# Engineering Flexible Service-Oriented Transactions

David Paul(✉) and Frans Henskens

School of Electrical Engineering and Computer Science,
University of Newcastle, Callaghan, NSW 2308, Australia
{David.Paul,Frans.Henskens}@newcastle.edu.au

**Abstract.** The traditional ACID properties for transactions are not always appropriate in service-oriented environments. Instead, it is often preferable to "relax" the transactional guarantees, reducing isolation or atomicity to ensure acceptable performance at a reasonable cost. Existing standards require providers to constantly offer a fixed level of transaction support to each client that requests a particular service. We present a mechanism that allows providers to dynamically alter the level of transaction support offered on a per-service-call basis. Further, we engineer a cost-based model, based on $\pi t-$calculus, that allows clients to automatically reason about workflows consisting of service requests with various levels of transaction support. The viability of this scheme is tested with a Web Services transactions simulator, with results indicating potential benefits for both clients and service providers.

**Keywords:** Web Service · Transaction · Workflow

## 1 Introduction

In service-oriented environments, such as that offered by Web Services, clients use messages to send service requests to providers. The respective providers then respond either that the request was successful, or that the service was not performed. However, it is often the case that no single provider offers a service that completely satisfies a client's requirements. Instead, clients frequently have a need to combine multiple services, possibly from different providers, into a single workflow.

When combining multiple services into a single workflow, it is possible that some services will fail to complete successfully. However, since the client sees its workflow as a single task, the failure of one service may affect the rest of the workflow. For example, when booking a holiday, there is no point in booking a hotel in the destination city if the client is unable to book a flight to take them there.

One possible technique that can be used to ensure correct behaviour of client workflows is transactions. Transactions combine services provided by multiple, possibly completely unrelated, parties into a single action, with well-defined


K.-H. Krempels and A. Stocker (Eds.): WEBIST 2013, LNBIP 189, pp. 37–52, 2014.
DOI: 10.1007/978-3-662-44300-2_3

behaviour whenever a part of the transaction fails. While traditional transactions offer strong guarantees of atomicity or isolation, the restraints these guarantees require are often considered unacceptable in service-oriented environments. Instead, weaker guarantees are often preferred [1,2].

However, different clients may have different views as to which transactional guarantees are important for a given service call. Further, since client workflows can include services from multiple unrelated providers, different levels of transaction support may need to be combined to ensure the client achieves an acceptable result. In this paper we describe a technique that allows providers to dynamically change the level of transaction support they offer for a particular service, and a means for clients to reason about such dynamic workflows. A simulator [3] is used to show the viability of this approach.

## 2 Motivating Example

Consider a public screen shared by multiple companies to display various pieces of information content. The screen provides a Web Service that allows clients to book time on the display. Further, consider a client that wishes to use the screen to inform the public of sporadically-changing data. The client is notified whenever new data are available, and can then use a Web Service to access the data. That data can be passed to a Web Service offered by a render farm to convert it into a visualisation suitable for the public screen. Each Web Service has a cost associated with its use.

On notification that new data are available, the client can retrieve the data, pass the data to the render farm, and display the visualisation at the time the client has the public screen booked. Since the data are updated sporadically, the client cannot book the display in advance, but can only attempt to do so when new data are released. However, since there is a fee associated with each Web Service, the client would prefer to only retrieve and render the data if it is guaranteed time on the display. Similarly, the client would not wish to retrieve the new data if ever the render farm was too busy to process the client's request.

The client can use service composition to achieve its aim of only retrieving data if the render farm is available, and only attempting either action if the screen can be booked, if each of the providers offers suitable levels of transaction support. However, different clients using the shared screen may not all have identical transactional requirements.

## 3 Service Composition Platforms

It is often the case that no single service in a service-oriented environment completely meets a client's needs. Thus it is often necessary for clients to combine multiple services, possibly from multiple providers, to complete the desired actions. Chakraborty and Joshi [4] identify five issues that must be provided by a comprehensive service composition platform so that such dynamic service compositions can be formed. Those requirements are:

**Service Discovery.** Before multiple services can be combined into a service composition, it is necessary to find a service or services that can perform the required tasks. Service discovery requires semantic information so the system can automatically determine which services (or combination of services) offer the functionality that is required by the client. This issue is beyond the scope of this paper, but efforts in the Semantic Web are making such discovery possible for Web Services.

**Uniform Information Exchange Infrastructure.** Different services may operate in different ways (e.g. taking different parameters, or returning information differently), but a service composition platform should abstract over these differences. With Web Services, it is possible to define an abstract service [5] that knows how to interact with each of the different services that offer similar functionality. This abstract service, when requested by a client, contacts actual providers, converting the client's request into the form that is understood by each particular provider, receiving any response(s), and returning them to the client in a uniform format.

**Adaptiveness.** To allow true adaptiveness, it must also be possible to dynamically compose services when necessary to provide required functionality [6]. One approach to implementing dynamic compositions is to have semantic contracts with each provider [7]. Then, by utilising a good service discovery mechanism, an abstract service that performs its own service composition could be created to provide the functionality required by the client.

**Service Coordination and Management.** To combine multiple services into a single composition, it is necessary to be able to communicate with each of the services in the composition to ensure their correct behaviour.

**Fault Tolerance and Scalability.** The system should correctly handle faults, and efficiently use the resources it has available.

Transactions allow multiple actions to be combined into a seemingly single action, can be arbitrarily nested, and each nested transaction can specify how it handles errors. Thus, it is possible to provide the service coordination and management and fault tolerance and scalability requirements of a good service composition platform by supporting transactions in the Web Services environment.

## 4 Web Services Transactions

Transactions typically provide the ACID properties of Atomicity, Consistency, Isolation, and Durability. However, in service-oriented environments, complete support for ACID properties is often undesirable. In these environments, transactions typically have a much longer execution time than is experienced in traditional transaction systems. The different services involved in a transaction may be offered by completely autonomous providers, and communication between these providers already increases the duration of any transaction. The independence of each provider means they will often be unwilling to reduce their autonomy to such an extent that complete ACID support is possible. Thus,

certain reductions to the ACID properties are typically supported in service-oriented environments, such as replacing atomicity with semantic atomicity [1], or offering tentative holds [2].

Most Web Services transactions standards are limited in their ability to allow clients to combine services with different transactional guarantees into a single workflow [8]. Mikalsen et al. introduce a new way to ensure transactional reliability in the Web Services environment. Their technique allows providers to indicate the "transactional attitude" of the provider for each service it provides; that is, the kind of transactional support that the provider offers for the service. Clients also indicate their transactional attitude by utilising a Web Service to create and manage a pre-defined transactional pattern. Once providers and clients have specified their transactional attitudes, middleware is used to automatically manage the context and transactional interaction between a client and a set of providers.

With transactional attitudes, a provider must specify the level of transaction support it offers along with its service definition. This makes it impossible for the provider to alter the level of transaction support it offers for a particular service. In some cases, different clients may require different levels of transactional support for a particular service. Paul et al. [9,10] suggest a method that allows providers to dynamically alter the level of transaction support on offer for a particular service. In this paper, we further argue this point, and describe a formal model that allows arbitrarily complex client workflows to utilise such enhanced transactional properties.

## 5 Engineering Dynamic Transactions

When providers offer a service, there is a level of transactional support they offer along with that service. In the simplest case, there is no explicit transactional support offered; the service either completes successfully or fails. More advanced cases allow the provider to deliver information or guarantees to a client before the client utilises the service, or allow the client to perform the operation but later cancel it if required. This section discusses the levels of transactional support that a provider can offer.

On examining the possible reductions, it becomes apparent most can be described using a combination of five basic operations that can be requested by a client [3]:

**Enquire.** Allows the client to query whether a request would currently be successful, without any guarantee that a later request will succeed.

**Prepare.** Allows the client to query whether a request would currently be successful and, if so, guarantees that any such request sent by the client within a timeout period will succeed. On receipt of a successful reply, the client has the option to cancel the request, which relieves the provider of its responsibility to guarantee the resources to the client.

**Commit.** Performs the client's request. This is the only required operation for a service.

**Compensate.** Performs actions to undo a previously committed request, provided that the call to compensate is received within a timeout period.

**Callback.** Allows a provider to notify a client that has previously received a response from the provider, in the circumstance that the provider's situation has changed, that a revision of the provider's previous response is now available.

Using these five operations, a traditional ACID transaction can be described as a Prepare/Commit pattern with an infinite timeout for the Prepare stage. Similarly, semantic atomicity is provided by having the provider offer a Commit/Compensate pattern, and support for tentative holds is achieved through an Enquire/Callback/Commit pattern.

The concept of resilience [11] cannot be described using these operations. However, replacing concrete services with abstract services [5] can allow resilience in a way that is transparent to both clients and service providers. Clients use the abstract services rather than the services offered by the concrete providers, and the abstract service acts as a broker between all the providers offering alternative services. These abstract services can thus be used in transactions described by the operations defined above.

On becoming aware of the transactional pattern supported by a provider for a particular interaction, a client utilises the service by following through the states depicted in Fig. 1. The interaction begins in the *Initial* state, and a change in state occurs when the client sends a message to the provider, or the provider sends a message to the client. Before entering this state, the client receives a contract from the provider indicating the level of transaction support that will be supplied for the requested service (see Sect. 5.1).

From the *Initial* state, the client can choose to *accept* or *reject* the contract. If the client rejects the contract, then the interaction transitions to the *Failed* state. Otherwise the interaction moves into the *Active* state. From the *Active* state, either party can *cancel* the interaction, moving it to the *Failed* state. Other options from the *Active* state allow the client to request either *enquire*, *prepare*, or *commit* operations.

In the case that an *enquire* request is sent, the provider may reply that the request was successful (indicating that the provider can currently successfully complete the activity), which moves the interaction to the *EnquirySuccessful* state. If, on the other hand, the provider replies that the enquiry is unsuccessful (indicating that the provider cannot currently complete the required activity successfully), the interaction moves to the *EnquiryFailed* state. From either of these states, the client can initiate a new enquiry, though any enquiry callbacks remove the need as they transition the client between the *EnquiryFailed* and *EnquirySuccessful* states without requiring a new request. Similarly to behaviour in the *Active* state, either party can *cancel* the interaction from the *EnquiryFailed* and *EnquirySuccessful* states, or the client can request the *prepare* and *commit* operations.

When a *prepare* request is sent, the provider can either send a *cannotComplete* reply to indicate that the prepare has not occurred, or a *prepared* message

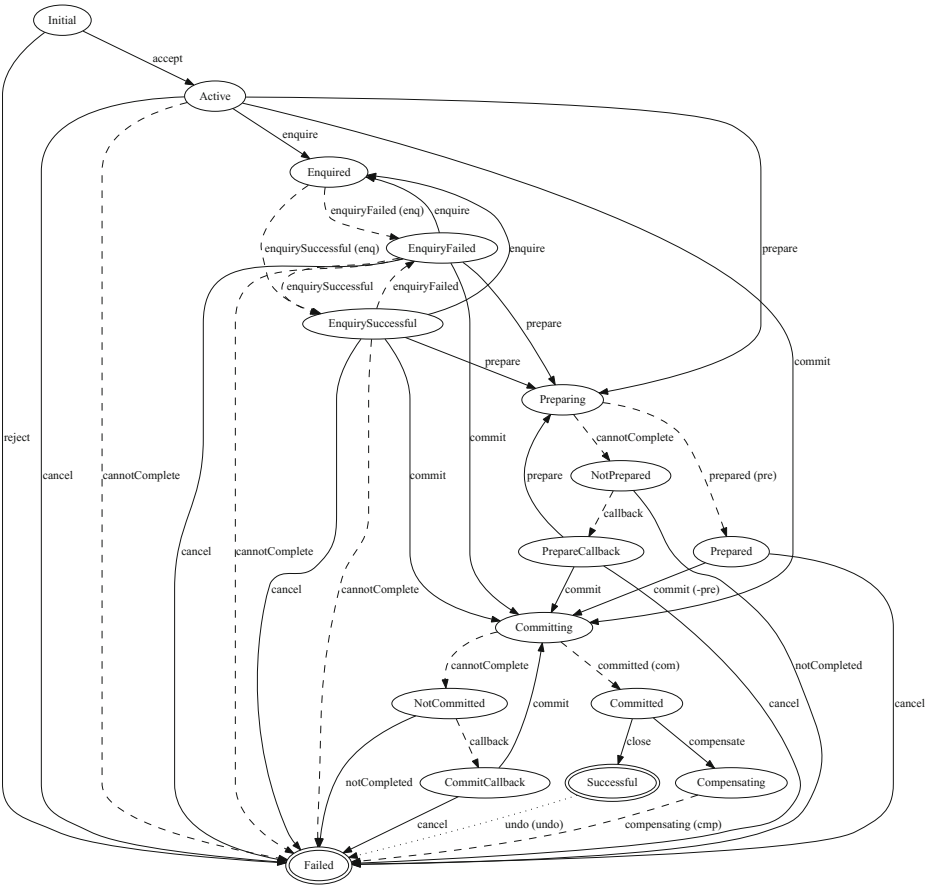


**Fig. 1.** State transition diagram for a client's processing of an activity. Solid lines represent messages sent by the client. Dashed lines represent messages sent by the service provider. Dotted lines indicate client-side only transitions. Values in parentheses represent the associated cost to the client.

to indicate that the prepare has been performed successfully. On receipt of a *cannotComplete* message, the client can either acknowledge the failure, moving to the *Failed* state, or, if applicable, wait for a call back from the provider and then send another *prepare* request. If the prepare is completed successfully, the client can choose to either *cancel* the activity, or request to *commit*.

After a request to *commit*, the provider can either perform the requested service successfully, or indicate that it cannot be completed at the current time. If the *commit* request was sent from the *Prepared* state then the second option should not be possible, as the *prepare* operation guarantees that a later call to *commit* will succeed. On receipt of a *committed* message, the interaction moves to the *Committed* state. With compensation support, the client can choose to

either *close* the interaction, moving it to the *Successful* state, or *compensate* the activity, moving to the *Failed* state. If the provider indicated that it could not complete the commit operation, the interaction moves to the *Failed* state unless the provider sends a callback to the client to indicate that the commit operation would now succeed.

Finally, from the *Successful* state, the client may choose to take some action to undo the operation performed by the provider, without the provider's knowledge. In this case, the provider would believe that the interaction completed successfully, while the client would see it as a failure. This allows the client to compensate the action performed by the provider even when the provider does not support compensation. For example, if the client is requesting an item from the provider that has already shipped to the client, the client could immediately dispose of the item to effectively have it appear as though the operation failed.

### 5.1 Provider Contracts

When a provider receives a request from a client, it must decide on the level of transactional support it wishes to provide to the client. The provider may be willing to support more than one possible level of transaction support, but at this stage will not know which level would be preferable for any particular client.

Once the provider has decided on the level of transaction support it is willing to provide, it must notify the client of the decision. This can be achieved by having the client first request the level of transactional support that the provider is willing to offer for a given service request [10]. The provider must then inform the client of any transactional interaction patterns it is willing to provide for that service request, by sending contract offers describing the level(s) of support the provider is willing to support. The client must then agree to one of these contracts before the service request can be handled in the required transactional manner. If the client does not agree with the initial patterns offered by the provider, it may wish to renegotiate with the provider until both sides have found a level of transactional support they are willing to accept [10]. Once the client has accepted one of the offers from the provider, the interaction continues following the pattern specified in the agreed-upon contract.

### 5.2 Combining Contracts

Once the client begins accepting agreements from various service providers, it must choose how to proceed. The client can achieve this by prioritising those yet-to-be-completed activities in its workflow, and determining whether the risk of performing the next stage of the first activity in the prioritised list is acceptable. If the risk is acceptable, then the client performs the next stage of the first activity.

When the client deems the risk of performing the next stage of the first activity to be too great, behaviour depends on the activity being performed. If the client thinks it likely that waiting will reduce the risk, the client may choose to wait. Otherwise, if there is an alternate activity that can replace the current

activity, the client may choose to cancel the current activity and replace it with the alternative. If the risky activity is optional for the success of the workflow, then the client may choose to cancel the activity. If, however, the activity is necessary and none of the previous options are available, the client must cancel its workflow.

In order to allow a client to calculate risk, it is assumed that the client has knowledge of a price for each action it performs. This could be prescribed by a provider, for example as part of the contract, by specifying a particular price for each of the five supported basic transaction operations (enquire, prepare, commit, compensate, callback) [10].

To allow the client to make decisions based on these costs, it is assumed the client has a maximum budget it is willing to spend to have its workflow complete successfully, and a maximum budget it does not want to exceed should the workflow fail. It is possible that a cost to the client could be multidimensional. For example, if a client orders a physical item from a provider, it will have to pay the provider as well as have storage space for the item. The model presented here assumes a single-dimensional cost, though this could be extended to more dimensions by specifying a cost for each action in each dimension.

A contract identifying the level of transaction support that a provider will offer to a client for a particular service call, and the cost for this support, can be defined as a tuple $(enq, ecb, eth, pre, pcb, com, ccb, cmp)$, where:

$enq \in \mathbb{R}$ is the cost of an enquiry operation.
$ecb \in \{0, 1\}$ specifies whether the provider may call back after an unsuccessful enquire operation if the enquiry would now be successful.
$eth \in \{0, 1\}$ specifies whether the provider may call back if a tentative hold on an item is revoked.
$pre \in \mathbb{R}$ is the cost of a successful prepare operation.
$pcb \in \{0, 1\}$ specifies whether the provider may call back after an unsuccessful prepare request if the prepare would now be successful.
$com \in \mathbb{R}$ is the cost of a successful commit operation.
$ccb \in \{0, 1\}$ specifies whether the provider may call back after an unsuccessful commit request if the commit would now be successful.
$cmp \in \mathbb{R}$ is the cost of a successful compensate operation.

To support the client's undoing of a successfully completed interaction without the provider's knowledge, it is also necessary for the client to specify a cost $undo \in \mathbb{R}$ to indicate the cost to compensate a completed interaction on the client's side. For each cost, a value of $\infty$ indicates that the associated operation is not supported by the interaction. It is also possible to use a contract to specify time out behaviour for an interaction. However such details are beyond the scope of this paper, in which it is assumed that time outs never occur.

Figure 1 indicates which transitions trigger a cost to the client. Using this information, a client can determine the minimal cost required for an interaction to complete in the *Successful* state, and the maximum cost required to move the interaction into the *Failed* state. Table 1 displays the *success* and *fail* costs for each state.

**Table 1.** Costs required for a client to guarantee the completion of an interaction. *success* defines the minimal cost required to transition to the *Successful* state, and *fail* defines the maximum cost necessary to transition to the *Failed* state.

| State | *success* | *fail* |
|---|---|---|
| Initial | $com$ | 0 |
| Active | $com$ | 0 |
| Enquired | $enq + com$ | $enq$ |
| EnquiryFailed | $com$ | 0 |
| EnquirySuccessful | $com$ | 0 |
| Preparing | $com$ | $\max\{0, pre\}$ |
| NotPrepared | $com$ | 0 |
| PrepareCallback | $com$ | 0 |
| Prepared | $-pre + com$ | 0 |
| Committing | $com$ | $\max\{com + \min\{undo, cmp\}, 0\}$ |
| NotCommitted | $com$ | 0 |
| CommitCallback | $com$ | 0 |
| Committed | 0 | $\min\{undo, cmp\}$ |
| Compensating | $\infty$ | $cmp$ |
| Successful | 0 | $undo$ |
| Failed | $\infty$ | 0 |

Client workflows typically combine multiple interactions with multiple providers. Using a system inspired by $\pi t$-calculus [12], we describe a workflow $W$ as:

$$\begin{array}{rcll} W \triangleq & done & & \text{(success)} \\ | & abort & & \text{(failure)} \\ | & W|W & & \text{(parallel)} \\ | & W;W & & \text{(sequence)} \\ | & W,W & & \text{(alternative)} \\ | & (p)(s,c,u) & & \text{(client interaction)} \end{array}$$

where *done* and *abort* are activities that respectively indicate the successful or unsuccessful completion of a workflow. The parallel operation $W|W$ executes two workflows in parallel, with no guarantee as to which will complete first. In contrast, the sequence operator $W;W$ guarantees that the workflow on the right will only succeed if the workflow on the left succeeds first. The alternative operator $W,W$ means that the workflow succeeds if either the workflow on the left succeeds and the one on the right fails, or, given the first workflow's failure, the second workflow succeeds (i.e. only one of the two ever succeeds).

The final operation is a client interaction, $(p)(s,c,u)$. $p$ is a unique identifier for a client's interaction with a provider; $s$ is the current state of the interaction, which can be any of the states in Fig. 1; $c$ is a contract as defined above; and $u$

$$
\begin{aligned}
success((p)(s,c,u)) &= success(s) & fail((p)(s,c,u)) &= fail(s)\\
success(done) &= 0 & fail(done) &= \infty\\
success(abort) &= \infty & fail(abort) &= 0\\
success(V|W) &= success(V) + success(W) & fail(V|W) &= fail(V) + fail(W)\\
success(V;W) &= success(V) + success(W) & fail(V;W) &= fail(V) + fail(W)\\
success(V,W) &= \min\{success(V) + fail(W), & fail(V,W) &= fail(V) + fail(W)\\
&\quad fail(V) + success(W)\}
\end{aligned}
$$

**Fig. 2.** The minimal success and maximal failure cost of a workflow.

is the undo cost to the client to move the interaction from the *Successful* state to the *Failed* state.

Given the above definition, and workflows $V, W, X$, the following structural congruences hold:

$$
\begin{aligned}
V|W &\equiv W|V & V,(W,X) &\equiv (V,W),X\\
V|(W|X) &\equiv (V|W)|X & V;(W;X) &\equiv (V;W);X\\
done|W &\equiv W & done;W &\equiv W\\
abort|abort &\equiv abort & abort;abort &\equiv abort\\
abort,W &\equiv W & W,abort &\equiv W
\end{aligned}
$$

Seven of these congruences are directly analogous to those in $\pi t-$calculus [12]. The remaining three specify that $\_,\_$ is associative and that *abort* is an identity to the operation.

Using this definition of workflows, it is possible to calculate the minimal success cost, *success*, and maximal failure cost, *fail* of a workflow as specified in Fig. 2.

The success or failure cost of an interaction with a particular provider is the success or failure cost of the interaction as defined in Table 1. *done* represents the successful completion of a workflow interaction, and thus has a success cost of 0 and a failure cost of $\infty$ (as it is not undoable). Similarly, *abort* represents a failure of the workflow, so its failure cost is 0 and its success cost is $\infty$. Both the success and failure costs of workflows performed in parallel $(V|W)$ or in sequence $(V;W)$ are simply the sums of associated costs of each component included in the workflow. Alternatives such as $(V,W)$ are successful whenever exactly one of the alternatives succeeds, making the minimum success cost the minimum cost required to have one of the alternatives fail and the other succeed. The failure cost of alternatives is the sum of the failure cost of each component, as both must fail for the alternatives to fail.

By combining the client's success and fail budgets with the costs calculated by the above *success* and *fail* functions, a client is able to determine whether its workflow can succeed. Each workflow begins with the client having not accepted any of the provider-offered contracts and, since the *fail* cost of an interaction from the *Initial* state is always 0, it is guaranteed that the client can initially

cancel the workflow without exceeding its failure budget. The client then chooses the next action to perform by ensuring that the action will not cause the cost of the workflow to exceed the client's success or failure budgets.

For a single client interaction, the set of possible next steps is specified by the transitions in Fig. 1 (with some transitions being removed if the contract does not support them. For example, if $pcb = 0$ then the transition from *NotPrepared* to *PrepareCallback* is removed). The next step progresses the workflow by changing the state of the interaction, while keeping the contract and *undo* cost constant. If the next step requires a provider-generated message, the client's action is to wait for that message to arrive. When a number of client interactions are combined in a workflow, the possible next steps are the union of the possible next steps for each interaction in the workflow.

The client begins by considering all of the possible actions that can be performed from its current state. Each of these actions has a possible cost (as displayed in Fig. 1) and moves one of the client interactions in the workflow to a new state. If the cost of a potential action, plus the *success* or *fail* cost of the resultant workflow, exceeds the success or failure budget (respectively), that action is removed from the set of possible actions. In this way, the client creates a set of possible actions it can perform, given the current state of its workflow, which will not exceed the client's budgets.

The aim of the client is firstly to complete the workflow successfully, by reducing it to *done*. If that becomes impossible then the aim shifts to completing the transaction by reducing it to *abort*. The client thus chooses an action, from its set of possible actions, to bring it closer to one of these outcomes. Once an action has been performed, the client uses structural congruences to simplify the workflow as much as possible. The client also uses the following reduction rules:

- Any client interaction that is in the *Failed* state is replaced with *abort*.
- Any workflow that has no alternatives, and all client interactions in the *Successful* state, is replaced with *done*.

The first rule indicates that a failed interaction is equivalent to *abort*, since they are both failures. The second rule ensures that any sequence or set of workflows running in parallel succeeds if each client interaction in the workflow succeeds. Further, since the structural congruence rules only remove an alternative if it is *abort*, the restriction on alternatives ensures that, for each set of alternatives, exactly one alternative succeeds.

By following this strategy, the client is assured that it will only ever perform an action that will not exceed its budgets. However, the client can still optimise the strategy for its own purposes. Once the client has determined the set of possible actions for its current state, the client can choose which action to perform based on other criteria. For example, the client may choose the next action that minimises either the *success* or *fail* cost of the workflow. Regardless of how the action is chosen, the client performs the action, which moves the workflow to a new state. The client then repeats the process for the new workflow state, until the workflow is complete.

Note that, while this work borrows from the algebraic laws of $\pi t-$calculus [12], this system augments these laws with client reasoning about the individual interactions included in its combined workflow. Thus the presented system is not a process calculus, but instead a tool to allow client reasoning. In particular, a client only requires a model of its own workflow and a knowledge of the required transactional behaviour of the providers it is using to allow its reasoning, whereas most process calculi require a complete description of not only all service providers that may be used by the client, but also of any other clients that may utilise those provider's services while the first client's workflow is still processing [13]. By thus reducing the complexity of the model, it is possible for a client to guarantee its correct behaviour, rather than having the system grow to a size that such analysis become intractable (as in [14]).

## 6 Simulation

Web Services transactions have been shown to be very different from traditional transactions. Multiple parties work together to achieve a client's aims, but still wish to remain autonomous. Further, the different service providers may have different levels of transactional support, making verification and testing of different transaction schemes very complex. While theoretical analysis is essential to ensure that transaction schemes work correctly, these models (e.g. [14,15]) do not easily allow comparison between different transaction techniques. In particular, these models make it difficult to determine which transaction schemes are best suited for certain conditions, applications, or environments.

Instead, simulation can be used to provide an indication of practical results. In simulation, some or all of a system is abstracted so that only the features important to the current investigation are tested. When simulating Web Services transactions, details such as the networking topology, the timing of events, and the actual services being used can be abstracted. This can allow intricate comparison of various transaction strategies, by allowing the parameters of interest to be studied while ensuring that all other factors are kept constant.

Most available Web Services simulation environments replace a Web Service with a simple, usually local, program that sends and responds to messages in a way that is appropriate to the service being simulated [3]. However, when examining Web Services transactions, further abstraction can occur [3]. Messages are not required to be sent in the exact same format as with real services; it is only the transaction interaction patterns that need to be simulated. A simulator, based on the model introduced in Sect. 6, that models transaction flow rather than message flow, has been described previously [3,10].

The simulator accepts a description of the scenario to process, which includes details of the providers and clients to simulate. Provider information describes the number of resources available from each provider, cost information for accessing those resources, and the availability of abstract providers that provide resilience for the simulation. Client definitions contain a description of a workflow, which consists of a sequence of activities to be performed, and specifies whether successful completion of each activity is required, or if the workflow is successful when

at least one of the included activities succeeds. An activity can either be a workflow or a service call. Finally, the scenario description contains timing information to specify the length of time that messages take to be sent between the different participants in the system being modelled.

The simulator also accepts various parameters to specify the level of transaction support offered by the various service providers, and the risk-taking behaviour of the clients. These parameters can be changed independently of the scenario description, and any changes to results can be directly attributed to these changed values. The level of transaction support offered by providers is specified based on the five operations described in Sect. 5 and, when combined with abstract services, this allows providers to utilise all of the reductions to the standard ACID properties. Client risk-taking behaviour is specified using budgets, as described in Sect. 5.2.

The simulator models the passing of messages between the various participants in the simulated scenario. Each provider tracks the state of its resources to ensure the provider can support any transactional contract it offers to a client. Clients similarly monitor their interactions with providers to track the progress through their workflows, and to determine the next action the client should take. By tracking all messages sent through the simulator it is possible to extract details such as whether a particular client's workflow succeeded or failed, and the cost for that outcome, or the number of a provider's resources that were utilised, and the amount it was paid.

Thus, by defining appropriate scenarios, it is possible to use the simulator to measure the way different transactional support from providers, or different risk-taking behaviour of clients, affects the outcome of the simulation. The following section will describe a scenario, based on the preceding motivating example, to demonstrate the effects of using a dynamic transaction scheme rather than a scheme in which the transaction support offered by each provider remains fixed.

## 7 Results

The Web Services transactions simulator described in Sect. 6 was used to validate the dynamic transaction model introduced in Sect. 5. The validation experiment was based on the motivating example described in Sect. 2.

According to the scenario, the display system offered either 100 or 1000 time units for clients to book. A total of 500 clients, each requiring between 1 and 5 time units, were included in the simulation. Each client had a success budget that allowed for successful completion of both the display booking and the other required services. 100 of the clients were given a fail budget of 0, meaning that they would only perform an action that required payment if success was guaranteed. The remaining clients had a budget that allowed either the booking of the display to fail while the other actions succeeded, or for the other actions to succeed and the booking of the display to fail. The other services were modelled as a service that was successful 80 % of the time, with no special transaction support, giving a sufficient error rate to allow comparison of different transactional support levels.

Three different levels of transactional support offered by the display booking service were simulated. The first level provided semantic atomicity, allowing clients to book a time unit and then later cancel that booking without charge. The second level offered tentative holds. When granted a tentative hold, 50 % of the clients immediately converted that tentative hold into a booking, and then attempted the other services. The remaining 50 % retained the hold and attempted the other services, only confirming the booking if the other services completed successfully. The final level of transaction support was a variable scheme which offered semantic atomicity until 50 % of the time units had been booked, and then offered tentative holds instead. This final level was only possible with dynamic transaction support.

Assuming the data service is *data*, the rendering service *render*, and the visualisation service *display*, and each has a cost per service call of 1, the system can be modelled as follows. Each client has a success budget of 4, a fail budget of 0 or 2 (as specified above), and the following workflow:

$$(data)(\text{Initial}, c_1, \infty); ((render)(\text{Initial}, c_2, \infty)|(display)(\text{Initial}, c_3, \infty))$$

where $c_1$, the contract for the data service, is $(\infty, 0, 0, \infty, 0, 1, 0, \infty)$, $c_2$, the contract for the render service is similarly $(\infty, 0, 0, \infty, 0, 1, 0, \infty)$, and $c_3$, the contract for the display service, is $(\infty, 0, 0, 0, 0, 2, 0, \infty)$ if semantic atomicity is offered, or $(0, 0, 0, \infty, 0, 2, 0, \infty)$ if tentative hold is offered.

The results for these simulations can be seen in Table 2. The first column indicates the level of transaction support that was offered by the display-booking service. The "Offered" and "Booked" columns give, respectively: the number of time units that the provider offered to clients; and the number of time units that were actually booked. The "Succeeded", "Failed", and "Penalised" columns indicate, respectively: the number of client workflows that completed successfully; the number of client workflows that failed with neither the display booking nor the other services requiring payment; and the number of client workflows that failed with either the display booking service or the other services completing successfully (and thus requiring payment). The "Reserved Time" column indicates the total length of time during which the provider held resources for a client that later cancelled its request (i.e. the time resources were withheld from other clients, ultimately unnecessarily).

**Table 2.** Results of simulation based on motivating example.

| Transactionality | Offered | Booked | Succeeded | Failed | Penalised | Reserved Time |
|---|---|---|---|---|---|---|
| Semantic atomicity | 100 | 100 | 42 | 458 | 0 | 119 |
| Tentative hold | 100 | 100 | 48 | 420 | 32 | 0 |
| Variable | 100 | 100 | 44 | 444 | 12 | 76 |
| Semantic atomicity | 1000 | 963 | 393 | 107 | 0 | 1350 |
| Tentative hold | 1000 | 805 | 330 | 130 | 40 | 0 |
| Variable | 1000 | 914 | 367 | 112 | 21 | 588 |

When only 100 time units were offered, each time unit was booked by a client. This is unsurprising, as the large number of clients compared to the available resources meant that each time unit was highly contended. In contrast, when 1000 time units were offered, more time units were booked when semantic atomicity was offered. This is because the clients with a 0 fail budget would only attempt their workflow if such a level of transactional support was offered. However, as can be seen from the "Reserved Time" column, semantic atomicity places a higher burden on the service provider, as the provider must ensure that a completed action can be undone until the client determines whether or not it will compensate the booking, whereas tentative holds have no such restrictions. Using the variable scheme, the provider obtained results that were a compromise between only offering tentative holds and only offering semantic atomicity.

From the point of view of clients, the benefits of better transactional support can be seen in the "Penalised" column of Table 2. Tentative holds give no future guarantee about the state of resources, so a client may commit one part of its workflow, but have the tentative hold expire before completing the booking. In contrast, when semantic atomicity is offered, the client can perform the action before attempting the rest of its workflow. If any part of the workflow later fails, the performed action can be compensated to allow completion without penalty. Thus, no clients failed with a penalty when the display provider offered semantic atomicity, though, as can be seen by the "Reserved Time" column, transactions did take longer to be processed. The use of the variable scheme reduced the number of clients that were forced to pay for an incomplete workflow compared to when only tentative holds were used, and the average transaction length was shorter than when only semantic atomicity was offered.

## 8 Conclusions

Service-oriented environments have different transactional requirements to traditional transaction systems. Reductions to the ACID properties are often used to help retain autonomy for service providers while still offering an acceptable level of service for clients. In some cases, it is desirable to dynamically alter the level of transaction support offered for a particular service as the provider's environment changes, but the level of transaction support is typically specified along with the definition of the service.

This paper describes a technique whereby providers offer transactional contracts to clients on a per-service-call basis. Once a provider and client agree upon a level of transaction support, the client can include that service call in its workflow. A formal model, inspired by $\pi t$-calculus, was presented to allow automated reasoning about arbitrarily complex client workflows. This model allows a client to ensure its workflow has an acceptable outcome.

A Web Services transactions simulator was developed to allow the investigation of different transactional strategies. By varying the offered level of transaction support, the provider was able to better balance the strength of the transactional guarantees it supported and the number of clients that completed successfully, offering benefits to both clients and service providers.

## References


1. Garcia-Molina, H.: Using semantic knowledge for transaction processing in a distributed database. ACM Trans. Database Syst. **8**, 186–213 (1983)
2. Fauvet, M.-C., Duarte, H., Dumas, M., Benatallah, B.: Handling transactional properties in web service composition. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.-Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 273–289. Springer, Heidelberg (2005)
3. Paul, D., Henskens, F. A., Hannaford, M.: Simulating web services transactions. In: Special Session on Web Services Principles and Applications (WSPA 2011) at the 7th International Conference on Web Information Systems and Technologies (WEBIST-2011), Noordwijkerhout, The Netherlands (2011)
4. Chakraborty, D., Joshi, A.: Dynamic service composition: State-of-the-art and research directions. Technical report, University of Maryland (2001)
5. Schäfer, M., Dolog, P., Nejdl, W.: Engineering compensations in web service environment. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 32–46. Springer, Heidelberg (2007)
6. Milanovic, N., Stantchev, V., Richling, J., Malek, M.: Towards adaptive and composable services. In: International Conference on Internet, Processing, Systems, Interdisciplinaries (IPSI2003) (2003)
7. Milanovic, N.: Contract-based web service composition. Ph.D. thesis, Humboldt-Universität zu Berlin (2006)
8. Mikalsen, T., Tai, S., Rouvellou, I.: Transactional attitudes: reliable composition of autonomous web services. In: Workshop on Dependable Middleware-based Systems (WDMS'02) at the Dependable Systems and Network Conference (DSN'02), Bethesda, MD, USA (2002)
9. Paul, D., Henskens, F.A., Hannaford, M.: Per-request contracts for web services transactions. In: 6th International Conference on Web Information Systems and Technologies (WEBIST-2010), Valencia, Spain, INSTICC (2010)
10. Paul, D.: Deliberate cooperation in service-oriented environments: dynamic transactional workflows for web services. Ph.D. thesis, University of Newcastle (2012). http://hdl.handle.net/1959.13/932268
11. Younas, M., Eagelstone, B., Holton, R.: A formal treatment of a SACReD protocol for multidatabase web transactions. In: Ibrahim, M., Küng, J., Revell, N. (eds.) DEXA 2000. LNCS, vol. 1873, pp. 899–908. Springer, Heidelberg (2000)
12. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
13. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
14. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A logical verification methodology for service-oriented computing. Technical report, Universita' degli Studi di Firenze (2009)
15. Casado, R., Tuya, J., Younas, M.: A framework to test advanced web services transactions. In: IEEE 4th International Conference on Software Testing, Verification and Validation (ICST'11), Berlin, Germany (2011)