordinator is found amongst these behaviours, the exploration stops. This coordinator may or may not be the best coordinator (in terms of the user-defined non-functional requirements). The existence of a coordinator is 'proven' using a simulation relation. That approach is different to our formally-based on-the-fly algorithm, which, by construction, generates all correct behaviours, and then chooses the best one.

3 The Verifier phase

In this phase, the local and remote services are specified using finite-state machines, and the functional properties are specified using temporal logic formulae. From these specifications we generate in this phase a finite-state machine called the *verified parallel composition* (VPC) of the services.

Given finite-state machine presentations of services ς_1 and ς_2 that transfer data messages with size x and y (resp.), we first derive an Extended Parallel Composition (EPC). The derivation process is shown in Fig. 4.

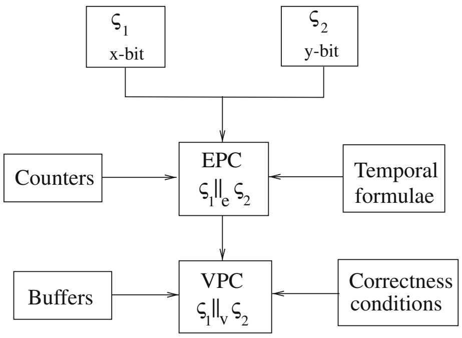


Fig. 4 The formal derivation of the VPC

The EPC is the cross product of the input services that adds counters and temporal formulae to each state. A counter sums the number of data *units* (e.g. characters) in each buffer in a state, and the temporal formulae describe functional properties. The EPC is further transformed by applying these properties and placing bounds on the counters. This results in the verified parallel composition (VPC), which is the output of this phase. The derivation of the VPC begins with the formal derivation of the input services.

Definition 1 *Service* A service is represented as an extended finite-state machine $\varsigma = \langle S, L, R, \delta, s_0 \rangle$, where:

- S is a finite set of states.
- L and R are finite sets of local and remote actions respectively. These actions are either input/output control actions or data actions, denoted as $L = L^{I_c} \cup L^{O_c} \cup L^{I_d} \cup L^{O_d}$ and $R = R^{I_c} \cup R^{O_c} \cup R^{I_d} \cup R^{O_d}$ respectively. Control actions L^{I_c}, L^{O_c}, R^{I_c} and R^{O_c} are Booleans while data actions L^{I_d}, L^{O_d}, R^{I_d} and R^{O_d} correspond to (real) data.
- $\delta \subseteq S \times 2^L \times 2^R \times S$ is a set of transitions. Each transition $t_{i,j} \in \delta$ is written $t_{i,j} = (s_i, L_{i,j}, R_{i,j}, s_j)$, where $s_i, s_j \in S$, $L_{i,j} \subseteq L$ and $R_{i,j} \subseteq R$ labels $t_{i,j}$ with local and remote actions.
- $s_0 \in S$ is the initial and the final state.

Conventionally, when reactive systems are modelled the initial and final states are defined to be the same. This is also an assumption that is required by the Markov analysis, and is not a restriction in practice.

In Fig. 5 we show a fragment of a finite-state machine that represents an *Airline* service. We see in this figure a state s_1 that reaches s_2 by receiving request denoted as 'req^a?'. In the figure, actions with suffix '?' are input actions, and actions with suffix '!' are output actions. The superscript 'a' denotes the *Airline* service. State s_2 has three outgoing transitions leading to state s_2, s_3 and s_4 resp. These transitions are labelled by different output actions, e.g. 'ava^a!' means the service outputs the flight is available, 'ava_0^a' means it outputs the flight is unavailable, and '$search^a$!' means it is searching for a flight. Note that this self-loop represents the possibility that the service will need to wait for a response. During this waiting period, one or more external services may be invoked. Including waiting time does not increase the size of model. However, it does affect the transition probabilities, and this will impact the performance of the mediators. We discuss this in more detail in 5. Notice that there

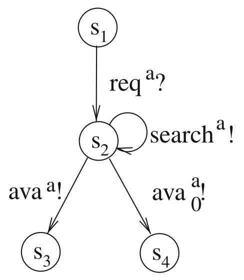


Fig. 5 A fragment of the *Airline* service

are no input actions specified in this fragment. This is important as it means that no external entity can control this service if it is in state s_2: the service may in fact take any one of these three transitions ‘spontaneously’. It may do so because there is no input action to control its behaviour. A state that exhibits this (multitude of) behaviours is said to be *uncontrollable*.

The notion of uncontrollability requires the input control actions labelling two transitions from a state to be the same (or non-existent), and either input data actions, or the local output actions, or the (input or output) remote actions to be different. Note that transitions from a state must be unique, so if input actions are identical, then the output actions must differ. Non-determinism occurs when two transitions have both inputs and outputs identical. Non-determinism is rare in practice because one would not expect or specify different behaviour for the same input and output actions. We in fact do not allow non-determinism in this work. The concept of uncontrollability, in contrast, allows services to ‘take the initiative’ in the sense that, given particular input, the service may decide for itself what the appropriate output should be, and hence show different behaviour.

We define ‘uncontrollability’ formally as follows.

Definition 2 *Uncontrollable service* A service $\varsigma = \langle S, L, R, \delta, s_0 \rangle$ is *uncontrollable* iff there exists at least one state $s_i \in S$ and its child states $s_j, s_m \in S$, where $s_j \neq s_m$ such that $\exists t_{i,j} = (s_i, L_{i,j}, R_{i,j}, s_j) \in \delta$ and $\exists t_{i,m} = (s_i, L_{i,m}, R_{i,m}, s_m) \in \delta$, and

$$(L^{I_c}_{i,j} = L^{I_c}_{i,m}) \wedge ((L_{i,j} \cup R_{i,j}) \neq (L_{i,m} \cup R_{i,m}))$$

States s_j and s_m are said to be *twins*.

Twin states represent uncontrollable behaviours in which the mediator is not able to select a single behaviour in a given state. If one of the behaviours violates any of the correctness conditions however, then the mediator may be able to avoid reaching the state where a choice needs to be made. Returning to the example in Fig. 5, the three outgoing transitions of s_2 are labelled by the same input, namely the empty input. The children of s_2 are s_3, s_4 and itself. Each pair of these states is a twin (so the three states form a triplet). Twin states might cause a problem because the transition from s_2 to one of the twins may lead to correct behaviour, but another transition from s_2 to another twin may be invalid. In practice, this means that one service may send control or data messages that the other service cannot handle. Recognising the existence of uncontrollable states is important in defining correct behaviours, as we shall see later.

In contrast to an uncontrollable state such as s_2, a controllable state has outgoing transitions that are labelled by distinct input control actions. From a controllable state, a mediator can decide and force the services to execute a particular behaviour by sending an input control action that labels the transition that goes to a particular child state.

A specification of a system comprises a specification of all the local and remote services. This specification, together with the temporal properties, forms the input of the Verifier. As we can see in Fig. 4, to synthesise a mediator, we need to define the extended parallel composition (EPC) of all the local and remote services. In the following definition, we assume that the mediator requires just one buffer, but it can be easily extended to involve more buffers.

Definition 3 *Extended parallel composition (EPC) of services* Given services $\varsigma_1 = \langle S^a, L^a, R^a, \delta^a, s^a_0\rangle$ and $\varsigma_2 = \langle S^b, L^b, R^b, \delta^b, s^b_0\rangle$, and a set of formulae Ψ, an EPC is defined by $\varsigma_1 \ ||_e \ \varsigma_2 = \langle S^e, \mathcal{L}, L^e, R^e, \delta^e, s^e_0\rangle$, where:

- $S^e \subseteq S^a \times S^b \times K$, and $K \subset \mathbb{N}$ is the set of *buffer* sizes (which counts the number of data elements in the buffer at a state). A state in S^e is denoted as $s_{i,g,k}$, where $s_i \in S^a$, $s_g \in S^b$ and $k \in K$.
- $\mathcal{L}$ is a state labelling function: $S^e \rightarrow 2^{\Psi}$. For each state $s_{i,g,k} \in S^e$, if $s_{i,g,k} \models \mathcal{L}(s_{i,g,k})$, then $s_{i,g,k}$ is a *valid state*.
- $L^e = L^a \cup L^b$ and $R^e = R^a \cup R^b$.
- $\delta^e \subseteq S^e \times (2^{L^e} \times 2^{R^e}) \times S^e$. For each transition $t_{i,j} = (s_i, L_{i,j}, R_{i,j}, s_j) \in \delta^a$ in ς_1 and $t_{g,h} = (s_g, L_{g,h}, R_{g,h}, s_h) \in \delta^b$ in ς_2, there exists a set of transitions in δ^e of the form:
$$t_{i,j}t_{g,h} = (s_{i,g,k}, L_{i,j} \cup L_{g,h}, R_{i,j} \cup R_{g,h}, s_{j,h,l})$$
where $s_{i,g,k}, s_{j,h,l} \in S^e$
- $s^e_0 = (i^a, i^b, 0) \in S^e$

Notice that we define buffers to store data, and we define the set of properties that the EPC is expected to satisfy. A counter is used to record the number of data units that are stored in a buffer at each state. This counter is incremented when data is received by the mediator, and decremented when data is sent. The properties Ψ are expressed as CTL formulae.

An EPC is *uncontrollable* iff $\exists s_{i,g,k} \in S^e$ such that s_i is an uncontrollable state in ς_1, or s_g is uncontrollable in ς_2. Two states $s_{i,g,k}$ and $s_{j,h,l}$ are *twins* iff s_i and s_j are twins in ς_1, or s_g and s_k are twins in ς_2. If $s_{i,g,k} \models \mathcal{L}(s_{i,g,k})$ then the state is said to be a *valid twin*; alternatively, if $s_{i,g,k} \not\models \mathcal{L}(s_{i,g,k})$, then the state is said to be an *invalid twin*. Analogously for $s_{j,h,l}$. Note that a safety issue may occur if we remove invalid twins. Because the behaviours are uncontrollable, the invalid twin states may be unavoidable in the sense that the service ς_1 (or ς_2) may still send the signal that leads to one of the removed states. Particular valid twin states hence need to be removed from the mediator, so that the mediator can no longer be placed in this situation. The extra twin states that need to be removed are called *twin-unsafe*. The concept of twin-safe states can be formally defined (see [10]). In this work, we will explain the twin-safe issue by example later.

Given the EPC, the VPC can be defined as follows.

Definition 4 *Verified parallel composition (VPC)* Given an EPC defined by $\langle S^e, \mathcal{L}, L^e, R^e, \delta^e, s^e_0\rangle$ with the set of formulae Ψ and buffer size k^{m}, a VPC is defined by $\varsigma_1 \ ||_v \ \varsigma_2 = \langle S^r, \mathcal{L}^r, L^r, R^r, \delta^r, s^r_0\rangle$ where:

- $S^r \subseteq S^e$ is a finite set of states, and for each state $s_{i,g,k} \in S^r$ we have $0 \leq k \leq k^{\mathrm{m}}$, where k is the buffer size.
- $\mathcal{L}^r : S^r \rightarrow 2^{\Psi}$, $\forall s_{i,g,k} \in S^r$ we have $s_{i,g,k} \models \mathcal{L}^r(s_{i,g,k})$.
- $L^r \subseteq L^e$ and $R^r \subseteq R^e$, where $L^{O_r} \subseteq L^{I_e}$ is a set of output local actions, $L^{I_r} \subseteq L^{O_e}$ is a set of input local actions, and $R^{I_r} \subseteq R^{I_e}$ is a set of input remote actions, $R^{O_r} \subseteq R^{O_e}$ is a set of output remote actions,

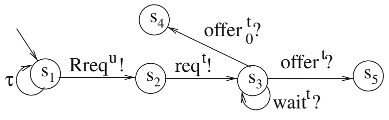


Fig. 6 Fragments of the *Travel-Agency* service

- $\delta^r \subseteq S^r \times (2^{L^r} \times 2^{R^r}) \times S^r$
- $s^r_0 = s^e_0$

For all $s_{i,g,k} \in S^e$, if $s_{i,g,k} \models \mathcal{L}^e(s_{i,g,k})$ and $s_{i,g,k}$ has no invalid-twin, then:

- $s_{i,g,k} \in S^r$
- $(s_{i,g,k}, \mathcal{L}^e(s_{i,g,k})) \in \mathcal{L}^r$
- for all transitions $(s_{i,g,k}, L^I_{i,j} \cup L^I_{g,h}, L^O_{i,j} \cup L^O_{g,h}, R_{i,j} \cup R_{g,h}, s_{j,h,l}) \in \delta^e$ then $(s_{i,g,k}, L^O_{i,j} \cup L^O_{g,h}, L^I_{i,j} \cup L^I_{g,h}, R_{i,j} \cup R_{g,h}, s_{j,h,l}) \in \delta^r$

Note that the input and output actions in each transition in a VPC are the same as those in the EPC, but the local actions are reversed. The remote actions in contrast are not reversed because they are not involved in transactions with the VPC. The basic algorithm that generates the VPC according to the above definition can be found in [10], but remote actions need to be handled as well.

Example 2 In the original example shown in Fig. 2, let us consider fragments of the *Travel-Agency* service shown in Fig. 6 and *Airline* service shown in Fig. 5 (the complete specifications of these services will be presented in Sect. 6). The *Travel-Agency* service receives requests for flights and hotels, denoted by '$Rreq^u$', from a user, and then waits until a result from the *Airline* service is received. The result can be either a possible flight offer, denoted by '$offer^t$', or an indication that the flight is not available, denoted by '$offer^t_0$'. In these two fragments, $Rreq^u$? is a remote action and all other actions are local.

The corresponding fragment of EPC of these two services can be seen in Fig. 7. There is one data channel involved in the fragments, with its own buffer. A request for a flight from the *Travel Agency* is relayed to the *Airline* (i.e. 'req^t' is translated into 'req^a'). We record the number of data in this buffer in each state, e.g. $s_{2,1,0}$ means that no data is stored when *Travel Agency* visits state s_2 and *Airline* visits s_1.

The functional properties that have been applied in this example are:

ϕ_1. AG($s_{1,1,0} \rightarrow$ AXAF$s_{1,1,0}$): from the initial state, the final state can always eventually be reached, which means every state is reachable and no deadlock occurs.

ϕ_2. AG($s_{1,1,0} \rightarrow$ AXA($\neg offer^t \; U \; ava^a$)): from the initial state, the mediator cannot send '$offer^t$' to the Agency until 'ava^a' has been sent by the *Airline* service.

ϕ_3. AG($s_{1,1,0} \rightarrow$ AXA($\neg offer^t_0 \; U \; ava^a_0$)): from the initial state, the mediator cannot send '$offer^t_0$' to the Agency until 'ava^a_0' has been sent by the *Airline* service.

ϕ_4. We let the maximum buffer size be zero. Formally, for each state $s_{i,j,k} \in S^r$ we have $k = 0$, where we assume the input/output data message sizes are the same.

These properties are input to the model checker, which, when applied to the EPC shown in Fig. 7, will detect that (i) states $s_{5,2,0}$ and $s_{5,4,0}$ violate ϕ_2, (ii) states $s_{4,2,0}$ and $s_{4,3,0}$ violate ϕ_3 and (iii) states $s_{1,2,1}$, $s_{2,2,1}$, $s_{3,1,-1}$ violate ϕ_4, and hence are invalid.

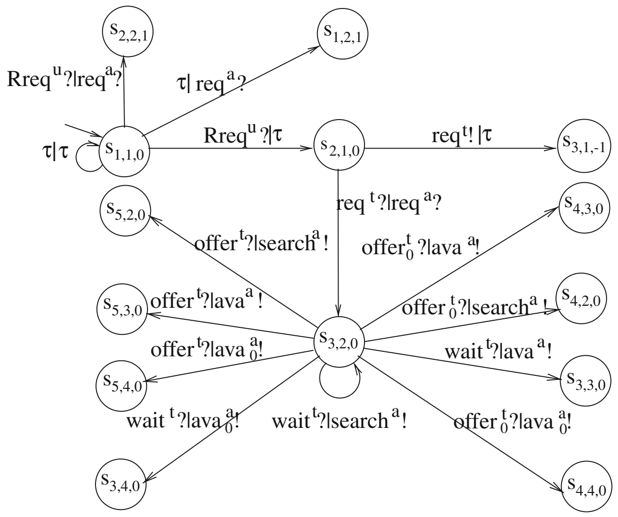


Fig. 7 A fragment of the EPC of the services shown in Figs. 6 and 5

These states are hence removed, and as this occurs 'on the fly', no children will be generated from any of these states.

Before removing invalid states, we need to consider the twin-unsafe issue. Let us consider invalid state $s_{5,4,0}$ and its twin state $s_{5,3,0}$ for example. A potential problem is that when the mediator makes the transition to $s_{5,3,0}$, it will expect to receive ava^a from the *Airline*, and send $offer^t$ to the *Travel-Agency*. However, the mediator cannot control the *Airline*, so whether it receives ava^a or ava_0^a from the *Airline* is uncontrollable. If ava_0^a is received, and $offer^t$ is sent, then the invalid state $s_{5,4,0}$ will be reached. Reaching this invalid state can be avoided with the existence of valid states $s_{4,4,0}$ and $s_{3,4,0}$ if the mediator sends $offer_0^t$ (or $wait^t$) after receiving ava_0^a. In that case $s_{4,4,0}$ (or $s_{3,4,0}$) will be reached instead. Therefore, states $s_{5,4,0}$ and $s_{5,3,0}$ are twin-safe in this case, and the invalid state $s_{5,4,0}$ can be removed safely. If $s_{4,4,0}$ and $s_{3,4,0}$ are invalid in this example, then $s_{5,4,0}$ and $s_{5,3,0}$ will become twin-unsafe.

The resulting fragment of the VPC is shown in Fig. 8. Note that in this process the directions of local actions on each transition are reversed (receive actions become send actions, and vice versa).

At the behaviour level, there is no need for the VPC to know the content of each data message, but we do handle data and control behaviours. For example in Fig. 7, when the *Airline* is searching, the *Travel-Agency* is waiting, as revealed by the self-loop at state $s_{3,2,0}$. In this state, if the *Airline* responds with 'not available', i.e. the Boolean control message ava $= 0$, then the *Travel-Agency* will receive 'no offer', i.e. the Boolean control message offer $= 0$ as a notification from the mediator. This can be seen on the transition label from $s_{3,2,0}$ to $s_{4,4,0}$. So $s_{4,4,0}$ will be selected as the next state. Similarly for the other four cases that might happen at state $s_{3,2,0}$.

The existence of a VPC in this example means that a functionally correct mediator can be synthesised. In fact, it may have been possible to generate a mediator using a

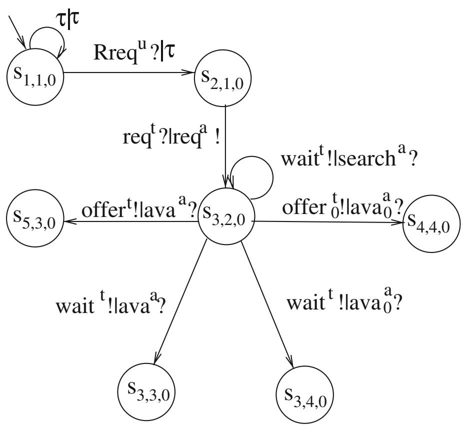


Fig. 8 The VPC that results from the EPC shown in Fig. 7

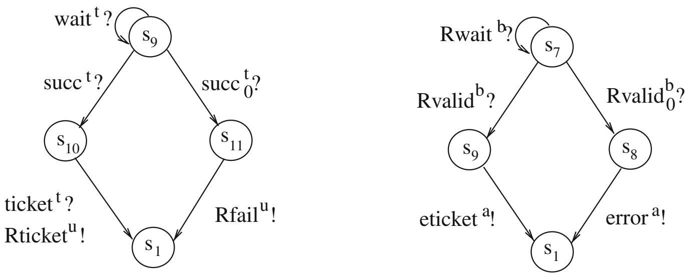


Fig. 9 Fragments of the *Travel-Agency* service (*left*) and *Airline* service (*right*)

simpler model that does not include remote services. However, in general, the incorporation of remote services in the model means that a mediator may exist in cases where it will not if the remote services are excluded. The next example illustrates this.

Example 3 Consider other fragments of the *Travel-Agency* and *Airline* services shown in Fig. 9. The EPC that is generated from these fragments is shown in Fig. 10. The corresponding VPC is shown in Fig. 11. For the sake of argument, let us assume that there is no buffer, or more precisely, the maximum buffer size is zero. In that case, states $s_{10,8,0}$ and $s_{11,9,0}$ in the EPC are invalid because their respective children $s_{1,1,-1}$ and $s_{1,1,1}$ are invalid because of buffer underflow and overflow respectively. The VPC can avoid reaching these invalid states in the following way. If the remote action '$Rvalid^b$' has been received by the *Airline*, then the mediator sends '$succ^t$' to the *Travel Agency* and reaches state $s_{10,9,0}$. Otherwise, if '$Rvalid_0^b$' has been received by the *Airline*, then the mediator sends '$succ_0^t$' to the *Travel Agency* and reaches state $s_{11,8,0}$. The result is the invalid states cannot be reached. We can therefore remove the invalid states from the VPC, and generate a valid mediator. Note that for simplicity, in each state only one buffer that is relevant to this example is shown, and the self loops in both services are ignored because they do not play any role in this case.

Let us now remove the remote services from the model. The remote actions in Fig. 9 are the Bank's actions '$Rvalid_0^b$', '$Rvalid^b$' and '$Rwait^b$', and the User's

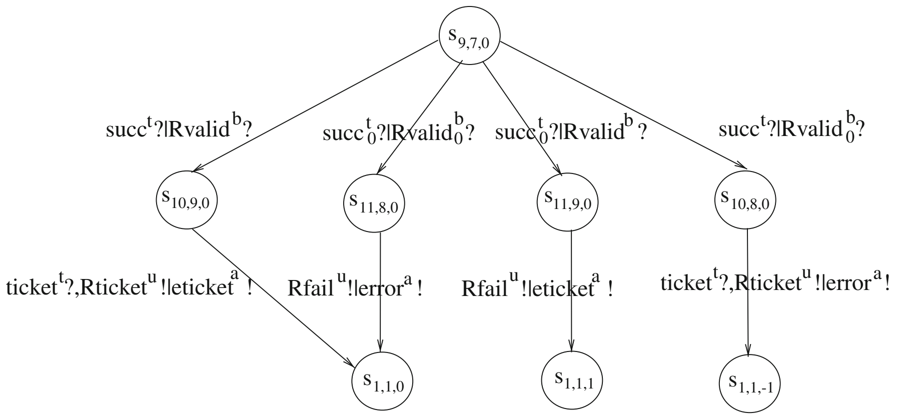
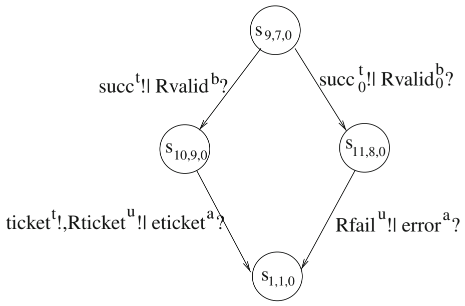


Fig. 10 The EPC that results from the service fragments shown in Fig. 9

Fig. 11 The VPC that results from the EPC shown in Fig. 10



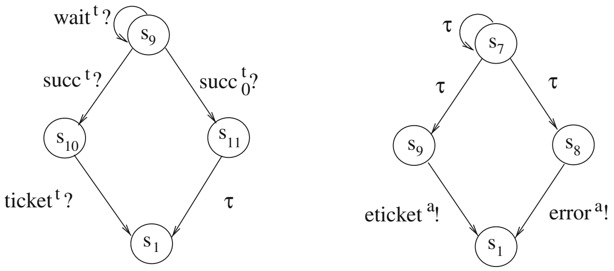


Fig. 12 Fragments of the *Travel-Agency* service (*left*) and *Airline* service (*right*) without remote transactions

actions '$Rfail^u$' and '$Rticket^u$'. We remove these actions by changing them into τ-transitions. The resulting simplified fragments are shown in Fig. 12. The EPC that is generated from these service fragments is shown in Fig. 13. No VPC can be synthesised from this EPC because all the children from $s_{9,7,0}$ are twin-unsafe. Since the interaction with the *Airline* is τ, i.e. empty, it is impossible to control: the VPC can send either '$succ_0^t$' or '$succ^t$', but both cases will lead to one of the invalid states $s_{11,9,0}$ or $s_{10,8,0}$. If we cannot generate a VPC, there is no mediator.

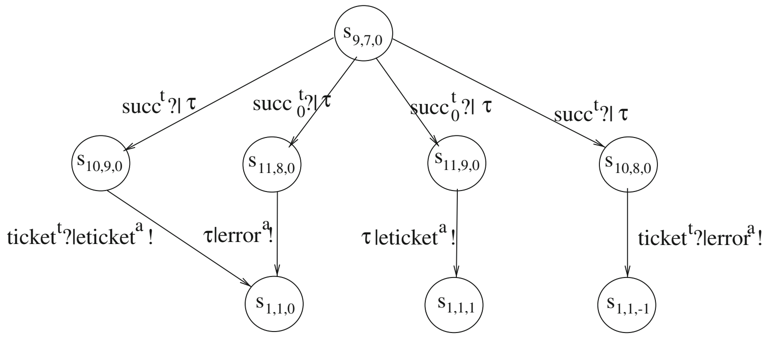


Fig. 13 The EPC that results from the service fragments that have no remote transactions, shown in Fig. 12

Only if the algorithm can guarantee that there will be no invalid behaviour will the algorithm generate a mediator. The above example shows us that the knowledge of remote services can enable a mediator to be synthesised in cases where it would otherwise not be possible.

3.1 Generating the VPC

A flowchart of the VPC generator algorithm is shown in Fig. 14. The process of VPC construction starts by generating the parallel composition. The VPC Generator Algorithm then employs depth-first search to traverse each state in the parallel composition. Once a new state is found, it is updated by adding buffer values and labelling formulae. The updated state is then checked if it satisfies all its labelled formulae. If it violates a formula then it will be removed immediately, hence the successive states will never be generated. Otherwise, the construction continues. In this sense the construction is *on-the-fly*. As a result, we do not need to generate the complete state space of the EPC.

To check if an updated state s satisfies all its labelled formulae $\mathcal{L}(s)$, the Tableau-Construction Algorithm shown in Algorithm 1 is used. There are three possible return values from the Tableau-Construction Algorithm:

1. if it returns true, then the state s is added to the VPC;
2. if it returns false, then s is invalid, and the Safety-Check Algorithm shown in Algorithm 2 is used to check if all twins of s are twin-safe;
3. if it returns undecided, then we update the parent states. This means that the validation of the current state is still unknown, which may depend on whether some future state will be reached.

In the first case, if the return value is true then we do not need to call the Safety-Check Algorithm. This is because, in the definition of a twin-safe state, the Safety-Check Algorithm is required only when an invalid state has been found. In the second case, an invalid state is found. If it is twin-unsafe according to the Safety-Check Algorithm, then the state will be removed from the VPC, and its parent states are updated. In the third case, the value is undecided. However, all the undecided states will return either true or false by the end of the Algorithm.

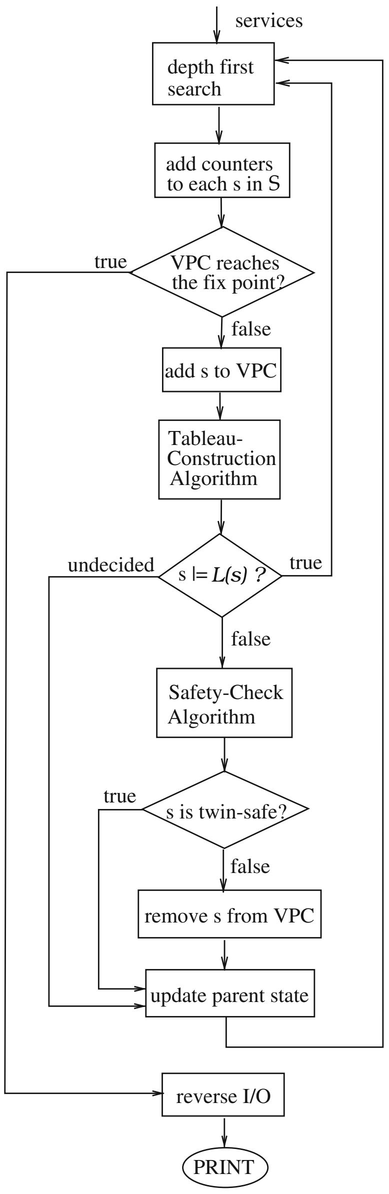


Fig. 14 The VPC generator algorithm

Using a fixed-point calculation, states are verified and new states are added to the VPC. If no more states can be found, then we say that the fixed point is reached by the algorithm. When the fixed point is reached, we reverse the inputs and outputs of all transitions between any two states in the VPC, and add the reversed transitions to the VPC. Conceptually, given a parallel composition, we generate a VPC by checking the correctness conditions at all the states and corresponding transitions.

Shown in Algorithm 1, the Tableau-Construction Algorithm checks temporal formulae using the tableau-method based model checker.

Algorithm 1 Tableau-Construction Algorithm

```
1:  L(s) = $\Psi$
2:  while $\Psi$ is non-empty do
3:      remove formula $\psi$ in $\Psi$
4:      if $\psi$ = true then
5:          continue
6:      else if $\psi$ = false then
7:          return $s \not\models \mathcal{L}(s)$
8:      else if $\psi = AG\psi_1$ then
9:          add $\psi_1$ and AXAG $\psi_1$ in $\Psi$
10:     else if $\psi = AX\psi_1$ then
11:         add $\psi_1$ to $\Psi_{AX}$, set $s$ as a $AX\_state$
12:     else if $\psi = A(\psi_1 U \psi_2)$ then
13:         add $\psi_3 = \psi_2 \vee (\psi_1 \wedge AX\psi)$ in $\Psi$
14:     else if $\psi = \psi_1 \vee \psi_2$ then
15:         create states $s_1 = s$, $\mathcal{L}(s_1) = \psi_1 \cup \Psi_{AX}$;
16:         create states $s_2 = s$, $\mathcal{L}(s_2) = \psi_2 \cup \Psi_{AX}$;
17:         add $(s_1, \mathcal{L}(s_1))$ and $(s_2, \mathcal{L}(s_2))$ to $\mathcal{L}$, and set $s$ to an $OR\_state$
18:         return undecided
19:     else if $\psi = \psi_1 \wedge \psi_2$ then
20:         add $\psi_1$ and $\psi_2$ in $\Psi$
21:     end if
22: end while
23: if $\Psi_{AX}$ is non-empty then
24:     $\Psi'_{AX} = \{\psi | AX\psi \in \Psi_{AX}\}$, $s$ is $AX\_state$
25:     for all successor state $s'$ of $s$ do
26:         create states $(s', \mathcal{L}(s') = \Psi'_{AX})$
27:     end for
28:     return undecided
29: end if
30: return $s \models \mathcal{L}(s)$
```

Following [36], this algorithm provides a set of rules to decompose formulae into sub-formulae. To illustrate, we apply the Tableau-Construction Algorithm to a fragment of an EPC shown in Fig. 15 on the left. At initial state i^R, we are given a property $AG(s_0^e \rightarrow AXA(b_0 \, U \, e_1))$ from Ψ, which is the label $\mathcal{L}(i^R)$ of the state. We first apply the *AG* rule (at Line 8) and express implication as disjunction to get

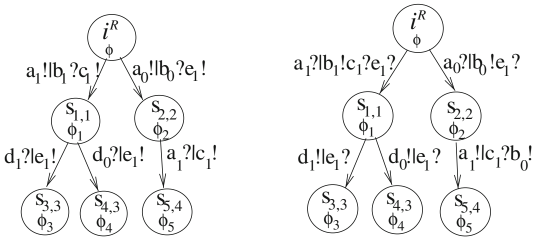


Fig. 15 A fragment of an EPC (*left*) and a fragment of the VPC (*right*)

$\phi = \neg s_0^e \vee AXA(b_0 \, U \, e_1)$. Since the first term of ϕ is false, we apply the AX rule (at Line 10) to the second term, and generate states $s_{1,1}$ and $s_{2,2}$ with $\phi_1 = \phi_2 = A\,(b_0 \, U \, e_1)$ and Ψ_{AX}. Similarly we expand $s_{1,1}$ by applying the AU rule (at Line 12) to generate $e_1 \vee (b_0 \wedge AXA(b_0 \, U \, e_1))$. Note that at $s_{1,1}$, the property ϕ_1 is satisfied only if $e = 1$, so $s_{1,1}$ is conditionally valid. The VPC will generate $e_1!$ with the result that $s_{1,1}$ is forced valid. Similarly, $s_{5,4}$ is also forced valid. Note as well that states $s_{2,2}$, $s_{3,3}$ and $s_{4,3}$ are valid. A fragment of the resulting VPC is shown in Fig. 15 on the right.

Algorithm 2 Safety-Check Algorithm

```
Input: 1) An invalid or twin-unsafe state $s_{i,g}$, where $s_i$ is in service $\varsigma_1$, and $s_g$ in service
    $\varsigma_2$; 2) a set $C$ of siblings from the same parent state $s_{x,y}$, and the transition from $s_{x,y}$
    to each child $s_{u,v} \in C$ that is labelled by $A(t_{x,u})$.
Output: Each state in $C$ is safe or not.
 1: for all $s_{u,v} \in C$ do
 2:     if $s_i$ and $s_u$ are twins and $s_g$ and $s_v$ are not twins then          ▷ Case 1
 3:         $s_{u,v}$ is twin-unsafe
 4:         for all $s_{l,m} \in C$ do
 5:             if $s_l = s_i$ and $s_m$ is not a twin of $s_v$ then
 6:                 $s_{u,v}$ is twin-safe
 7:                 if $s_g \neq s_v$ then
 8:                     if $A(t_{x,u})$ occurs in service $\varsigma_1$, then the mediator sends $A(t_{y,v})$ to $\varsigma_2$
 9:                     if $A(t_{x,l})$ occurs in service $\varsigma_1$, then the mediator sends $A(t_{y,m})$ to $\varsigma_2$
10:                 end if
11:                 break
12:             end if
13:         end for
14:     else if $s_i$ and $s_u$ are not twins and $s_g$ and $s_v$ are twins then     ▷ Case 2
15:         $s_{u,v}$ is twin-unsafe
16:         for all $s_{l,m} \in C$ do
17:             if $s_m = s_g$ and $s_l$ is not twin of $s_u$ then
18:                 $s_{u,v}$ is twin-safe
19:                 if $s_i \neq s_u$ then
20:                     if $A(t_{y,v})$ occurs in service $\varsigma_2$, then the mediator sends $A(t_{x,u})$ to $\varsigma_1$
21:                     if $A(t_{y,m})$ occurs in service $\varsigma_2$, then the mediator sends $A(t_{x,l})$ to $\varsigma_1$
22:                 end if
23:             end if
24:         end for
25:     else if $s_i$ and $s_u$ are twins and $s_g$ and $s_v$ are twins then          ▷ Case 3
26:         $s_{u,v}$ is twin-unsafe
27:     else                                                                         ▷ Case 4
28:         $s_{u,v}$ is twin-safe
29:     end if
30:     if $s_{u,v}$ is twin-unsafe then
31:         remove $s_{u,v}$
32:         call safety_check
33:     end if
34: end for
```

Shown in Algorithm 2, the Safety-Check Algorithm checks the twin-safe property defined in [10] by examining states in pairs. There are four cases to be considered, and each one can be identified by the corresponding labels on the right in the algorithm. As an example, let us revisit the EPC shown in Fig. 7. When we apply the VPC Generator Algorithm, the Tableau-Construction Algorithm will detect the invalid state $s_{5,4,0}$. Following this, the Safety-Check Algorithm will be applied. The input to this algorithm, state $s_{i,g}$, will be $s_{5,4,0}$, whose parent state $s_{x,y}$ is $s_{3,2,0}$. The rest of the children of the parent are contained in the set $C = \{s_{3,3,0}, s_{3,4,0}, s_{4,2,0}, s_{4,3,0}, s_{4,4,0}, s_{5,2,0}, s_{5,3,0}, s_{3,2,0}\}$, which are called siblings of $s_{5,4,0}$. There are no twins among s_3, s_4 and s_5 in *Travel-Agency* service, while states s_2, s_3 and s_4 are triples in *Airline* service. This belongs to Case 2 at Line 14 of the Safety-Check Algorithm. To illustrate, we take the same example we discussed earlier for instance: $s_{u,v} = s_{5,3,0}$. There is a state $s_{l,m} = s_{4,4,0}$ that satisfies the condition at Line 17 of Case 2. Therefore $s_{5,3,0}$ is twin-safe, which is the same result as we concluded earlier.

4 The Builder phase

In Fig. 4 we saw how the VPC was generated in the Verifier phase. In the Builder phase, we split the VPC into all its possible component behaviours. There may in fact be many behaviours, each of which corresponds to a different mediator that satisfies the given functional properties.

Example 4 Consider the fragment of a VPC $\varsigma_1 \;||_v\; \varsigma_2$ shown in Fig. 16. This VPC contains a state s_1 with two outgoing transitions. At state s_1, the VPC receives action $c_1?$ from service ς_2, and by sending $a_1!$ to service ς_1, the VPC will be in state s_2, or by sending $b_1!$, will be in state s_3. As the inputs are the same, the VPC can decide for itself which transition to take, and hence what the next state will be. Therefore in state s_1 two different behaviours are possible. We can 're-draw' this VPC to make the choice between the two component behaviours explicit. This is shown in Fig. 17. Each of the component behaviours in this figure is a new VPC.

Detecting the component behaviours of a VPC requires identifying states that are *splittable*. The state s_1 in the example above is such a state. A splittable state in a VPC has at least two outgoing transitions labelled by different output actions. These outgoing transitions are said to be *send-nonequivalent*. By sending different outputs, the component VPCs of course exhibit different behaviours. (Note that if the outgoing transitions are labelled by identical output actions then, as we exclude non-determinism in this work, these transitions must have different inputs.)

If a state is unsplittable, then there is no way for the mediator to influence which transition the state will take. Alternatively, if a state is splittable, then it can be

Fig. 16 A small VPC $\varsigma_1 \;||_v\; \varsigma_2$

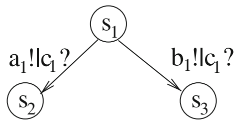


Fig. 17 The component behaviours of the VPC in Fig. 16

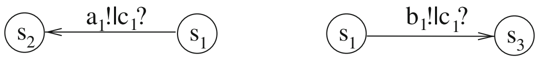
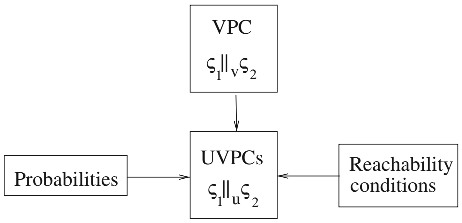


Fig. 18 A formal derivation of UVPC



controlled by the mediator. A mediator is free to choose any transition at a splittable state. If we consider all the possible choices that can be made at every splittable state, then we can generate a set of VPCs, each of which consists solely of unsplittable states. Such a VPC is called an Unsplittable VPC (UVPC).

Splitting the states in the VPC to produce a set of UVPCs is the main task of the Builder phase. We depict this in Fig. 18, where the probabilities and reachability conditions are also required to generate the UVPCs. The probabilities are used by the Estimator phase to estimate the performance of the service composition for each individual UVPC.

To formally model performance, we represent a UVPC as a Markov chain. This requires a formal definition of the splittable and unsplittable states in the VPC.

Definition 5 *Splittable and unsplittable states in a VPC* Given a state $s \in S^r$ of a VPC $\langle S^r, \mathcal{L}^r, L^r, R^r, \delta^r, s_0^r\rangle$, if there are two outgoing transitions t_1t_1' with label $(L^O(t_1) \cup L^O(t_1'), L^I(t_1) \cup L^I(t_1'), R(t_1) \cup R(t_1'))$, and t_2t_2' with label $(L^O(t_2) \cup L^O(t_2'), L^I(t_2) \cup L^I(t_2'), R(t_2) \cup R(t_2'))$ from state s, such that:

$$(L^{O_c}(t_1) = L^{O_c}(t_2)) \wedge ((L^I(t_1) \neq L^I(t_2)) \vee (R(t_1) \neq R(t_2))))$$

or

$$(L^{O_c}(t_1') = L^{O_c}(t_2')) \wedge ((L^I(t_1') \neq L^I(t_2')) \vee (R(t_1') \neq R(t_2'))))$$

where $L^{O_c}(t_1)$ is a set of output control actions that label t_1, and similarly for $L^{O_c}(t_1')$, $L^{O_c}(t_2)$ and $L^{O_c}(t_2')$; then transitions t_1t_1' and t_2t_2' are said to be *send-equivalent*. Otherwise t_1t_1' and t_2t_2' are said to be *send-nonequivalent*, and in that case, the state s is said to be *splittable*. If any two outgoing transitions from s are send-nonequivalent, then state s is said to be *unsplittable*.

Given a VPC, an algorithm to identify the splittable states in the VPC is shown in Algorithm 3. The output of the algorithm is a set of splittable states and a set of send-nonequivalent transitions of each transition in the VPC.

Algorithm 3 Splittable States Identifier

Input: VPC: A verified parallel composition of mediators
Output: S^s: A set of splittable states in VPC, $S^T(t)$: A set of send-nonequivalent transitions of each transition t

```
1:  $S^s = \emptyset$
2:  for each state $s$ in VPC do
3:      for each outgoing transition $t$ of $s$ do
4:          $S^T(t) = \emptyset$
5:          for each outgoing transition $t' \neq t$ of $s$ do
6:              if $t$ and $t'$ are send-nonequivalent then
7:                  add $t'$ to $S^T(t)$
8:                  add $s$ to $S^s$
9:              end if
10:         end for
11:     end for
12: end for
```

Using Definition 5, we can define an unsplittable VPC as follows.

Definition 6 *Unsplittable VPC (UVPC)* Given a set of probabilities $\mathcal{P}$ and a VPC $\langle S^r, \mathcal{L}^r, L^r, R^r, \delta^r, s_0^r \rangle$, where each state is $s_{i,g,k} \in S^r$ and each transition is $(s_{i,g,k}, L_{i,j} \cup L_{g,h}, R_{i,j} \cup R_{g,h}, s_{j,h,l}) \in \delta^r$, a UVPC can be represented by a Markov chain $\langle B, U, L^u, R^u, \mathcal{P}, \varrho, s_{0,0} \rangle$, where:

- $B \subseteq [0..k^{\mathrm{m}}]$ is a finite set of integers that represents buffer sizes, and $B = \{k | s_{i,g,k} \in S^r\}$.
- $U \subseteq S^r$ is a set of unsplittable states. A state $s_{i,g,k} \in S^R$ that has k data units in the buffer is represented by $s_{x,k} \in U$ in a UVPC, where $x = (i, g)$.
- $L^u \subseteq L^r$ and $R^u \subseteq R^r$
- $\varrho \subseteq U \times 2^{L^u} \times 2^{R^u} \times [0, 1] \rightarrow U$ is a function that labels transitions between states. Transitions are labelled by actions and probabilities $\mathcal{P}$. The probabilities of all the outgoing transitions of a state sum to one. Each transition in ϱ is written $(s_{x,k}, A_{x,y}, \mathcal{P}_{(x,k),(y,l)}, s_{y,l})$, where $s_{y,l} = x_{j,h,l} \in S^r$, $s_{x,k}, s_{y,l} \in U$, $\mathcal{P}_{(x,k),(y,l)} \in \mathcal{P}$, and $A_{x,y} = (L_{i,j} \cup L_{g,h}, R_{i,j} \cup R_{g,h})$.
- $s_{0,0} = s_0^r$ is the initial and final state, which can reach any state in U and can be reached by any state in U.

The one-step probabilities $\mathcal{P}$ generally come from historical simulation data (for example [14]). Using this definition, a Design-Space Generation algorithm can be written. This is shown in Algorithm 4.

Given the VPC, the one-step probabilities of each behaviour, and $S^T(t)$ that is generated by Algorithm 3, the Design-Space generator commences in Line 1 by initialising the design space, represented by the variable SUM, with the input VPC. It then iteratively adds unsplittable VPCs to this set until no more states can be split. A state is split by separating its outgoing transition t and its send-nonequivalent transitions, if there are any. Once a t that has one or more send-nonequivalent transitions in $S^T(t)$ is found, then it generates VPC$_{\text{temp}}$ by removing t from c in Line 6. If VPC$_{\text{temp}}$ is not already in the design space (Line 8), then it is added. For each send-nonequivalent transition $t' \in S^T(t)$, a temporary VPC is generated by removing t' from c, as shown on Line 13 and 14. Removing a transition may result in states becoming unreachable. The reachability condition in the definition of a UVPC can be divided into two parts: first, all states should be reachable from the initial state, and second, the final state should be reachable from all states (hence no livelock can occur). To satisfy the reachability condition, we remove livelocks and unreachable states in Line 7 and 15. Algorithms to remove livelocks can be found in [38] (the algorithms are called the *Livelock_Test Algorithm* and *Starvation_Test Algorithm* in that work). If the resulting VPC$_{\text{temp}}$ is not empty and not already in the design space (Line 8), then this VPC is added (Line 17), and the loop over transitions terminates. If all possible unsplittable VPCs have been added to SUM, then the original (splittable) VPC is removed (Line 26). Otherwise one-step probabilities are added to *c* (Line 28). When all VPCs in SUM have been split, and no new VPCs can be generated, the algorithm terminates.

Algorithm 4 Design-Space Generator

```
Input: VPC: A verified parallel composition, $\mathcal{P}$: one-step probabilities, $S^T(t)$: a set of
    send-nonequivalent transitions of each transition $t$ in the VPC
Output: SUM: A set of UVPCs
 1: SUM = {VPC}
 2: for each UVPC $c \in$ SUM do
 3:     flag = false
 4:     for each transition $t$ in $c$ and $S^T(t) \neq \emptyset$ do
 5:         VPC_temp=c
 6:         remove $t$ from VPC_temp
 7:         remove livelocks, and unreachable states in VPC_temp
 8:         if VPC_temp is not empty and not in SUM then
 9:             add VPC_temp to SUM
10:             flag = true
11:         end if
12:         for each $t' \in S^T(t)$ do
13:             VPC_temp=c
14:             remove $t'$ from VPC_temp
15:             remove livelocks, and unreachable states in VPC_temp
16:             if VPC_temp is not empty and not in SUM then
17:                 add VPC_temp to SUM
18:                 flag = true
19:             end if
20:         end for
21:         if flag then
22:             BREAK
23:         end if
24:     end for
25:     if flag then
26:         remove $c$ from SUM
27:     else
28:         add probabilities in $c$
29:     end if
30: end for
```

Example 5 Given the fragment VPC shown in Fig. 8, using Algorithm 3 we detect a controllable state $s_{3,2,0}$. For example, receiving ava^a from the *Airline*, the VPC can choose between sending $offer^t$ or $wait^t$ to the *Travel Agency*. Hence the transition from $s_{3,2,0}$ to $s_{5,3,0}$ and the transition from $s_{3,2,0}$ to $s_{3,3,0}$ are send-nonequivalent. Similarly, the transition from $s_{3,2,0}$ to $s_{4,4,0}$ and the one from $s_{3,2,0}$ to $s_{3,4,0}$ are send-nonequivalent.

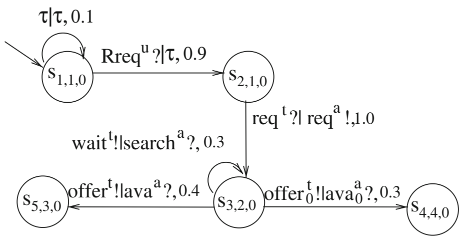


Fig. 19 A fragment UVPC$_1$ derived from the VPC shown in Fig. 8

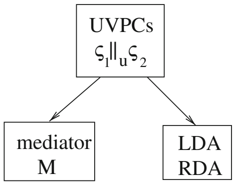


Fig. 20 A formal derivation of mediators

Given the one-step probabilities of each behaviour in the VPC, we can apply Algorithm 4. This algorithm splits the state $s_{3,2,0}$, and results in four UVPC fragments. These correspond to the four combinations of children of $s_{3,2,0}$: (i) $s_{3,2,0}$, $s_{5,3,0}$ and $s_{4,4,0}$, (ii) $s_{3,2,0}$, $s_{3,3,0}$ and $s_{4,4,0}$, (iii) $s_{3,2,0}$, $s_{3,3,0}$ and $s_{3,4,0}$ and (iv) $s_{3,2,0}$, $s_{5,3,0}$ and $s_{3,4,0}$. Each of these four UVPC fragments have the same number of states and transitions. As an example, we show in Fig. 19 the fragment corresponding to combination (i) above, which we shall call UVPC$_1$.

5 The Estimator phase

In Fig. 18 we saw that a design space of UVPCs are generated in the Builder phase. In this section we describe how the user chooses the best mediator from the design space, given a set of non-functional requirements. A UVPC may contain a very large number of states because the buffer contents are included in state representations. To reduce the number of states, states in the UVPC are merged by abstracting out the contents of the buffers. We depict this in Fig. 20 where we see the result of merging states in the UVPC is a mediator M, and also two models of its behaviour, referred to as the LDA, which stands for *Local data abstraction*, and RDA, which stands for *Remote data abstractions*. We saw in Definition 6 that local and remote data transfer actions in a UVPC are separated, and both can use buffers, so the abstracted buffers may correspond to either local or remote data transfers. The abstractions LDA and RDA in fact partition the 'buffered' and 'unbuffered' behaviour of the mediator into its local and remote components. A state in a data abstraction corresponds to a set of states in the mediator that vary only in the buffer size. A mediator is defined formally as follows.

Definition 7 *Mediator* Given a UVPC $\langle B, U, L^u, R^u, \mathcal{P}, \varrho, s_{0,0}\rangle$, a mediator is represented by a finite-state machine $M = \langle S^z, L^z, R^z, \delta^z, s_0^z\rangle$, where:

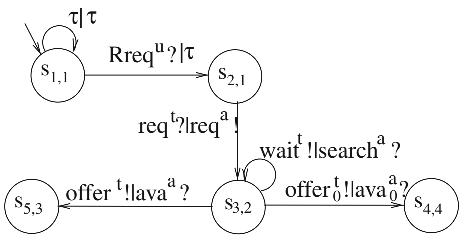


Fig. 21 A fragment of a mediator derived from the UVPC shown in Fig. 19

- S^z is generated from the states of U by removing the buffer sizes B. For each state $s_x \in S^z$, we have a set of states $s_{x,k} \in U$.
- $L^z = L^u$ and $R^z = R^u$.
- $\delta^z \subseteq S^z \times 2^{L^z} \times 2^{R^z} \rightarrow S^z$ is a function that labels transitions between states. For each transition $(s_{x,k}, A_{x,y}, \mathcal{P}_{(x,k),(y,l)}, s_{y,l}) \in \varrho$ we have $(s_x, A_{x,y}, s_y) \in \delta$.
- s_0^z is the initial and final state $s_{0,0}$ without buffer size.

Definition 8 *Local and remote data abstraction* A local data abstraction is given by LDA= $\langle B, L_d^u, \sigma, 0\rangle$, where:

- L_d^u is a set of local data actions of the mediator.
- $\sigma \subseteq B \times 2^{L_d^u} \rightarrow B$ is a function that labels transitions between number of data sizes. Each transition in σ is written (k, D, l), where $k, l \in B$, $D = A_{x,y} \cap L_d^u$ and $(s_{x,k}, A_{x,y}, \mathcal{P}_{(x,k),(y,l)}, s_{y,l}) \in \varrho$ in the mediator.
- 0 denotes the empty data size, and is the initial and the final value of the data size.

A remote data abstraction RDA= $\langle B, R_d^u, \sigma, 0\rangle$ is defined analogously, but the remote data actions R_d^u are used instead of the local data actions.

Example 6 After merging states, the corresponding mediator fragments that are derived from our UVPCs in Example 5 will contain the same number of states. No state can be merged in these fragments because for simplicity we have assumed that no data needs to be stored during the communication. The (fragment of the) mediator generated from the UVPC fragment shown in Fig. 19 can be seen in Fig. 21.

Note that the local data abstractions actually represent buffers in the mediator, and the remote data abstractions represent the storage in remote communication. Given the mediator model, we now present a performance model to estimate performance for each mediator and corresponding data abstractions.

5.1 Performance model

In Fig. 22 we illustrate the steps involved to estimate performance. Given a Markov model of services, we first calculate the steady-state and steady-transition probabilities. Using these probabilities, we then compute the data throughput and the energy cost by defining transition weights and determining the Hamming distances between states. These quantities are then used to estimate the performance of each of the mediators in the design space, and allow the user to select the best design.

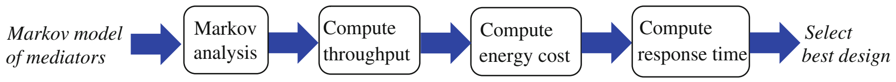


Fig. 22 The four steps involved in performance modelling

Intuitively, if we look at the VPC in Fig. 8, at state $s_{3,2,0}$, after receiving message ava^a the mediator could choose to send message $offer^t$ or $wait^t$, and reach state $s_{5,3,0}$ or $s_{3,3,0}$ respectively. Although both alternatives are correct, they will result in different performance. With no wait, the first alternative will finish the transaction using less states, and hence take shorter time. It also consumes less energy and results in a shorter response time. Therefore the mediator design that takes the first alternative, as shown in Fig. 21, will perform across the board better than the design that takes the second (the figure is not shown here).

The first three steps in Fig. 22 are similar to those required in hardware design [11,12], and hence the treatment here will be brief. The fourth step, which computes the response time of the web service, is new to this work. We describe each of the steps below. Once the system has generated the performance rankings, it is the engineer's responsibility to select the best mediator.

5.1.1 Step 1: Markov analysis

In this first step the probabilities in the UVPC are used to compute the steady-state probabilities $\mathcal{L}^u$ and steady-transition probabilities $\mathcal{T}^u$ of the UVPC. In turn, these probabilities are used to calculate the steady-state probabilities $\mathcal{L}^s$ and steady-transition probabilities $\mathcal{T}^s$ of each of the mediators. They are also used to calculate the steady-state probabilities of the corresponding local and remote data abstractions, $\mathcal{L}^{b_L}$ and $\mathcal{L}^{b_R}$, and similarly the steady-transition probabilities $\mathcal{T}^{b_L}$ and $\mathcal{T}^{b_R}$. This calculation involves expressing the LDAand RDAas a set of equations, called the *Chapman-Kolmogorov equations*.

Example 7 In Example 5 we showed how the UVPC fragment UVPC$_1$ was generated. To compute the steady-state probability $\mathcal{L}^u(s_{3,2,0})$ of state $s_{3,2,0}$, we must solve the (Chapman-Kolmogorov) equations:

$$\mathcal{L}^u(s_{2,1,0}) = 0.9\mathcal{L}^u(s_{1,1,0})$$
$$\mathcal{L}^u(s_{3,2,0}) = 0.3\mathcal{L}^u(s_{3,2,0}) + \mathcal{L}^u(s_{2,1,0})$$

and $\sum_{s_{i,j,k}} \mathcal{L}^u(s_{i,j,k}) = 1$. (The complete set of equations for the system will be shown in Sect. 6.) The result is $\mathcal{L}^u(s_{3,2,0}) = 0.203$. Using this steady-state probability, the steady-transition probabilities $\mathcal{T}^u$ of the transitions leaving this state can be computed. For example, $\mathcal{T}^u(s_{3,2,0} \rightarrow s_{5,3,0}) = 0.203 \times 0.4 = 0.081$ and $\mathcal{T}^u(s_{3,2,0} \rightarrow s_{4,4,0}) = 0.203 \times 0.3 = 0.061$.

We use the results of the Markov analysis and consider three measures of performance of a web service: the amount of data throughput, the energy consumption and the response time.

5.1.2 Step 2: compute the data throughput

The data throughput is the amount of data that is transferred through the mediator per time unit and is given by $\sum_{s_i,s_j} \mathcal{T}_{i,j} D_{i,j}$, where $\mathcal{T}_{i,j}$ is the steady-transition probability of the transition from state s_i to s_j, and $D_{i,j}$ is the number of data units that have been transferred between these states. The abstraction LDA records the data transferred as a result of local data actions, and hence we can compute the DTR of the local communication using an LDA. This is called the *Local data transfer rate*, and denoted LDTR. Similarly, the *Remote data transfer rate* is denoted RDTR, and is computed using the abstraction RDA.

In a local data abstraction LDA=$\langle B, L_d^u, \sigma, 0\rangle$, we have sets of input and output local data actions, $L_d^{I_u}, L_d^{O_u} \subseteq L_d^u$ resp., and a transition $t_{k,l}$ between buffer sizes k and l with label $L_{k,l}$. The number of data units that are transferred in the transition can be expressed by $(d_1 \times L_{k,l}^I + d_2 \times L_{k,l}^O)$, where d_1 and d_2 are the sizes of input and output local data messages.

Given the steady-transition probability of the LDA, denoted by $\mathcal{T}^{b_L}$, the local data transfer rate is defined as:

$$\text{LDTR} = \sum_{\forall \text{LDA}} \sum_{t_{k,l} \in \sigma} \mathcal{T}_{k,l}^{b_L} (d_1 \times L_{k,l}^I + d_2 \times L_{k,l}^O)$$

Similarly, for a remote data abstraction RDA=$\langle B, R_d^u, \sigma, 0\rangle$, we have sets of input and output remote data actions $R_d^{I_u}, R_d^{O_u} \subseteq R_d^u$, which have data message sizes d_3 and d_4 resp. Given the steady-transition probability of the RDA, denoted $\mathcal{T}^{b_R}$, the remote data transfer rate is defined as:

$$\text{RDTR} = \sum_{\forall \text{RDA}} \sum_{t_{k,l} \in \sigma} \mathcal{T}_{k,l}^{b_R} (d_3 \times R_{k,l}^I + d_4 \times R_{k,l}^O)$$

Given LDTR and RDTR, the total DTR of the mediator is defined as:

$$\text{DTR} = \text{LDTR} + \text{RDTR} \tag{4}$$

(In Sect. 6 we will see an example of this computation.)

5.1.3 Step 3: compute the energy consumption

The energy consumption q^{energy} is the power q^{pow} consumed in a given time interval. We assume that the time taken by a web service is proportional to the number of transitions that have carried out, denoted by $|\delta^z|$. We can therefore express the energy consumption as:

$$q^{energy} = q^{pow} \times |\delta^z|$$

Given a mediator $\langle S^z, L^z, R^z, \delta^z, s_0^z \rangle$ with local data abstractions LDA $= \langle B, L_d^u, \sigma, 0 \rangle$ and remote data abstractions RDA $= \langle B, R_d^u, \sigma, 0 \rangle$, we compute the power consumption using the formula [12,13]:

$$q^{pow} = \sum_{t_{i,j} \in \delta^z} \mathcal{W}_{i,j}^{log} \mathcal{H}_{i,j} + \text{DTR} + \text{DSR} \tag{5}$$

Given a mediator $\langle S^z, L^z, R^z, \delta^z, s_0^z \rangle$ with local and remote control actions $L_c^z \subseteq L^z$ and $R_c^z \subseteq R^z$, then:

$$\mathcal{W}_{i,j}^{log} = \mathcal{T}^s(t_{i,j})(1 + |L_{i,j} \cap L_c^z| + |R_{i,j} \cap R_c^z|)$$

is the transition weight from state s_i to s_j in S^z, and $\mathcal{H}_{i,j}$ is the Hamming distance of the transition between states s_i and s_j. DTR is calculated using Eq. (4), and DSR is the *data storage rate*, which is defined as:

$$\text{DSR} = \sum_{\forall \text{LDA}} \sum_{k \in B} k \times \mathcal{L}^{b_L}(k) + \sum_{\forall \text{RDA}} \sum_{k \in B} k \times \mathcal{L}^{b_R}(k)$$

where $\mathcal{L}^{b_L}$ and $\mathcal{L}^{b_R}$ is the steady-state probability of LDA and RDA resp. In fact, the first term in Eq. (5) calculates the power consumed by the unbuffered part of a mediator, and the second and third terms calculate the power consumed by the buffers in the mediator.

Example 8 In Example 7 we computed the steady-transition probability $\mathcal{T}^s(t_{i,j}) = 0.81$ of the transition between state $s_{3,2}$ and state $s_{5,3}$ in the mediator fragment in Fig. 21. There are two local control actions $offer^t!$ and $ava^a?$ on this transition. Using the formulation above, the weight $\mathcal{W}_{i,j}^{log}$ of this transition is given by $0.081 \times (1+2) = 0.243$. (The weights of all the transitions in M_1 will be shown in Sect. 6.)

5.1.4 Step 4: compute the response time

In the case study, the users of the web service send *requests* to the *Travel Agency*. We define a *request* to be a member of a set L^{I_z} that consists of local input control and data actions, which is a subset of all input actions. Corresponding to a request, there is a *reaction*, which is a member of a set L^{O_z} (consisting of local output control and data actions), which is a subset of all output actions. Corresponding to each request, denoted by $\gamma_i \in L^{I_z}$, there may be any number of possible reactions, given by the set $\mathcal{A}_i \subseteq L^{O_z}$. We denote each reaction $\alpha_j^i \in \mathcal{A}_i$.

We define the *response time* of a given request as the time a mediator takes to send a reaction in response to the request. The response time between a request $\gamma_i \in L^{I_z}$ and each possible reaction $\alpha_j^i \in \mathcal{A}_i$ is denoted by $\mathcal{R}(\alpha_j^i)$. If we have computed $\mathcal{R}(\alpha_j^i)$ for each $\alpha_j^i \in \mathcal{A}_i$, then we can compute the total response time of the request γ_i by:

$$\mathcal{R}^i = \sum_{\alpha^i_j \in \mathcal{A}_i} \mathcal{R}(\alpha^i_j) \tag{6}$$

To compute the response time $\mathcal{R}(\alpha^i_j)$, we need to introduce the concept of a *dialogue*. A dialogue $\mathfrak{D}^i_j$ is a finite-state machine that starts from the state whose incoming transition is labelled by γ_i, and ends at the transition whose incoming transition is labelled by α^i_j. It is formally defined as follows.

Definition 9 *Dialogue* We are given a mediator $\langle S^z, L^z, R^z, \delta^z, s^z_0\rangle$, a request $\gamma_i \in L^{I_z}$, and a set of corresponding reactions $\mathcal{A}_i \subseteq L^{O_z}$, where $L^{I_z}, L^{O_z} \subseteq L^z$ is a set of local input and output actions respectively. Request γ_i occurs on transition $t_{l,m} \in \delta^z$ between states $s_l \in S^z$ and $s_m \in S^z$, and a reaction $\alpha^i_j \in \mathcal{A}_i$ occurs on transition $t_{u,v} \in \delta^z$ between states $s_u \in S^z$ and $s_v \in S^z$. These states can be identical. A dialogue, denoted by $\mathfrak{D}^i_j$, is a finite-state machine $\langle S^d, \Sigma^d, \delta^d, s^d_0, s^d_f\rangle$ where:

- $S^d \subseteq S^z$ is a set of states, where $s_m, s_u, s_v \in S^d$, and for each state $s \in S^z$ where s is reachable from s_m without visiting s_u, and s_u is reachable from s, we have $s \in S^d$.
- $\Sigma^d \subseteq L^z \cup R^z$
- $\delta^d \subseteq S^d \times 2^{\Sigma^d} \times S^d$ and $\delta^d \subseteq \delta^z$.
- $s^d_0 = s_m$ is the initial state
- $s^d_f = s_v$ is the final state

Given a dialogue $\mathfrak{D}^i_j = \langle S^d, \Sigma^d, \delta^d, s^d_0, s^d_f\rangle$ and steady-transition probabilities $\mathcal{T}^s$, the response time $\mathcal{R}(\alpha^i_j)$ between request γ_i and the corresponding reaction $\alpha^i_j \in \mathcal{A}_i$ is the time to execute the dialogue $\mathfrak{D}^i_j$, which is computed by:

$$\mathcal{R}(\alpha^i_j) = \sum_{t_{x,y} \in \delta^d} \mathcal{T}^s_{x,y} |\delta^z| \tag{7}$$

where $|\delta^z|$ is the total number of transitions of the underlying mediator. We use $|\delta^z|$ instead of $|\delta^d|$ here because the calculation of the steady-transition probability $\mathcal{T}^s$ is based on the underlying mediator. Note that if $s_l = s_u$ and $s_m = s_v$, then the dialogue is empty, and the corresponding response time is zero.

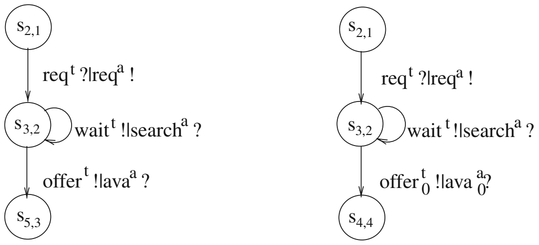
Given $\mathcal{R}(\alpha^i_j)$ of each $\alpha^i_j \in \mathcal{A}_i$, the response time $\mathcal{R}^i$ of the request γ_i can be computed using Eq. (6). If we assume there are n requests in the mediator, after calculating the response time $\mathcal{R}^i$ of all the requests $\gamma_i \in L^{I_z}$, the *average response time*, denoted by r, of all requests and hence of the mediator, is given by:

$$r = \frac{\sum_{\forall \gamma_i \in L^{I_z}} \mathcal{R}^i}{n} \tag{8}$$

To see the variation in the average response time, we calculate the standard deviation $\sigma(r)$ of r as follows.

$$\sigma(r) = \sqrt{\frac{\sum_{\forall \gamma_i \in L^{I_z}} (\mathcal{R}^i - r)^2}{n}} \tag{9}$$

Fig. 23 Two dialogues $\mathfrak{D}_1^1$ and $\mathfrak{D}_2^1$ (*left* and *right*) for request γ_1



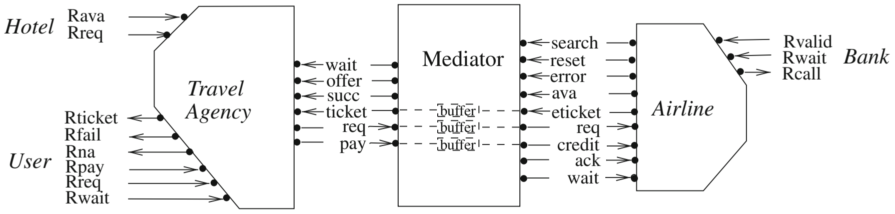


Fig. 24 The transaction interfaces for a mediator between a *Travel Agency* and an *Airline*

Example 9 In the fragment mediator in Fig. 21, a request $\gamma_1 = req^t?$ with two corresponding reactions $\alpha_1^1 = offer^t!$ and $\alpha_2^1 = offer_0^t!$ are involved. For request γ_1 and reactions α_1^1 and α_2^1, the corresponding dialogue $\mathfrak{D}_1^1$ and $\mathfrak{D}_2^1$ are shown in Fig. 23. Using dialogue $\mathfrak{D}_1^1$ and Eq. (7), the response time between γ_1 and α_1^1 is computed by $\mathcal{R}(\alpha_1^1) = (0.061 + 0.081) \times 18 = 2.556$. Similarly for α_2^1, we compute $\mathcal{R}(\alpha_2^1) = (0.061 + 0.061) \times 18 = 2.196$. Using Eq. (6), the response time of request γ_1 is $\mathcal{R}^1 = \mathcal{R}(\alpha_1^1) + \mathcal{R}(\alpha_2^1) = 4.752$.

6 The *Travel Agency* and *Airline* service case study

In this section we present the complete case study of the *Travel-Agency* and *Airline* services shown in Fig. 2. The transaction interfaces between these services are shown in Fig. 24. We remind the reader that transactions between the remote and local services have the prefix ‘R’, and this differentiates them from local transactions. In the remote services in the figure we cannot see which transactions are for data and which are for control. This will be defined in the individual specification of these remote services. We first define the *Travel-Agency* service.

Travel-Agency **service** the specification of the *Travel-Agency* service is represented by $\varsigma_1 = \langle S, L, R, \delta, s_0\rangle$ where:

- $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}\}$
- $L = L^{I_c} \cup L^{O_c} \cup L^{I_d} \cup L^{O_d}$, where:
  - $L^{I_c} = \{offer^t, wait^t, succ^t\}$
  - $L^{O_c} = \emptyset$
  - $L^{I_d} = \{ticket^t\}$
  - $L^{O_d} = \{req^t, pay^t\}$
- $R = R^{I_c} \cup R^{O_c} \cup R^{I_d} \cup R^{O_d}$, where:

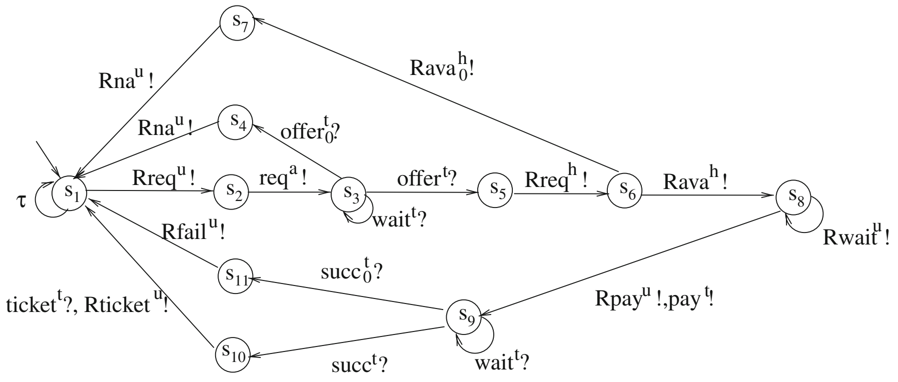


Fig. 25 A specification of the behaviour of the *Travel-Agency* service

- $R^{I_c} = \{Rava^h, Rwait^u\}$
- $R^{O_c} = \{Rfail^u, Rna^u\}$
- $R^{I_d} = \{Rpay^u, Rreq^u\}$
- $R^{O_d} = \{Rticket^u, Rreq^h\}$

– $\delta = \{s_1 \xrightarrow{Rreq^u?} s_2, s_2 \xrightarrow{req^t!} s_3, \text{etc}\}$
– $s_0 = s_1$

This specification is shown graphically in Fig. 25.

A full list of services and their denotation is as follows:

Travel-Agency	t
Airline	a
Hotel	h
User	u
Bank	b

For example, '$Rreq^u$?' is a remote input action for the *User* service, and 'req^t!' is a local output action for the *Travel-Agency* service. In web services, actions generally have Boolean value true. If a control action is false, this is indicated by the subscript zero. Control actions that have no subscript are assumed to be true. For example, the control action '$succ^t_0$?' corresponds to a false on the local '$succ$' interface, whereas the control action '$succ^t$?' corresponds to a true and is equivalent to '$succ^t_1$?'.

The *Travel-Agency* service receives requests for flights and hotels '$Rreq^u$' from a user, and then waits until a result from the *Airline* service is received. The result can be either a possible flight offer, denoted by '$offer^t$', or an indication that the flight is not available, denoted by '$offer^t_0$' (as shown in Example 2). If the flight is not available, then the *Travel Agency* will notify the user with remote signal 'Rna^u'. Otherwise, the *Travel Agency* will continue with the hotel request by sending the request '$Rreq^h$' to the hotel service. If the hotel service replies with '$Rava^h_0$'. which says the room is

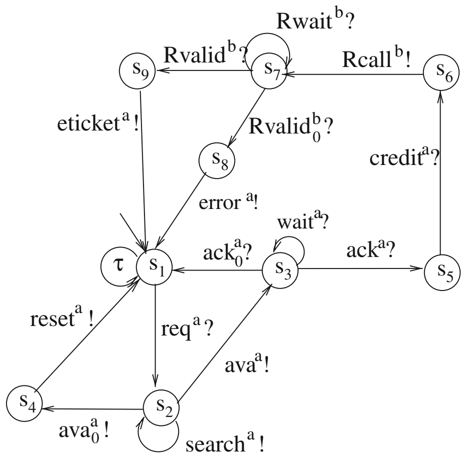


Fig. 26 A specification of the behaviour of the *Airline* service

unavailable, then the *Travel Agency* will notify the user and terminate. Otherwise, the *Travel Agency* waits for the user to pay, and sends payment details 'pay^t' to the *Airline* service. If payment is successful, the *Travel Agency* will receive a ticket '$ticket^t$' from the *Airline* service and will then forward '$Rticket^u$' to the user.

Note that in Fig. 25, states s_1, s_6 and s_8 are uncontrollable. Consider s_6 for example. This state has two outgoing transitions labelled by different remote actions '$Rava^h$?' and '$Rava_0^h$?', but it has no local actions, so this state satisfies the conditions in Definition 2 of an uncontrollable service.

We next consider the *Airline* service, and note that the fragment shown in Fig. 5 is taken from this service.

***Airline* service** the complete behaviour of the Airline service is shown graphically in Fig. 26 and the specification is given by $\varsigma = \langle S, L, R, \delta, s_0 \rangle$ where:

- $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$
- $L = L^{I_c} \cup L^{O_c} \cup L^{I_d} \cup L^{O_d}$, where
  - $L^{I_c} = \{ack^a, wait^a\}$
  - $L^{O_c} = \{reset^a, search^a, ava^a, error^a\}$
  - $L^{I_d} = \{req^a, credit^a\}$
  - $L^{O_d} = \{eticket^a\}$
- $R = R^{I_c} \cup R^{O_c} \cup R^{I_d} \cup R^{O_d}$, where
  - $R^{I_c} = \{Rvalid^b, Rwait^b\}$
  - $R^{O_c} = \{rcall^b\}$
  - $R^{I_d} = \emptyset$
  - $R^{O_d} = \emptyset$
- $\delta = \{s_1 \xrightarrow{req^a?} s_2, s_2 \xrightarrow{ava^a!} s_3, \text{etc}\}$
- $s_0 = s_1$

There are three data channels in the example shown in Fig. 24, each with its own buffer. In one of the channels, a request for a flight from the *Travel Agency* is relayed to the *Airline* (i.e. 'req^t' is translated into 'req^a'). In another channel, the payment from the *Travel Agency* is sent to the *Airline* ('pay^t' is translated into '$credit^a$'). In response, in the third channel, the flight ticket is sent back to the *Travel Agency* ('$eticket^a$' is translated into '$ticket^t$').

The functional requirements that we will use to synthesise a mediator in this case study are shown below.

ϕ_1. $AG(s_{1,1,000} \rightarrow AXAFs_{1,1,000})$: from the initial state, the final state can always eventually be reached, which means every state is reachable and no deadlock occurs.

ϕ_2. $AG(s_{1,1,000} \rightarrow AXA(\neg offer^t \; U \; ava^a))$: from the initial state, the mediator cannot send '$offer^t$' to the Agency until 'ava^a' has been sent by the *Airline* service.

ϕ_3. $AG(s_{1,1,000} \rightarrow AXA(\neg offer^t_0 \; U \; ava^a_0))$: from the initial state, the mediator cannot send '$offer^t_0$' to the Agency until 'ava^a_0' has been sent by the *Airline* service.

ϕ_4. $AG(s_{1,1,000} \rightarrow AXA(\neg ack^a \; U \; Rava^h))$: from the initial state, the mediator cannot send 'ack^a' to the *Airline* service until the Agency receives remote action '$Rava^h$'. This is similar to ϕ_2, but an acknowledgement replaces an offer.

ϕ_5. $AG(s_{1,1,000} \rightarrow AXA(\neg ack^a_0 \; U \; Rava^h_0))$: from the initial state, the mediator cannot send 'ack^a_0' to the *Airline* service until the Agency receives remote action '$Rava^h_0$'.

ϕ_6. $AG(s_{1,1,000} \rightarrow AXA(\neg succ^t \; U \; Rvalid^b))$: from the initial state, the mediator cannot send '$succ^t$' to the Agency until the *Airline* service receives remote action '$Rvalid^b$'.

ϕ_7. $AG(s_{1,1,000} \rightarrow AXA(\neg succ^t_0 \; U \; Rvalid^b_0))$: from the initial state, the mediator cannot send '$succ^t_0$' to the Agency until the *Airline* service receives remote action '$Rvalid^b_0$'.

ϕ_8. The minimum value of k^{m} (which denotes the maximum buffer size) is 1. Formally, for each state $s_{i,j,k_1k_2k_3} \in S^r$ we have $0 \le k_1 \le 1$, $0 \le k_2 \le 1$ and $0 \le k_3 \le 1$: where we assume the input/output data message sizes are the same. This means that there can be no buffer overflow and underflow.

The resulting VPC for the *Travel-Agency* and *Airline* services is shown in Fig. 27. It has 30 states in total.

Building the design space given a VPC for the combined *Travel-Agency* and *Airline* service shown in Fig. 27 and the one-step probabilities of behaviours, applying Algorithm 4 results in a design space comprising 240 UVPCs. The UVPC in the design space that has the minimum number of states (12 states), denoted $\mathrm{UVPC}_{\mathrm{min}}$, is shown in Fig. 28. The UVPC with the maximum number of states (25 states) $\mathrm{UVPC}_{\mathrm{max}}$is shown in Fig. 29. Note that all states in $\mathrm{UVPC}_{\mathrm{min}}$have an empty buffer, hence this UVPC does not require a buffer. In contrast, $\mathrm{UVPC}_{\mathrm{max}}$uses all three buffers.

Merging the states in each of the 240 UVPCs in the design space results in 85 different mediators being generated, which we denote $M_1 \ldots M_{85}$, where the mediators are ordered from smallest to largest in size (measured in terms of the number of transitions).

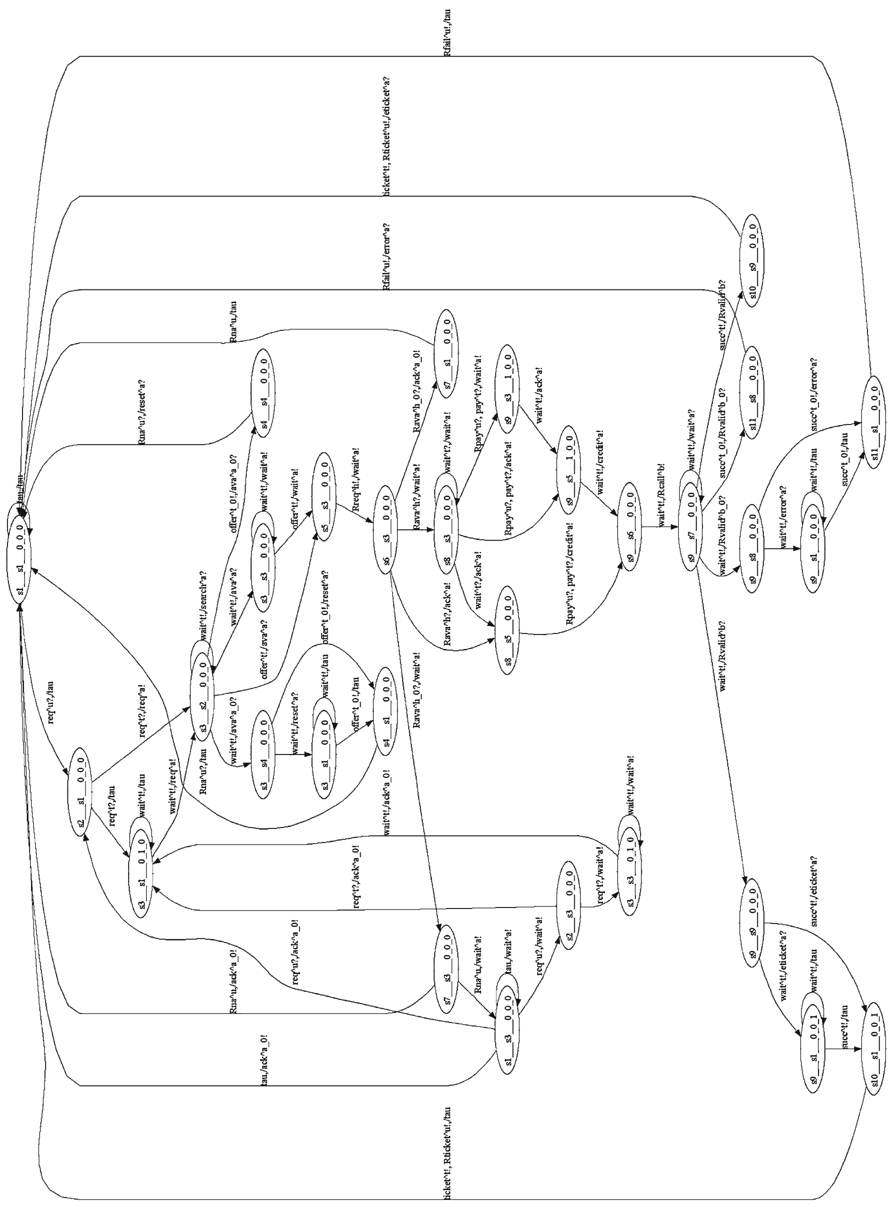


Fig. 27 A VPC for a *Travel Agency* and *Airline* service

The smallest UVPC, which we called UVPC_{min}, and shown in Fig. 28, has three buffers that are always empty, so no states can be merged. The mediator with the smallest number of transitions for this service therefore is $M_1 = \text{UVPC}_{\text{min}}$. The largest UVPC, UVPC_{max}, which is shown in Fig. 29, has three buffers. Merging states in this UVPC results in the mediator M_{85}.

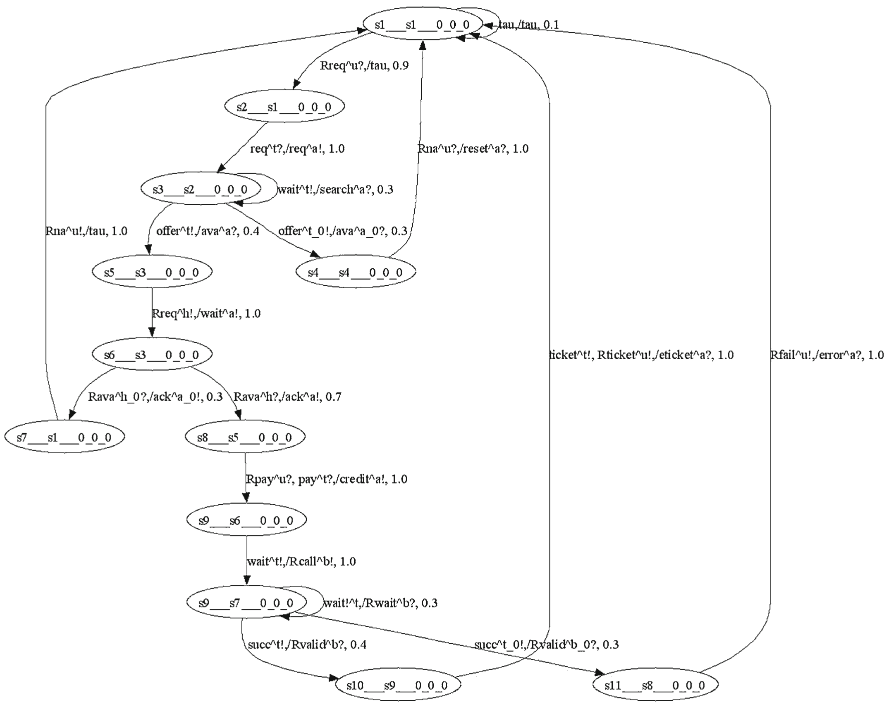


Fig. 28 An unsplittable VPC, called UVPC$_{\text{min}}$, for the *Travel-Agency* and *Airline* services

We now estimate the performance of each mediator using the design framework shown in Fig. 22.

Step 1: to compute the steady-state probabilities $\mathcal{L}^u$ and steady-transition probabilities $\mathcal{T}^u$ of the UVPC$_{\text{min}}$for example, we must solve the equations:

$$\begin{aligned}
\mathcal{L}^u(s_{1,1}) &= \mathcal{L}^u(s_{7,1}) + \mathcal{L}^u(s_{10,9}) + \mathcal{L}^u(s_{11,8}) + 0.1\mathcal{L}^u(s_{1,1}) + \mathcal{L}^u(s_{4,4})\\
\mathcal{L}^u(s_{2,1}) &= 0.9\mathcal{L}^u(s_{1,1})\\
\mathcal{L}^u(s_{3,2}) &= \mathcal{L}^u(s_{2,1}) + 0.3\mathcal{L}^u(s_{3,2})\\
\mathcal{L}^u(s_{5,3}) &= 0.4\mathcal{L}^u(s_{3,2})\\
\mathcal{L}^u(s_{4,4}) &= 0.3\mathcal{L}^u(s_{3,2})\\
\mathcal{L}^u(s_{6,3}) &= \mathcal{L}^u(s_{5,3})\\
\mathcal{L}^u(s_{7,1}) &= 0.3\mathcal{L}^u(s_{6,3})\\
\mathcal{L}^u(s_{8,5}) &= 0.7\mathcal{L}^u(s_{6,3})\\
\mathcal{L}^u(s_{9,6}) &= \mathcal{L}^u(s_{8,5})\\
\mathcal{L}^u(s_{9,7}) &= 0.3\mathcal{L}^u(s_{9,7}) + \mathcal{L}^u(s_{9,6})\\
\mathcal{L}^u(s_{10,9}) &= 0.4\mathcal{L}^u(s_{9,7})\\
\mathcal{L}^u(s_{11,8}) &= 0.3\mathcal{L}^u(s_{9,7})
\end{aligned}$$

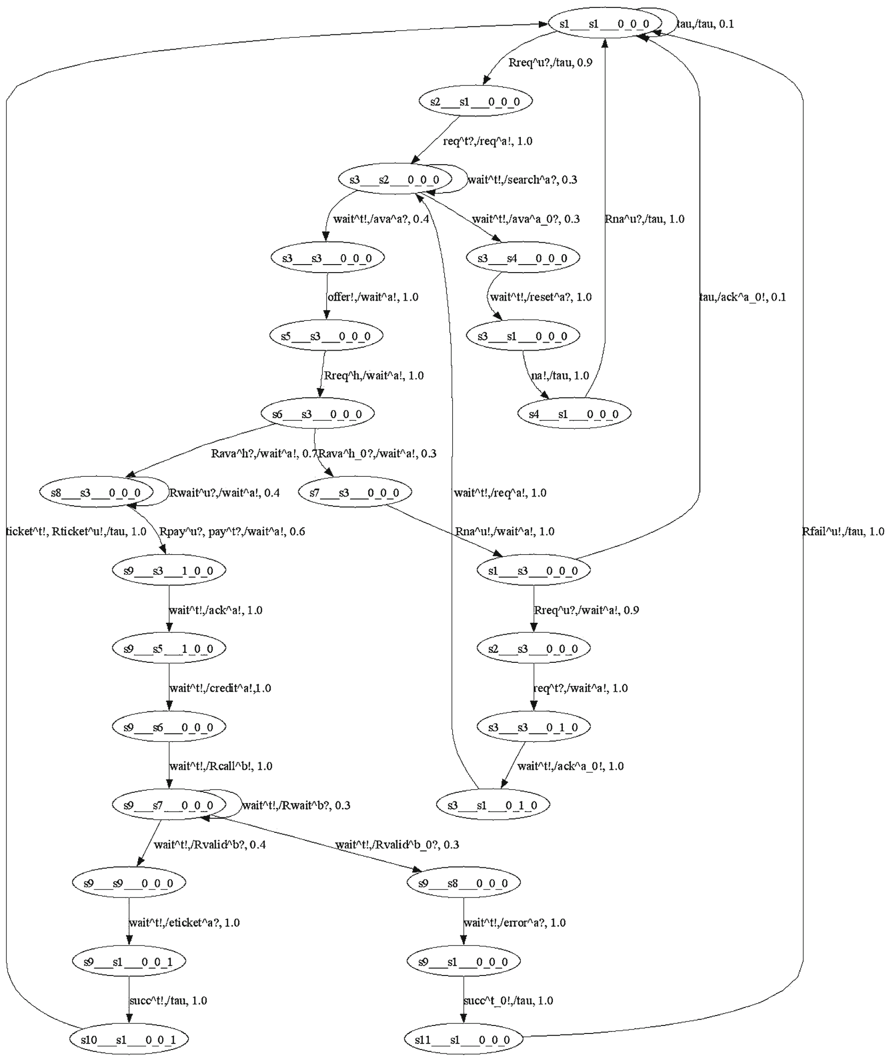


Fig. 29 An unsplittable VPC, called UVPC$_{\max}$, for the *Travel-Agency* and *Airline* services

and $\sum_{s_{i,j}} \mathcal{L}^u(s_{i,j}) = 1$, where each of the three buffers in each state is empty, hence are not shown here. This set of equations has solution:

$$\mathcal{L}^u(U) = [0.157, 0.141, 0.203, 0.081, 0.061, 0.081, 0.024, 0.057, 0.057, 0.081, 0.032, 0.024]$$

where the order of states is the same as the equations' order. The steady-transition probability is computed as:

$$\mathcal{T}^u(\varrho) = \begin{bmatrix} 0.016 & 0.141 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.141 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.061 & 0.081 & 0.061 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.081 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.061 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.024 & 0.057 & 0 & 0 & 0 & 0 \\ 0.024 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.057 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.057 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.024 & 0.033 & 0.024 \\ 0.033 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.024 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where the rows and columns have the same order as $\mathcal{L}^u(U)$. As M_1 = UVPC$_{\min}$, the mediator M_1's steady-state probabilities $\mathcal{L}^s = \mathcal{L}^u$ and the mediator M_1's steady-transition probabilities $\mathcal{T}^s = \mathcal{T}^u$.

Step 2: assuming the data message size of all data actions of the *Travel-Agency* and *Airline* services is 8 units (e.g. characters), we compute DTR for the mediators M_1 and M_{85}. Given the steady-transition probabilities $\mathcal{T}^u$ from the previous step, we compute the DTR for M_1 by DTR $= 0.141 \times 8 + 0.141 \times 16 + 0.081 \times 8 + 0.057 \times 24 + 0.033 \times 24 = 6.06$. The DTR for M_{85} is 4.10.

Step 3: to compute the energy cost, the transition weight of M_1 is:

$$\mathcal{W}^{log} = \begin{bmatrix} 0.016 & 0.141 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.141 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.183 & 0.243 & 0.183 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.162 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.183 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.072 & 0.171 & 0 & 0 & 0 & 0 \\ 0.048 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.057 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.171 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.072 & 0.099 & 0.072 \\ 0.033 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.072 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and buffered power is DTR + DSR = 6.06, where DSR = 0. Hence the total power $q^{pow} = 2.17 + 6.06 = 8.23$. Given $|\delta^z| = 18$ (counting the transitions in Fig. 28), the total energy of M_1 is therefore $q^{energy} = q^{pow} \times |\delta^z| = 148.14$.

Similarly, we compute the energy of M_{85}. Its resulting power is $q^{pow} = 2.85 + 4.10 + 0.40 = 7.35$, and with $|\delta^z| = 33$, the total energy $q^{energy} = q^{pow} \times |\delta^z| = 242.55$.

Step 4: in mediator M_1, the requests and corresponding reactions are listed as follows:

- $\gamma_1 = req^t?$ with reactions $\alpha_1^1 = offer^t!$ and $\alpha_2^1 = offer_0^t!$
- $\gamma_2 = ava^a?$ with reactions $\alpha_1^2 = ack^a!$ and $\alpha_2^2 = ack_0^a!$
- $\gamma_3 = pay^t?$ with reactions $\alpha_1^3 = succ^t!$ and $\alpha_2^3 = succ_0^t!$

Table 1 The performance of 20 different mediators for the *Travel Agency* and *Airline*

mediator	$\lvert\delta^z\rvert$	q^{energy}	r	DTR
M_1	18	148.14	4.35	6.06
M_{16}	20	162.50	5.13	5.77
M_{17}	20	163.34	5.22	5.80
M_{18}	21	163.09	5.10	5.52
M_{19}	21	164.16	5.19	5.50
M_{25}	21	172.29	5.64	5.64
M_{26}	22	171.46	5.60	5.31
M_{27}	22	171.91	5.61	5.31
M_{28}	22	173.28	5.77	5.35
M_{45}	25	184.76	7.21	5.01
M_{46}	25	184.91	7.54	5.05
M_{47}	25	185.10	7.60	5.05
M_{49}	26	185.67	7.78	4.91
M_{50}	26	186.41	7.85	4.98
M_{51}	26	188.52	7.97	5.10
M_{68}	30	219.33	10.13	4.48
M_{69}	30	224.26	10.24	4.52
M_{70}	30	227.36	10.38	4.56
M_{83}	33	238.14	10.71	4.10
M_{85}	33	242.55	11.28	4.10

As shown in Example 9 the response time of request γ_1 is $\mathcal{R}^1 = 4.752$. Similarly, the response time of request γ_2 is computed by $\mathcal{R}^2 = \mathcal{R}(\alpha_1^2) + \mathcal{R}(\alpha_2^2) = (0.105 + 0.138) \times 18 = 4.374$, and of request γ_3 is computed by $\mathcal{R}^3 = \mathcal{R}(\alpha_1^3) + \mathcal{R}(\alpha_2^3) = (0.114+0.105) \times 18 = 3.942$. Therefore the average response time of M_1 is computed by $r = \frac{\mathcal{R}^1+\mathcal{R}^2+\mathcal{R}^3}{3} = 4.35$. Applying Eq. (9), we now calculate the standard deviation of r: $\sigma(r) = \sqrt{\frac{(\mathcal{R}^1-r)^2+(\mathcal{R}^2-r)^2+(\mathcal{R}^3-r)^2}{3}} = 0.33$.

We now compute the response time for M_{85}. The request and corresponding reactions (which are $\gamma_1 \ldots \gamma_3$ and $\mathcal{A}_1 \ldots \mathcal{A}_3$) are the same in M_{85}, but the response time of each request is different. Specifically, request γ_1 is sent twice, and we compute $\mathcal{R}^1 = 0.268 \times 33 = 8.844$ the first time, and $\mathcal{R}^{1'} = 0.296 \times 33 = 9.768$ the second time. We compute it for the other two requests as $\mathcal{R}^2 = 0.429 \times 33 = 14.157$ and $\mathcal{R}^3 = 0.374 \times 33 = 12.342$. The average response time of M_{85} is therefore $r = \frac{\mathcal{R}^1+\mathcal{R}^{1'}+\mathcal{R}^2+\mathcal{R}^3}{4} = 11.28$. The standard deviation of this r is hence $\sigma(r) = \sqrt{\frac{(\mathcal{R}^1-r)^2+(\mathcal{R}^{1'}-r)^2+(\mathcal{R}^2-r)^2+(\mathcal{R}^3-r)^2}{4}} = 2.10$.

This completes all the steps and results in estimates of the energy consumed, the (average) response time and the data throughput. For reasons of space, we show the results for just 20 mediators in Table 1.

In the first column we list the mediators; in the second column the number of transitions ($|\delta^z|$); and in the remaining columns the energy consumption (q^{energy}), average response time (r) and the data throughput (DTR). The mediators are listed in increasing order of the number $|\delta^z|$ of transitions, and increasing order of q^{energy} when $|\delta^z|$ are the same. We make the following observations:

- Generally, as the size $|\delta^z|$ increases, the response time and energy also increase, but the data throughput decreases. There are minor exceptions, which could be attributable to computational errors.
- If we take two designs that have the same number of transitions, then the design that consumes the most energy and has longest response time, also has the highest throughput. The difference is small, but reasonably consistent.
- Overall, the best mediator design is M_1. It has the lowest energy consumption and shortest response time, and the highest throughput. The worst mediator is M_{85}, which has the highest energy consumption and slowest response time, and the lowest throughput.
- By selecting the best mediator, the web-service designer can save 39 % in energy, 65 % in average response time, and improve data throughput by 48 %, compared to the worst mediator.

7 Contribution and future work

The work extends the earlier work of the authors in hardware design [9,10,12]. The main extension is accounting for remote services. Using the Markov analysis, the affect that these remote services can have on the performance of the mediator (they may be slow to react because they need to access a large database for example) can be modelled by choosing appropriate probabilities in the model. Normally the actions of remote processes would be assumed to be constant, or out of scope, but in our approach they play an important role. The ability of the system engineer to model remote processes allows him to experiment with the design: and determine which designs are most effective in dealing with remote servers under different conditions.

It is an open question how closely the performance of the model will match the actual performance of a fully implemented system. In hardware design, an analogous theoretical performance model has been used in [11]. This model was *validated* using hardware simulation tools that are routinely used in hardware design. This validation resulted in all the designs being ranked for each measure of theoretical performance. The designs were then translated to a hardware description language and simulations were carried out. The performance of the simulations of all the designs were then ranked. The *fidelity* between the rankings of the theoretical performance and the actual simulations, where the fidelity is a measure how well the two rankings are correlated, were statistically analysed. This involved computing Spearman's *rank correlation coefficients* [24], and carrying out a *linear regression analysis*. In both cases, the correlation was above 0.9 for each measure of performance, where a correlation of 1 is a perfect fit. It is not possible to use the same validation technique in web-service design. Nor is it possible to know whether the theoretical model we use is equally valid in both hardware and web-service design. That must remain future work.

The measures of performance that have been used in this research have been chosen arbitrarily. Other kinds of performance such as delays, availability and reliability could be modelled. It is an open question which measures of performance are the most predictable. Depending on the systems and user environment, some measures of performance may be more sensitive to changes in the probabilities in the Markov model than others. This is an issue worthy of research as a system engineer would naturally want to know what the ‘sensitive’ areas of a design are. To test the predictions of the performance model, a starting point could be to use the Standard Performance Evaluation Corporation benchmark SPECweb2005 for evaluating the performance of servers.

8 Summary and perspective

In this work, a model checker is used to synthesise a design space of mediators that satisfy temporal properties that are set by an engineer. The engineer can then select the best mediator based on a theoretical model of performance. By using model checking we can guarantee that all the mediator designs behave correctly. The performance model is based on Markov chains. A Markov analysis of various performance measures allows the engineer to rank the mediator designs. Using these rankings, the user must select the design he sees as most suitable for the environment in which the system must function.

A case study involving two local services, namely a *Travel Agency* and an *Airline*, and three remote services, has been used to demonstrate the methodology. We showed how to generate a mediator between the local services. In the first phase of this methodology a verified parallel composition of the services is generated from formal descriptions of the services. In the second phase, a design space consisting of 240 possible mediators is generated. All the mediator designs are ranked in terms of the different measure of performance. In the third phase, a Markov analysis selects the best mediator from the design space. This mediator design has substantially lower energy consumption, faster response time, and higher data throughput than the worst design. Placing this work in context, however, the reader should realise the following:

– The design rankings are theoretical: there is no guarantee that if each of the designs were to be implemented that the actual performances would be consistent with the design rankings.
– The experience with hardware design however is that the theoretical rankings are consistent with the rankings of actual implementations.
– It is well known in software development that decisions made at design-time (which often concern architectural issues) generally have a greater impact on the performance than those made at implementation time (which concern low-level details). This suggests that design-time performance estimation has validity.
– Being able to compare the (theoretical) performance of all possible designs is a luxury rarely afforded to system engineers. Conventionally, a single design is developed and carried through to implementation, and the design is revisited only if errors are found during the development. There is no set of designs for the engineer to compare. Our technology is focussed on design-space exploration and selection.
– It is possible to experiment with different user behaviours in the performance model by changing the probabilities. If the best design is carried through to imple-

mentation, but at some later time, user behaviour changes, then the performance model allows all the designs to be re-compared. With the existing implementation acting as yardstick, it can be estimated whether the development effort is worth the gain in performance offered by a new design.
- The performance model provides the engineer with a tool to forecast the likely behaviour of different designs under different conditions, and thereby make more informed decisions.

References

1. Al-Masri E, Mahmoud Q (2007) QoS-based discovery and ranking of web services. In: Proceedings of 16th IEEE international conference on computer communications and networks, Hawaii USA, pp 529–534
2. Ardagna D, Tanelli M, Lovera M, Zhang L (2010) Black-box performance models for virtualized web service applications. In: Proceedings of the ACM joint WOSP/SIPEW international conference on performance engineering, pp 153–164
3. Berardi D, Calvanese D, de Giacomo G, Hull R, Mecella M (2005) Automatic composition of transition-based semantic web services with messaging. In: VLDB'05, proceedings of the 31st international conference on very large databases, pp 613–624 (VLDB Endowment)
4. Berardi D, Calvanese D, de Giacomo G, Lenzerini M, Mecella M (2005) Automatic service composition based on behavioral descriptions. Int J Coop Inf Syst 14(4):333–376
5. Bhat G, Cleaveland R, Grumberg O (1995) Efficient on-the-fly model checking for CTL*. In: Proceedings of the 10th annual symposium on logic in computer science, Los Alamitos, pp 388–397
6. Bultan T, Fu X, Hull R, Su J (2003) Conversation specification: a new approach to design and analysis of e-service composition. In: WWW'03, proceedings of the 12th ACM international conference on world wide web, Budapest, pp 403–410
7. Cabral L, Domingue J, Galizia S, Gugliotta A, Tanasescu V, Pedrinaci C, Norton B (2006) IRS-III: a broker for semantic web services based applications. The semantic Web-ISWC, pp 201–214
8. Cao J (2011) A formal verification- and performance-driven design methodology for converters. Phd thesis, Department of Computer Science and Engineering, UNSW, Sydney, Australia
9. Cao J, Nymeyer A (2009) Formal model of a protocol converter. In: CATS'09 15th computing, the Australasian theory symposium, vol 94 of CRPIT, pp 107–117
10. Cao J, Nymeyer A (2010) The 'best' valid safe protocol converter. In: SSIRI'10, 4th IEEE Computer Society international conference on secure software integration and reliability improvement, Singapore, pp 237–243
11. Cao J, Nymeyer A (2010) High-fidelity Markovian power model for protocols. In: DATE'10, ACM proceedings of the conference on design, automation and test in Europe, pp 267–270
12. Cao J, Nymeyer A (2010) A Markov model for low-power high-fidelity design-space exploration. In: DSD'10, 13th Euromicro conference publication services on digital system design, pp 115–122
13. Cao J, Nymeyer A (2011) A Markov performance model for buffered protocol design. In: ISVLSI, IEEE Computer Society annual symposium on VLSI, pp 170–175
14. Casale G, Mi N, Smirni E (2008) Bound analysis of closed queueing networks with workload burstiness. In: SIGMETRICS'08, the proceedings of the ACM international conference on measurement and modeling of computer systems, pp 13–24
15. Casati F, Ilnicki S, Jin L, Krishnamoorthy V, Shan M (2000) Adaptive and dynamic service composition in eFlow. In: Advanced information systems engineering. Springer, pp 13–31
16. Clarke EM, Grumberg O, Peled DA (2000) Model checking. MIT Press, Cambridge
17. Clements P, Northrop L (1996) Software architecture: an executive overview. Technical report CMU/SEI-96-TR-003. Carnegie Mellon University, Pittsburgh, PA
18. de Giacomo G, de Leoni M, Mecella M, Patrizi F (2007) Automatic workflows composition of mobile services. IEEE Computer Society, pp 823–830
19. Deutsch A, Sui L, Vianu V, Zhou D (2006) Verification of communicating data-driven web services. In: Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, pp 90–99

20. Dustdar S, Schreiner W (2005) A survey on web services composition. Int. J Web Grid Serv 1(1):1–30
21. Franks G, Woodside M (1998) Performance of multi-level client-server systems with parallel service operations. In: WOSP '98, proceedings of the ACM 1st international workshop on software and performance, pp 120–130
22. Hermanns H (2002) Interactive Markov chains. Springer, Berlin
23. Ivanovic D, Carro M, Hermenegildo M (2010) Towards data-aware QoS-driven adaptation for service orchestrations. In: Proceedings of 2010 IEEE international conference on web services, ICWS '10. IEEE Computer Society, pp 107–114
24. Javaid H, Ignjatvic A, Parameswaran S (2010) Fidelity metrics for estimation models. In: ICCAD'10, proceedings of the IEEE/ACM international conference on computer-aided design, pp 1–8
25. Kazhamiakin R, Pistore M, Santuari L (2006) Analysis of communication models in web service compositions. In: WWW'06, proceedings of the 15th ACM international conference on world wide web. pp 267–276
26. Kumaran S, Nandi P (2002) Conversational support for web services: the next stage of web services abstraction. http://www.ibm.com/developerworks/webservices/library/ws-conver/.
27. Li L, Yang Y (2008) E-business process modelling with finite state machine based service agents. In: CSCWD'08, computer supported cooperative work in design. Springer, pp 261–272
28. Mitra S, Kumar R, Basu S (July 2007) Automated choreographer synthesis for web services composition using I/O automata. In: ICWS'07, proceedings of the IEEE international conference on web services. pp 364–371
29. Pacifici G, Spreitzer M, Tantawi A, Youssef A (2003) Performance management for cluster based web services. IEEE J Sel Areas Commun 23:2333–2343
30. Paolucci M, Soudry J, Srinivasan N, Sycara K (2004) A broker for OWL-S web services. Extending web services technologies, pp 79–98
31. Peltz C (2003) Web services orchestration and choreography. Computer 36:46–52
32. Petry F, Ladner R, Gupta KM, Moore P, Aha DW (2009) Design of an integrated web services brokering system. Int J Inf Technol Web Eng 4:58–77
33. Rosen M (2008) Applied SOA: service-oriented architecture and design strategies. Wiley, Indianapolis
34. Sangiovanni-Vincentelli A (2007) Quo vadis, SLD? Reasoning about the trends and challenges of system level design. Proc IEEE 95(3):467–506
35. Serhani MA, Dssouli R, Hafid A, Sahraoui H (2005) A QoS broker based architecture for efficient web services selection. In: ICWS'05, proceedings of IEEE international conference on web services. IEEE, pp 113–120
36. Sinha R, Roop PS, Basu S (2008) A model checking approach to protocol conversion. Electr Notes Theor Comput Sci 203(4):81–94
37. Sycara K, Paolucci M, Soudry J, Srinivasan N (2004) Dynamic discovery and coordination of agent-based semantic web services. IEEE Internet Comput 8:66–73
38. Tai K (1994) Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In: ICPP'94, IEEE international conference on parallel processing, vol 2. pp 69–72
39. Tanelli M, Ardagna D, Lovera M, Zhang L (2008) Model identification for energy-aware management of web service systems. In: ICSOC'08, proceedings of the 6th international conferernce on service-oriented computing. Springer, Berlin, pp 599–606
40. ter Beek MH, Bucchiarone A, Gnesi S (2008) Formal methods for service composition. Ann Math Comput Teleinf 1(5):1–10
41. Yu T, Lin K (2004) The design of QoS broker algorithms for QoS-capable web services. In: EEE'04, proceedings of IEEE international conference on e-technology, e-commerce and e-service, pp 17–24
42. Yu T, Lin K (2005) A broker-based framework for QoS-aware web service composition. In: EEE'05, proceedings of IEEE international conferernce on e-technology, e-commerce and e-service. pp 22–29