ICT-2011.8
GET Service Project
2012-318275

Deliverable D7.2

Design of a Reconfigurable Transportation Orchestration Engine

24 December 2014
Public Document

The GET Service project (http://www.getservice-project.eu) has received funding from the European Commission under the 7th Framework Programme (FP7) for Research and Technological Development under grant agreement n°2012-318275.

Project acronym:	GET Service
Project full title:	Service Platform for Green European Transportation

Work package:	7
Document number:	D7.2
Document title:	Design of a Reconfigurable Transportation Orchestration Engine
Version:	1

Delivery date:	31 December 2014 (M27)
Actual publication date:	31 December 2014
Dissemination level:	Public
Nature:	Report

Editor(s) / lead beneficiary:	Remco Dijkman
Authors(s):	Shaya Pourmirza Remco Dijkman Paul Grefen
Reviewer(s):	Andreas Raptopoulos Claudio Di Ciccio

Contents

Executive summary

During the execution of a service collaboration, a party may drop out for technical reasons or business reasons. In that case, that party must be replaced in the collaboration, at run-time, by a new party. Ideally, the new party can pick up where the old party left. Currently, algorithms exist that may help with the selection and adaptation of the new party to incorporate it in the collaboration. Also, algorithms exist that can help to pick up a business process where it was left off. However, to the best of our knowledge, no algorithms currently exist that can help a new party in a collaboration to pick up where the old party left off. This deliverable fills that gap, by providing an overview of the components and operations that are necessary to enable a party in a collaboration to be replaced by another party at run-time. In addition the deliverable presents two strategies, and the corresponding algorithms, that realize the architecture. As a proof-of-concept, a tool was developed that implements both strategies.

This deliverable is an extended version of the paper that appeared as:
S.Pourmirza, R.M. Dijkman, and P.W.P.J. Grefen. Switching Parties in a Collaboration at Run-time. Proceedings of EDOC 2014, Ulm, Germany, 2014.

1 Introduction

This deliverable presents a technique for updating a running process in a service orchestration engine to a changed process. This section provides the background to this deliverable, by presenting the goal of the project as a whole, the goal of the work package of which the deliverable is a part, and the goal of the deliverable itself. Finally, it presents the structure of the remainder of the deliverable.

1.1 Project Goal

The GET Service project develops a platform that provides transportation planners and drivers with the means to plan and execute transportation routes more efficiently and to respond quickly to unexpected events during transportation. To this end, it connects to existing transportation management systems (TMS) and improves on their performance by enabling sharing of selected information between transportation partners, logistics service providers and authorities. In particular, the GET Service platform consists of components that: (i) enable aggregation of information from the raw data that is shared between partners and transportation information providers; (ii) facilitate planning and re-planning of transportation based on that real-time information; and (iii) facilitate real-time monitoring and control of transportation, as it is being carried out by own resources and partner resources. By providing this functionality, the GET Service platform aims to reduce the number of empty miles that are driven, improve the modal split, and reduce transportation times and slack, as well as response times to unexpected events during transportation. Thus it reduces CO_2 emissions and improves efficiency.

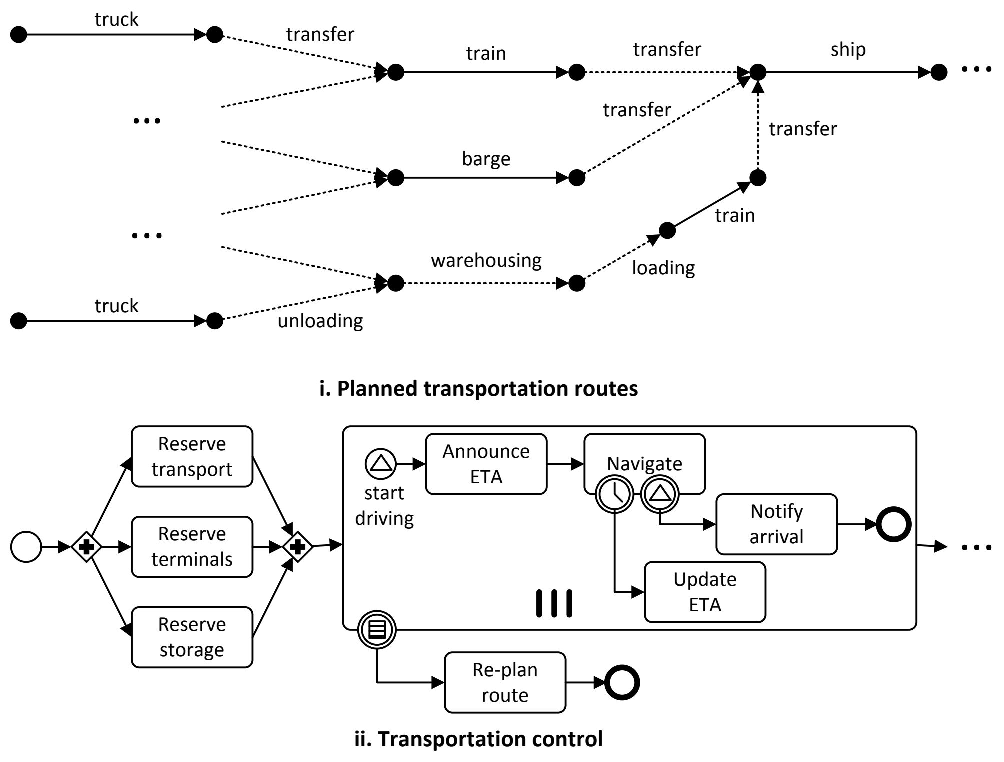


Figure 1. Example planning and control output

1.2 Work package goal

The goal of Work Package 7 (WP7) is to develop support for monitoring and executing a detailed transportation plan, also called a ‘transportation process’. Figure 1 illustrates the relation between a transportation plan and a transportation process with an example. In the example, a planner of a transportation company must plan the bulk transportation of several electronics components on behalf of one of the company’s clients, from different locations in Europe to the far east. The planner enters the pick-up locations of the components into the system, as well as the times at which the components are estimated to be available for pick-up, the time at which the components should be in the factory in the far east and several other criteria for planning, such as the desire to minimize CO_2 emission. The planner does this through the graphical user interface (GUI) that is provided by the transportation planning service. The transportation planning service computes an optimal route, which may look like the one shown in Figure 1.i. The planner is informed of this option through the GUI. Alternatively, the planner may be provided with multiple alternatives from which he can select one, or the planner can manually adapt the planned route via the graphical user interface. After the planner selects a route, a service composition is created that will control the route. This composition is created based on the planned route and individual control services that are associated with the different segments of the route. This composition may look like the one shown in Figure 1.ii.

1.3 Deliverable goal

During the execution of a transportation process, one of the parties in the collaboration may drop out, for example, due to re-planning. For example, it is quite common that a truck is replaced by another truck or ship, due to changes in the master planning of the transportation company. In such a scenario, a web service that enables tracking and tracing of the truck and its cargo could be replaced by the web service of another truck. This could be a complex operation, depending on the information about the state that must be kept and transferred from one web service to its replacement. Such information could include administrative information about the cargo, and tasks that must be performed, such as inspections, picking up a container, loading the cargo into the container, preparing customs forms for the container, and so on.

Consequently, technology is required to support switching a web service in a collaboration with another web service at run-time and transferring the state from the original web service to the new web service. Currently, technology exists that facilitates selecting a web service to incorporate into a collaboration at run-time [10, 11, 12]. In addition, technology exists that facilitates making changes to running instances of a business process that organizes tasks that occur within a single organization [18, 19]. However, to the best of our knowledge no support exists for transferring the state of a running instance between parties in a ccollaboration.

Therefore, the goal of this deliverable is to present technology that enables a party in a collaboration to be switched with another party and its state to be transferred to that new party. In developing this technology, this deliverable contributes novel techniques and algorithms to the area of service-oriented computing. Prototype tool support is developed to demonstrate the feasibility of the technology.

1.4 Deliverable structure

The remainder of this deliverable is organized as follows. Section 2 provides an abstract example of a collaboration of web services and what it means to replace one web service in that collaboration by another web service at run-time. Section 3 presents the concepts in terms of which we develop our technology. Section 4 describes the actual technology to switch parties in a collaboration at run-time. Section 5 defines the algorithms that implement part of that technology. Section 6 presents prototype tool support that we developed as a demonstrator of the feasibility of the technology. Section 7 discusses the application of the technology in the context of the GET Service platform. Section 8 describes related work and section 9 the conclusions.

2 Running Example

This section presents a running example of switching parties in a collaboration at run-time. This example will be used in the remainder of the deliverable to explain the concepts that are introduced.

A highly abstract representation of a collaboration is shown in Figure 2. The figure shows a collaboration between three parties, in which a party is represented by a rectangle with a thick border. Each party has an internal behaviour (represented by a Petri net). Some of the elements of this behaviour occur at the border of a party. These elements represent interactions with other parties. The places and arcs that link elements of different parties represent communication relations between the parties.

Figure 3 shows the detailed behaviour of the middle party in the collaboration. This behaviour is more detailed than the behaviour of that party that is displayed in the collaboration, because this behaviour is actually executed by the party, while the choreography focuses on the (emergent) externally observable behaviour of the parties only (i.e. the behaviour that the parties exhibit at their interfaces). Clearly the internal behaviour of the parties must be consistent with their externally observable behaviour. The relations between these types of behaviour and techniques to derive one from the other have been studied in previous work [2].

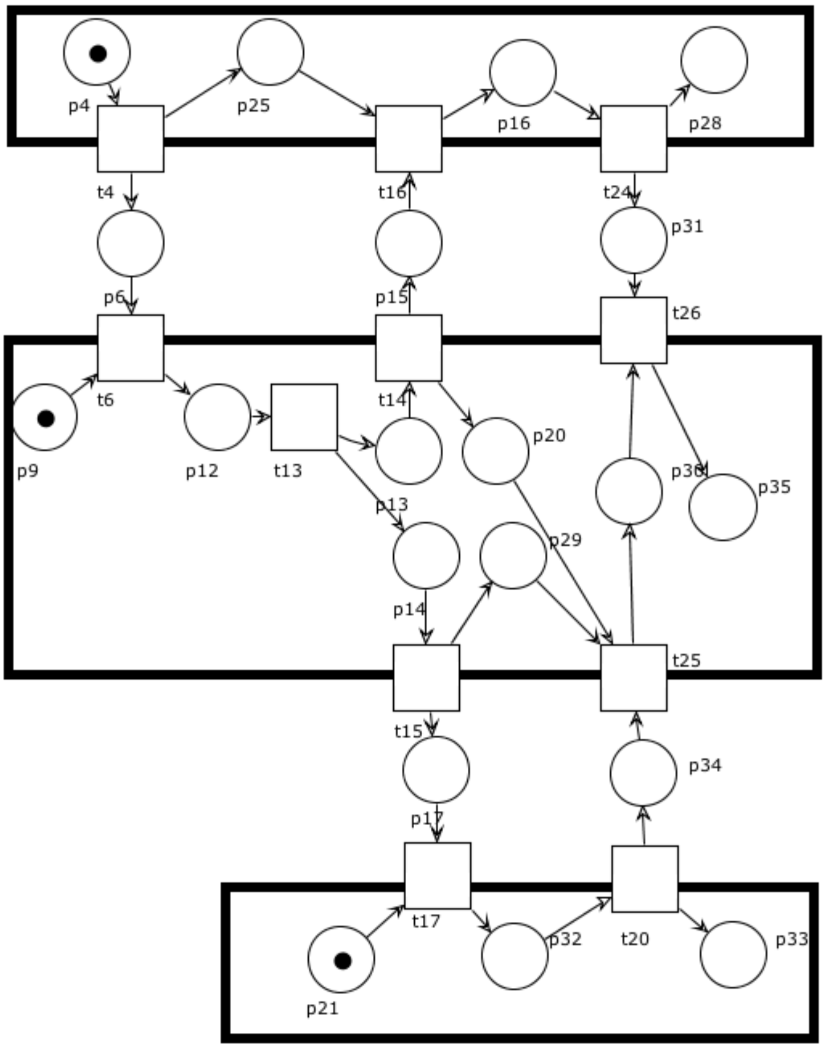


Figure 2. Choreography model of a collaboration

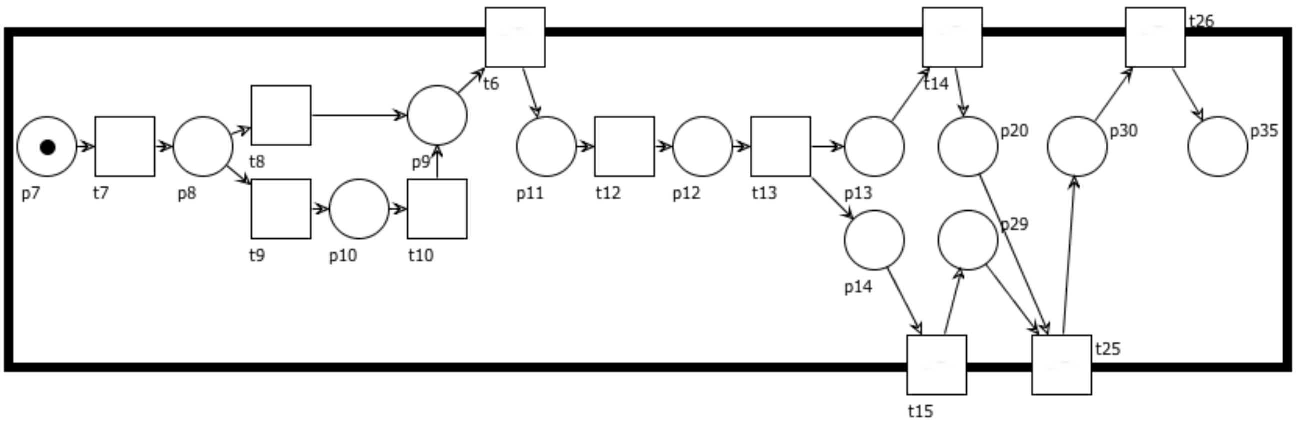


Figure 3. Orchestration model of party 2 in the collaboration

In the example, we replace the party presented in Figure 3 with the party from Figure 4. Please note here that the internal behaviours of these two parties differ, which poses a challenge when performing the switch.

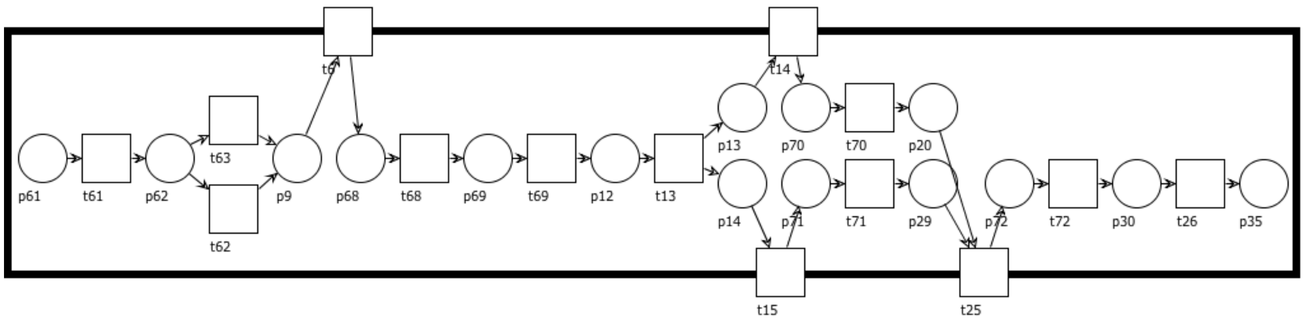


Figure 4. Orchestration model of the replacement of party 2 in the collaboration

3 Inter-Organizational Collaboration

This section defines the elements that constitute an inter-organizational collaboration as well as their behaviour in terms of an orchestration or choreography.

3.1 Elements

An inter-organizational collaboration is defined as a collaboration of multiple *Parties*. These parties perform autonomous *Activities*, but also communicate with each other. These communications are performed over *Communication Relations*. In a Service Oriented Architecture, communication relations mainly provide *Asynchronous* communication means, by providing a channel for passing *Messages* between parties.

The combination of parties performing activities autonomously and communicating with other parties, leads into two different viewpoints on a collaboration [3]. On one hand, each party has an *Orchestration* to manage its own internal activities, and on the other hand, the collaboration has a *Choreography* to manage the communication between the parties. Consequently, an orchestration is a more detailed viewpoint on the behaviour of a single party, depicting the complete flow of activities that should be performed by that party, while the choreography is a more abstract viewpoint on the behaviour of all parties, depicting only the message exchanges between the parties.

Consequently, we define a collaboration as a set of parties PRT, where each party $prt \in PRT$ consists of a set of activities that are performed by that party. The sets of activities of the parties are mutually disjoint ($prt_1, prt_2 \in PRT$: $prt_1 \cap prt_2 = \emptyset$ and $A = \bigcup PRT$ is the set of all activities). For example, Figure 2 shows a collaboration PRT with three parties. The parties' activities are represented by squares (as will be explained in the next subsection). Consequently, we can define $PRT = \{prt_1, prt_2, prt_3\}$ where $prt_1 = \{t_4, t_{16}, t_{24}\}$.

The communication relation CR of a collaboration, relates activities from different parties that involve sending and receiving messages, such that $CR : A \to A$ is an injective function, where for $(a_s, a_r) \in CR$, it holds that there exist $prt_s, prt_r \in PRT$, for which $prt_s \neq prt_r$, such that $a_s \in prt_s$ and $a_r \in prt_r$. For $(a_s, a_r) \in CR$, we call a_s the send activity and a_r the receive activity. Each activity can only appear once as a sent or receive activity: if $(a_s, a_r) \in CR$, there does not exist $(x, a_s) \in CR$, or $(a_r, x) \in CR$, or $(a_s, x) \in CR$ such that $x \neq a_r$ or $(x, a_r) \in CR$ such that $x \neq a_s$. For example, Figure 2 contains, among others, the communication relations $\{(t_4, t_6),(t_{14}, t_{16})\}$.

Based on the communication relation, the activities in a collaboration can be classified as *Internal Activities* and *Border Activities*. The set of border activities BA is the set of activities that represent either sending a message $SBA=\{a|(a,x)\in CR\}$ or receiving a message $RBA=\{a|(x,a)\in CR\}$, such that $BA=SBA\cup RBA$. The set of internal activities $IA=A-BA$ is the set of activities that is performed by a single party without communication.

Two border activities communicate over a communication relation via *Messages*. If a party sends a message via a border activity, this activity will be categorized as *Send Border Activity*, and the message will be stored in the party as *Sent Messages*. Likewise, if a party receives a message via a border activity, this activity will be categorized as *Receive Border Activity* and the message will be stored in the party as *Received Messages*.

3.2 Behaviour

The elements of an inter-organizational collaboration engage in a behaviour together. The behaviour of individual parties is also called an *orchestration*. The emergent behaviour that the parties display when they collaborate is also called a *choreography*. We use Petri nets to represent orchestrations and choreographies, acknowledging that - in practice - both are usually described using the Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL) specifications. We use Petri nets for conceptual purposes, enabling us to define algorithms for run-time adaptation of behaviour more easily. While this means that the algorithms that we define can be used in BPMN and BPEL specifications as well (also considering that mappings of BPMN and BPEL to Petri nets exist [4, 5]), future work is still necessary to make the translation and to investigate to which extent the algorithms can be used.

A *Marked Labelled Petri net* is a tuple (P,T,F,ℓ,M_0) such that:

- P is a finite set of places;
- T is a finite set of transitions, where $P \cap T = \emptyset$;
- $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relationship* between places and transitions;
- ℓ: $T \rightarrow A \cup \{\tau\}$ is the labeling function, which we use to relate transitions to the activities in a choreography or orchestration as defined in the previous section. The set of labels includes the label τ, which represents the occurrence of a transition that does not represent an activity.
- $M_0 : P \rightarrow \mathbb{N}$ is the initial marking of the net that represents its initial *state* in the collaboration.

Typically, and also in Figure 2, places are represented by circles, transitions are represented by squares and flows are represented by arrows. In Figure 2 we keep the labels of the transitions abstract (e.g.: $\{t_4, t_{16}, t_{24}\}$. The marking is indicated by black dots that represent the number of 'tokens' on a place.

We define the helper functions $\bullet x = \{n: (n,x) \in F\}$ that returns the places and transitions from which x can be reached, and $x \bullet = \{n: (x,n) \in F\}$ that returns the places and transitions that can be reached from x.

The behaviour that is represented by a Petri net can be defined as follows. A transition $t \in T$, and consequently the activity $\ell(t)$ represented by that transition, can occur in a marking M if $\forall p \in \bullet t$ it holds that $M(p)>0$. When the transition occurs, this leads to a new marking M', such that for each $p \in P$: $M'(p)=M(p)-|\{p|(p,t) \in F\}|+|\{p|(t,p) \in F\}|$.

Each party in an inter-organizational collaboration contains one static aspect (that does not change at run-time) and two dynamic aspects (that change at run-time). The static aspect is the orchestration that describes the party's internal behaviour. It is represented by the Petri net that consists solely of transitions that are marked with activities from the party, i.e.: the orchestration of a party *prt* is described by a Petri-net $PN(P,T,F,\ell,M)$, such that $\ell \subseteq T \rightarrow prt \cup \{\tau\}$. The dynamic aspects are *Marking History* and *Message History*. The marking history (*Mh*), is a sequence containing ordered pairs of a transition t and a marking M that is the result of executing t in the marking that precedes it in the sequence. The first element in the marking history is the initial marking M_0 that is not preceded by any transition, which we denote as $\perp$. Without loss of generality, we denote subsequent occurrences of the same transition t (when it is part of a loop) as t_1, t_2, ... Consequently, the marking history could, for example, look like $<(\perp, M_0), (t_1, M_1), (u_1, M_2), (u_2, M_3),\ldots>$. Assuming that the set *MSG* represents the set of all possible messages, the message history ($Msh \subseteq prt \times MSG$) of a party *prt* keeps track of the messages that were sent or received by that party, for every occurrence a_i of a border activity a (i.e.: for which there exists a (t_i, M) in the marking history, such that $\ell(t_i)=a$). Distinguishing between send and receive border activities, we can also distinguish between the sent (MSG_{si}) and received (MSG_{ri}) messages of a party.

Jointly, the parties engage in an emergent behaviour that is also called a choreography. We describe this behaviour as the behaviour in which the individual parties are connected via their communication relations. A communication can be represented in a Petri net as a send transition (corresponding to the send border activity of a party) and a receive transition (corresponding to the receive border activity of another party) as illustrated in Figure 5. In addition, in a choreography, the internal activities can be removed, because we are only interested in the interactions between the parties. Consequently, the choreography of a set of parties *PRT* can be represented by a Petri net that can be formed as the union of (the elements of) their orchestrations and:

- including $p \in P$, $(t_s, p) \in F$, and $(p, t_r) \in F$ for each $(a_s, a_r) \in CR$, such that $\ell(t_s)=a_s$ and $\ell(t_r)=a_r$; and
- replacing each $(t, ia) \in \ell$ for which $ia \in IA$ by $(t, \tau) \in \ell$.

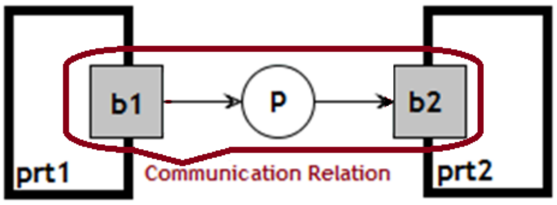


Figure 5: Behaviour of a Communication Relation

4 Switching Parties

This section presents two solutions to deal with the runtime party changes in a collaboration. First, it presents the assumptions under which the solutions work. Second, it presents the main flow of both solutions. Third, it presents two strategies that can be used to implement the main flow. Finally, it explains the switch with an example. Detailed algorithms are presented for these strategies in the next section.

4.1 Assumptions

First, we assume that the behaviour of a collaboration, as it is described in a choreography, is the same before and after the change. Consequently, our solution only functions in a situation in which all parties involved agree beforehand on the choreography that they will perform. There are many practical examples, such as Rosettanet[1], in which parties standardize their behaviour beforehand and, consequently, in which our algorithms can be applied. In addition, in related work in the area of service adapters [6], situations are also considered in which parties can interact even if their behaviour is not fully compatible.

Second, we assume a reliable communication. While this is not entirely realistic, most Internet communication is relatively reliable nowadays. Still, in future work, we will investigate ways to relax this assumption.

Third, and most importantly, we assume that it is possible to have a central party, which we will call the *Global Controller*, that can observe all communication and be the intermediary between all parties involved when one party is replaced by another. This assumption makes our solutions most suitable for collaborations between services within the same organization, because the level of control required by a global controller is unlikely to exist between organizations.

4.2 Main Flow

Figure 6 shows the main flow operations that is executed when a party in a collaboration is replaced by another party. It shows that the scenario begins with *removing* an existing party. In this scenario, the old party announces its removal to the global controller, but we can also support scenarios in which the global controller temporarily polls all parties, to determine who left the collaboration. The global controller is responsible for selecting and initiating a new party to replace the old party.

Once the global controller has added the new party to the collaboration, two important steps must be taken:

1. the new party must be taken to a state that is as close as possible to the state in which the old party left the collaboration; and
2. the other parties in the collaboration must be informed of the new state and possibly adapt their own states accordingly.

Key to determining the state of the new party is determining the *migratable area* in the choreography, i.e. the set of transitions that needs to be re-executed because of the change. Assuming that the old party has already left the collaboration and cannot pass on the information about the messages that it sent or received, the migratable area is determined as follows. First, the global controller retrieves all the communication relations that include the old party to find the parties who had a communication with an old party. Second, it retrieves the messages that were exchanged with the old party. Third, for each of the messages that were sent by the old party, the global controller checks with the new party, whether it accepts that message as a message that it could have sent itself. A party can decide itself whether it approves old messages or not. To this end, we assume the existence of an ‘approve’ function that returns true when a message can be approved (a simple implementation of this function would always return false). Consequently, let *old* be the party that is leaving the collaboration, then the migratable area contains the transitions: $t \in T$

[1] http://www.rosettanet.org

for which there exist an *a*∈*old*, such that ℓ(*t*)=*a* and not *approve*(*Msh*(*a*)). It also contains the receive transitions that are the counterparts of the send transitions T_s that are not approved: $t_r \in T_r$ for which there exist $t_s \in T_s$ and (ℓ(t_s), ℓ(t_t)) ∈ *CR*.

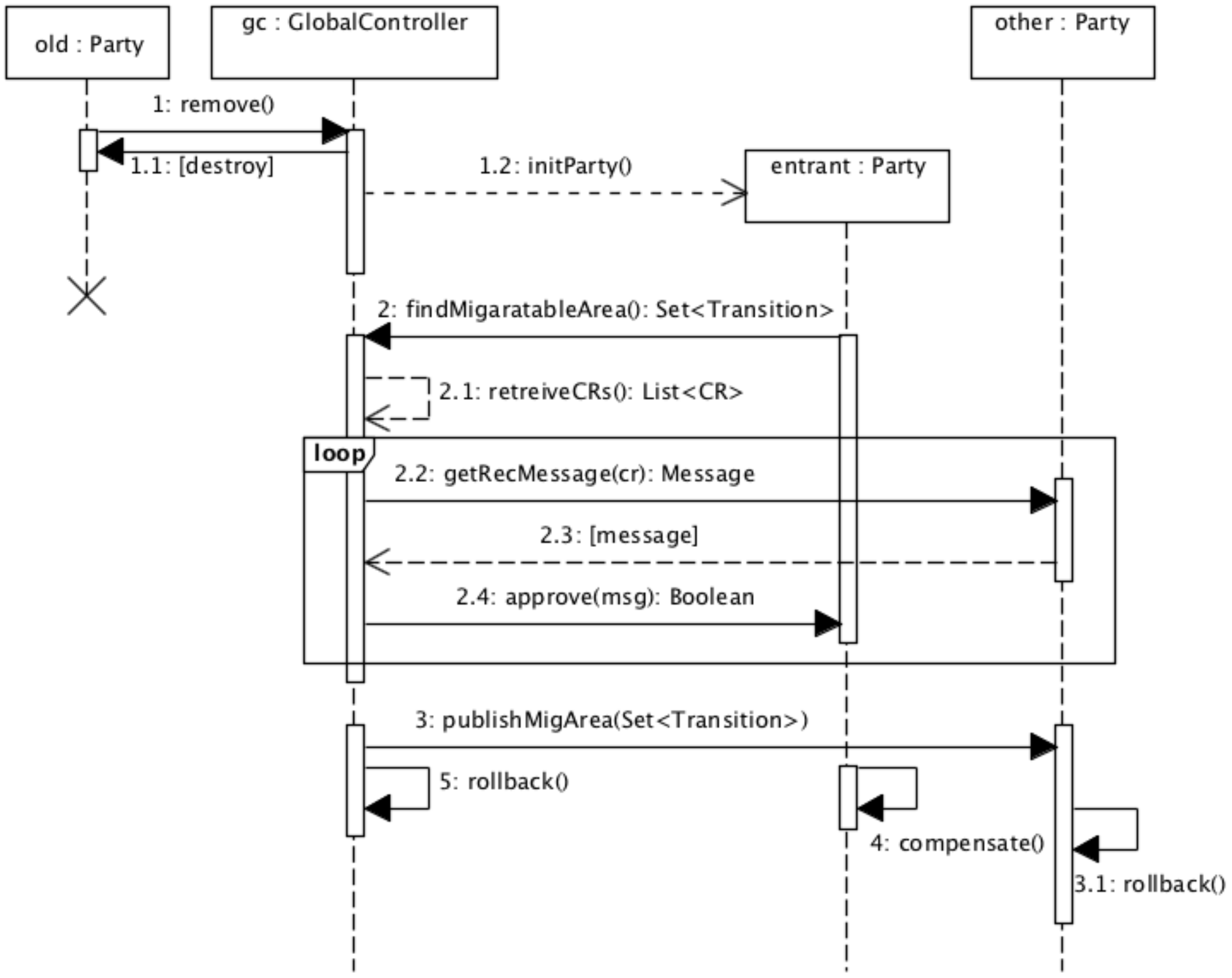


Figure 6. Main Flow of Switching Parties

Based on the migratable area, the entrant party can determine its *migratable state*. The migratable state of the entrant party is a state that can be reached by only sending approved messages, receiving messages, and performing internal activities. Since the entrant party may end up not sending or receiving exactly the same messages as its predecessor, the migratable area must be recomputed after the entrant party establishes its migratable state.

Based on the adapted migratable area, each (non-entrant) party can determine its *(non-entrant) migratable state*. The non-entrant migratable state is the state in a marking history *Mh* to which a party can migrate, because it can be reached by exchanging messages that are acceptable to all parties, including the new party. It satisfies the criteria that:

- the transition *t* of the next element (*t*,*M*) in *Mh* is an element of the migratable area; and
- there is no previous element (*t*,*M*) in *Mh* for which this holds; or
- if such a state does not exist, the migratable state is the last element of *Mh*.

If the migratable state of a party is not the last state in its message history that party needs to roll back the activities that happened after that state. In addition, the entrant party may need to take some compensatory actions to update its internal state. Finally, in case the global controller keeps track of the state of the choreography as a whole, it can update that state as well via rollback operations.

4.3 Adaptation Strategies

There are two strategies for migrating the parties involved to the latest migratable state: the global adaptation strategy and the self-responsible adaptation strategy.

In the global adaptation strategy, the global controller and the entrant party determine the migratable state together. They can then force the other parties to move to that state, by replaying the message history up to that point. This strategy is more suitable in a collaboration in which the other parties have no support for computing their own migratable state and/or have no operations that enable them to roll back previously executed activities.

In the self-responsible adaptation strategy, the other parties receive the migratable area and have to compute their migratable state themselves. This requires that they have operations to compute the migratable state and to roll back previously executed activities. However, this is clearly a much more efficient way of adapting a collaboration to a switch between parties. Note that the sequence diagram from Figure 6 corresponds to this strategy rather than the global adaptation strategy.

Of course a combination of the two strategies, in which the parties that can use the self-responsible strategy and the parties that cannot use that strategy rely on global adaptation, is probably the most useful in practice.

4.4 Example

For the example, suppose that the choreography from Figure 2 is in the state $\{(p_{16},1),(p_{13},1),(p_{29},1),(p_{32},1)\}$, while party 2 (Figure 3) is in the state $\{(p_{13},1),(p_{29},1)\}$. This means that party 2 has executed transitions to arrive in that state and consequently has a marking history that could look like $<(\perp,\{(p_7,1)\}),(t_7,\{(p_8,1)\}),\ldots,(t_{13},\{(p_{13},1),(p_{14},1)\}),(t_{15},\{(p_{13},1),(p_{29},1)\})>$. In addition, it has a message history that links border activities to messages, like $\{(t_6,msg_a),(t_{15},msg_b)\}$. (Note that we identify transitions and activities by the same labels.)

Assuming the self-responsible adaptation strategy, the algorithm must now determine the migratable area. Let's assume that the entrant party (Figure 4) does not accept msg_b. The migratable area then consists of $\{t_{15}, t_{17}\}$. As a result the migratable state of the entrant party is $\{(p_{13},1),(p_{14},1)\}$, because this is the latest possible state that it can reach that does not involve a message that cannot be accepted.

The party in the choreography that is dependent on msg_b must now also adapt its state by rolling back t_{17}, because that transition relies on msg_b being received and is part of the migratable area.

5 Algorithms

This section presents the algorithms that implement the operations that are introduced in the previous section. Algorithms are presented for finding the migratable area, as well as for the global adaptation strategy and the self-responsible adaptation strategy. These algorithms use operations to ‘approve’ a message, ‘rollback’ an activity and ‘compensate’ an activity, which we assume to be defined separately depending on the specific requirements of the parties.

5.1 Find migratable area

The algorithm that is used to compute the migratable area is presented in Figure 7. The inputs for this algorithm are an entrant party denoted by *prt*, and the message history of the old party (line 1). In addition, we assume that the global variables *SBA*, *RBA* and *CR* are available, because they are known by the global controller.

The algorithm retrieves all activities that correspond to send border activities (lines 3 and 4). If the new party does not approve the message that was sent as part of the border activity, it becomes part of the migratable area A_{mig} (lines 5 and 6). In that case, the corresponding receive activities also become part of the migratable area (lines 7 and 8). These activities are eventually returned as the migratable area.

```
1:  procedure FINDMIGRATABLEAREA($prt, Msh$)
2:      $A_{mig} \leftarrow \varnothing$
3:      for $a \in prt$ do
4:          if $a \in SBA$ then
5:              if $\neg approve(Msh(a))$ then
6:                  $A_{mig} \leftarrow A_{mig} \cup \{a\}$
7:                  for $(a, a') \in CR$ do
8:                      $A_{mig} \leftarrow A_{mig} \cup \{a'\}$
9:                  end for
10:             end if
11:         end if
12:     end for
13:     return $A_{mig}$
14: end procedure
```

Figure 7. Find Migratable Area Procedure

5.2 Global adaptation strategy

Figure 8 presents the algorithm that implements the global adaptation strategy.

The inputs for this algorithm are an entrant party denoted by *prt*, and the message history of the old party that it replaces (line 1). Note that the message history must be reconstructed by the global controller, in case the global controller does not know the message history of the old party.

The algorithm is initialized by first reinitializing the collaboration (line 2), obtaining the migratable area (line 3) and initializing the current marking *M*, the marking history *Mh* and the messaging history *Msh* (line 4-6).

After the initialization, the algorithm repeatedly fires the transition of the entrant party that it *can* fire according to the history of the old party. This is done as follows. First, the algorithm computes the enabled transitions of the entrant party in the usual way (line 9). It then chooses one of the transitions to fire, or rather, lets the entrant party choose the transition to fire (line 10). There are then three choices to proceed, depending on the type of transition that is chosen.

If the transition represents a send border activity that is accepted by the entrant party (i.e. that is not part of the migratable area), the algorithm sends the corresponding message to the other parties to progress in their own behaviour and adds that message to the messaging history of the entrant party (line 11-14). In these lines, *send* is the function that, given a receive activity, returns the corresponding send activity; *receive* is the function that, given a send activity, returns the corresponding receive activity.

If the transition represents a receive border activity, the corresponding message has been received and is not changed with respect to the collaboration in which the old party participated, we can assume that the collaboration can continue based on the behaviour of the old party (line 15-18). Consequently, we add that message to the messaging history.

If the transition represents an internal activity or silent transition, it can be executed at will (line 19-21).

```
1:  procedure GLOBALADAPTATION($prt, Msh_{old}$)
2:      $initialize()$
3:      $A_{mig} \leftarrow findMigratableArea(prt, Msh_{old})$
4:      $M \leftarrow M_0$
5:      $Mh \leftarrow <(\perp, M_0)>$
6:      $Msh \leftarrow \varnothing$
7:      repeat
8:          $continue \leftarrow false$
9:          $T_e = \{t \in T | \forall p \in \bullet t : M(p) > 0\}$
10:         choose $t \in T_e$
11:         if $\ell(t) \in SBA \wedge \neg\ell(t) \in a_{mig}$ then
12:             $Msh \leftarrow Msh \cup send(\ell(t))$
13:             $continue \leftarrow true$
14:         end if
15:         if $\ell(t) \in RBA \wedge recv(\ell(t)) \in Msh_{old}(\ell(t))$ then
16:             $Msh \leftarrow Msh \cup Msh_{old}(\ell(t))$
17:             $continue \leftarrow true$
18:         end if
19:         if $\ell(t) \in IA \cup \{\tau\}$ then
20:             $continue \leftarrow true$
21:         end if
22:         if $continue$ then
23:             for $p \in P$ do
24:                 $M(p) \leftarrow M(p) - |\{p|(p,t) \in F\}|$
25:                 $M(p) \leftarrow M(p) + |\{p|(t,p) \in F\}|$
26:             end for
27:             $Mh \leftarrow Mh \sqcup (t, M)$
28:         end if
29:     until $\neg continue$
30:     $A_{mig} \leftarrow findMigratableArea(prt, Msh)$
31:     return $A_{mig}$
32: end procedure
```

Figure 8. Global Adaptation Algorithm

For all these types of transition, the transition is fired and the resulting marking is stored in the marking history (line 22-28). In these lines $\sqcup$ is the function that adds an element to a sequence. The algorithm then continues to choose the next transition in the new party's behaviour to execute.

If the transition matches none of these cases, it cannot be fired in the same way as for the old situation. Consequently, in that case the algorithm completes and leaves all parties in the new state from which they can pick up their collaboration. Finally, the algorithm re-computes the migratable area, based on the messages that it actually sent and received itself (line 30) instead of the messages that the old part sent and received.

5.3 Self-adaptation strategy

Figure 9 presents the algorithm that implements the self-adaptation strategy. This algorithm is only used by other parties than the entrant party. When the self-adaptation algorithm is used, the entrant party follows the global adaptation strategy, but does not actually send or receive messages. The other parties in the collaboration then update their state based on the migratable area as follows.

The algorithm first finds the migratable state (line 3-5), which is the last state in the marking history before a transition that is in the migratable area. We use *#ix* to represent getting the *i*th element from a tuple *x* and *s*[*i*] to represent getting the *i*th element from a sequence *s*, assuming that the elements in a sequence are numbered 0 to |*s*|−1. After that, the algorithm rolls back all transitions that occurred after the migratable state (in reverse order) and removes the messages that were sent of received by those transitions from the message history (line 6-9). Finally, the algorithm updates the marking history.

```
1:  procedure SELFADAPTATION($A_{mig}$)
2:      $last \leftarrow 0$
3:      while $\ell(\#1Mh[last]) \notin A_{mig}$ do
4:          $last \leftarrow last + 1$
5:      end while
6:      for $i \leftarrow |Mh| - 1$ to $last + 1$ do
7:          $rollback(Mh[i])$
8:          $Msh \leftarrow Msh - \{(\ell(Mh[i]), Msh(\ell(Mh[i])))\}$
9:      end for
10:     $Mh \leftarrow Mh[0 \ldots last]$
11: end procedure
```

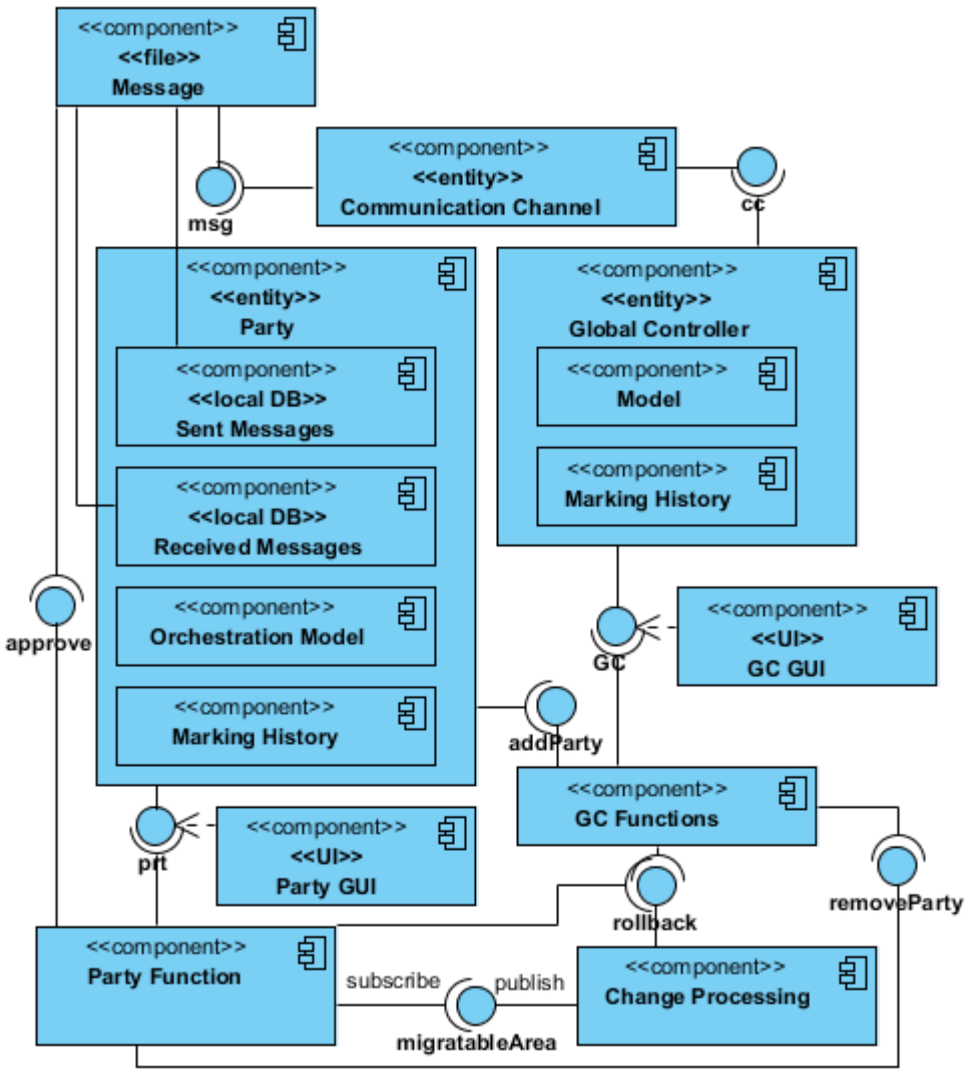
Figure 9. Self-Adaptation Algorithm

5.4 Prototype Tool Support

To demonstrate the technique and the algorithms that were developed to support switching parties in a collaboration at run-time, we developed a prototype tool. The tool can be obtained from `http://is.tm.tue.nl/research/adaptation` along with more information about the way in which it can be used.

Figure 10 shows the components of the tool in a UML component diagram. The two main components are *Party* and *Global Controller* (GC). While the global controller is only instantiated once, i.e. it has a static modifier, the parties can be instantiated multiple times. Both the Party component and the Global Controller component contain two sub-components: a *Model* and a *Marking History*. The model is a marked labelled Petri net and the marking histories are realized by an ordered list of pairs. In addition, a party contains two local data stores in order to store the sent and received messages.

In addition to the main components, the *Communication Channel* represents the communication relations between the parties. Communication is based on message passing. The *Message* component has knowledge of the messages that are being passed along communication channels. The functions that are used both in the Party and the Global Controller component are encapsulated in the *Party Function* component and the GC Function component respectively. These functions mainly include the instantiation of a model, creating an initial marking, finding the enabled transitions and firing enabled transitions. The *Change Processing* component is used to apply the runtime change functions.

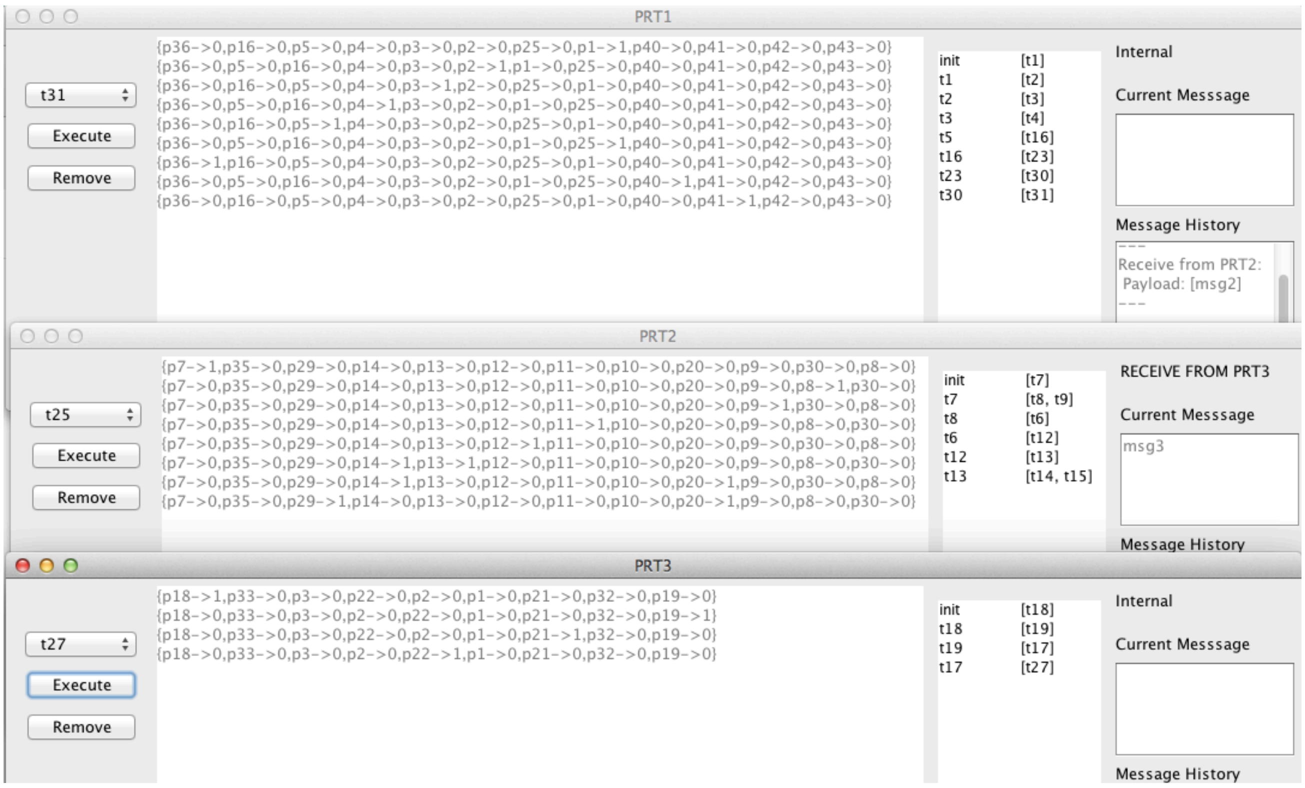


<<component>> <<file>> Message
<<component>> <<entity>> Communication Channel
msg
cc
<<component>> <<entity>> Party
<<component>> <<local DB>> Sent Messages
<<component>> <<local DB>> Received Messages
<<component>> Orchestration Model
<<component>> Marking History
<<component>> <<entity>> Global Controller
<<component>> Model
<<component>> Marking History
<<component>> <<UI>> GC GUI
GC
approve
addParty
<<component>> GC Functions
pft
<<component>> <<UI>> Party GUI
rollback
removeParty
<<component>> Party Function
subscribe
publish
migratableArea
<<component>> Change Processing

Figure 10. High-level Architecture of the tool

For prototyping purposes, two user interface components, *Party GUI* and *GC GUI*, are developed that enable an end-user to observe and control the state of the Party components and the Global Controller component.

For illustration purposes, Figure 11 shows a screenshot of the prototype's Party GUI. The screenshot shows the current state of three parties, consisting of their marking and message history. The user interface has controls for executing an activity and for removing a party. When a party is removed, the global controller produces a pop-up (Figure 12) that facilitates the selection of a new (entrant) party to replace the removed party and an adaptation strategy to realize the switch. The global controller then executed the chosen strategy and effectuates the switch.



PRT1

t31

Execute

Remove

```
{p36->0,p16->0,p5->0,p4->0,p3->0,p2->0,p25->0,p1->1,p40->0,p41->0,p42->0,p43->0}
{p36->0,p5->0,p16->0,p4->0,p3->0,p2->1,p1->0,p25->0,p40->0,p41->0,p42->0,p43->0}
{p36->0,p16->0,p5->0,p4->0,p3->1,p2->0,p25->0,p1->0,p40->0,p41->0,p42->0,p43->0}
{p36->0,p5->0,p16->0,p4->1,p3->0,p2->0,p1->0,p25->0,p40->0,p41->0,p42->0,p43->0}
{p36->0,p16->0,p5->1,p4->0,p3->0,p2->0,p25->0,p1->0,p40->0,p41->0,p42->0,p43->0}
{p36->0,p5->0,p16->0,p4->0,p3->0,p2->0,p1->0,p25->1,p40->0,p41->0,p42->0,p43->0}
{p36->1,p16->0,p5->0,p4->0,p3->0,p2->0,p25->0,p1->0,p40->0,p41->0,p42->0,p43->0}
{p36->0,p5->0,p16->0,p4->0,p3->0,p2->0,p1->0,p25->0,p40->1,p41->0,p42->0,p43->0}
{p36->0,p16->0,p5->0,p4->0,p3->0,p2->0,p25->0,p1->0,p40->0,p41->1,p42->0,p43->0}
```

init	[t1]
t1	[t2]
t2	[t3]
t3	[t4]
t5	[t16]
t16	[t23]
t23	[t30]
t30	[t31]

Internal

Current Messsage

Message History

```
---
Receive from PRT2:
 Payload: [msg2]
---
```

PRT2

t25

Execute

Remove

```
{p7->1,p35->0,p29->0,p14->0,p13->0,p12->0,p11->0,p10->0,p20->0,p9->0,p30->0,p8->0}
{p7->0,p35->0,p29->0,p14->0,p13->0,p12->0,p11->0,p10->0,p20->0,p9->0,p8->1,p30->0}
{p7->0,p35->0,p29->0,p14->0,p13->0,p12->0,p11->0,p10->0,p20->0,p9->1,p30->0,p8->0}
{p7->0,p35->0,p29->0,p14->0,p13->0,p12->0,p11->1,p10->0,p20->0,p9->0,p8->0,p30->0}
{p7->0,p35->0,p29->0,p14->0,p13->0,p12->1,p11->0,p10->0,p20->0,p9->0,p30->0,p8->0}
{p7->0,p35->0,p29->0,p14->1,p13->1,p12->0,p11->0,p10->0,p20->0,p9->0,p8->0,p30->0}
{p7->0,p35->0,p29->0,p14->1,p13->0,p12->0,p11->0,p10->0,p20->1,p9->0,p30->0,p8->0}
{p7->0,p35->0,p29->1,p14->0,p13->0,p12->0,p11->0,p10->0,p20->1,p9->0,p8->0,p30->0}
```

init	[t7]
t7	[t8, t9]
t8	[t6]
t6	[t12]
t12	[t13]
t13	[t14, t15]

RECEIVE FROM PRT3

Current Messsage

```
msg3
```

Message History

PRT3

t27

Execute

Remove

```
{p18->1,p33->0,p3->0,p22->0,p2->0,p1->0,p21->0,p32->0,p19->0}
{p18->0,p33->0,p3->0,p2->0,p22->0,p1->0,p21->0,p32->0,p19->1}
{p18->0,p33->0,p3->0,p22->0,p2->0,p1->0,p21->1,p32->0,p19->0}
{p18->0,p33->0,p3->0,p2->0,p22->1,p1->0,p21->0,p32->0,p19->0}
```

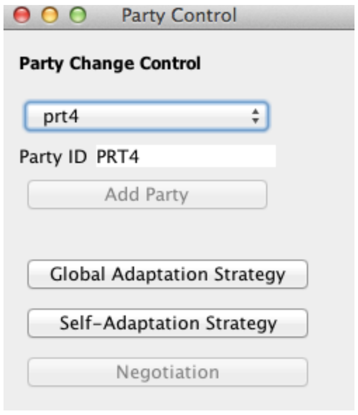
init	[t18]
t18	[t19]
t19	[t17]
t17	[t27]

Internal

Current Messsage

Message History

Figure 11. Screenshot of prototype



Party Control

Party Change Control

prt4

Party ID PRT4

Add Party

Global Adaptation Strategy

Self-Adaptation Strategy

Negotiation

Figure 12. Screenshot of adaptation pop-up

6 Related Work

Dynamic adaptation to change is a continuing concern within both service-oriented computing and business process management.

In the area of service-oriented computing several studies have been conducted towards changing a service composition at run-time. However, these primarily focus on composing new services (at run-time), rather than on what happens when an already running service is changed. For example, Zeng et al. [7] presented AgFlow, which is a middleware platform that facilitates selecting services to compose at run-time, based on QoS characteristics. Tan et al. [8] developed techniques to determine the compatibility of web services and ways of making them compatible without changing their internal behaviour. Similarly, Shan et al. [9] propose a framework that enables services to adapt to each other in case their behaviours are incompatible. Literature surveys exist, such as a [10], [11] and [12], that classify the different methods to service composition at run-time.

In the area of business process management several studies exist that investigate techniques for changing an already running business process. However, these techniques focus on intra-organizational business processes rather than inter-organizational collaborations. Overviews of possible run-time changes are developed by Rinderle-Ma et al. [13] and Schonenberg et el. [14]. In addition to that, Weber et al. [15] suggested a set of 18 change patterns and seven change support features that need to be supported by a flexible workflow system. In another study, Zimmermann and Doehring [16] present workflow adaptation patterns, which have been designed based on a combination of BPMN workflow models, rules and events. A study on the support of flexibility in a team-based process has been also conducted by Rinderle-Ma et al. [17]. In line with these studies, a comprehensive adaptive workflow system, called ADEPT2, has been developed [18]. In terms of Petri nets, Zhang et al. [19] proposed an algorithm to transfer multiple running instances from an old process model to a new one by introducing the concept of Dynamic Change Region. Bergemann et al. [20] presented an integration architecture that decouples a workflow client application from the underlying workflow management systems to increase the flexibility. While dynamically changing processes at run-time is one solution to facilitate process flexibility, other solutions are investigated as well. For example, van der Aalst et al. [21] designed a more flexible way of executing processes, based on a declarative model. A similar approach exists in the area of service-oriented computing [22].

7 Possible Application

The technology that is described in this deliverable has applications in the context of the scenarios that are introduced for the GET Service platform in Deliverable 1.1 and have been refined into demo scenarios in Deliverable 2.4.1.

The first demo scenario describes a situation in which a multi-modal route for a full container load is planned and tracked using the GET Service platform. The container load must be transported from Venlo to Rotterdam and then onward by ship. The transportation plan is to go from Venlo to Eindhoven by truck and take a train from there to Rotterdam. When using the GET Service platform, the planner has insight into real-time information about the transportation infrastructure, including both the current traffic conditions and transportation-related events, such as the waiting time at customs or the gate at the harbour. To this end a transportation process is constructed that describes the tasks that have to be executed during the transportation and that automatically invokes information aggregation services.

In the demo scenario, it is detected that the truck will arrive late and miss the connection to the train. This requires that the transportation plan is re-planned from the train connection onwards to the harbour. As the transportation plan is changed, the transportation process has to be changed with it as well as the statements in the transportation process that determine what information is aggregated during the execution of each task. For example, originally the transportation process would have contained a task for reserving the train and a task of the train driving, during which the GPS location of the train is being monitored. The changed transportation process would not contain those tasks, or would contain tasks for reserving and monitoring a different train.

When the transportation process corresponding to the old transportation plan is replaced by that corresponding to the new transportation plan, the state of the transportation process must be migrated. For example, if the truck was registered as currently executing the ‘driving’ task, it must be registered as such in the new process (i.e.: the state of the truck is migrated). While migrating the state, it would become clear that the state of certain tasks cannot be directly migrated or even needs to be compensated. For example, the state of the ‘reservation’ task for the train cannot be migrated and may even need to be compensated by cancelling the reservation, as we are no longer reserving the train or are reserving a different train.

The technology described in this deliverable can be used to automatically change the transportation process in this manner.

9 Conclusion

This deliverable presented technology that enables a web service in a collaboration to be switched with another web service, and its state to be transferred to the new web service. This technology can be used in case a transportation plan changes and the composition of web-services that helps to monitor and control the transportation plan changes with is. The deliverable presents both the technology and the algorithms that are required to facilitate such a switch, as well as a prototype tool that demonstrates their feasibility.

When transferring the state. Two steps must be taken into account. First it must be determined to which extent the state can be transferred from the removed party to the entrant party. Second, the new state must be communicated to the other parties, who may need to adapt as well. This deliverable presented two algorithms to perform these steps.

Both algorithms that we developed rely on the existence of a global controller that has the ability to monitor and control all web services in the collaboration. While the existence of such a global controller can be realistic when the web services in the collaboration are owned by a single organization, it is not realistic in a scenario where the web services are owned by different organizations. Therefore, as future work we aim to develop algorithms that do not require a global controller.

The deliverable introduces the technology on a conceptual level by means of Petri nets. However, in a practical setting the behaviour of web services is defined in terms of BPEL or BPMN. While a conceptual step is necessary in the development of the algorithms described in this deliverable, they must be translated to work on BPEL or BPMN in order to make them practically applicable. This is another direction for future work that we aim to pursue.

10 References

[1] P. Grefen, “Networked business process management,” *International Journal of IT/Business Alignment and Governance (IJITBAG)*, vol. 4, no. 2, pp. 54–82, 2013.

[2] R. Dijkman and M. Dumas, “Service-oriented design: a multi-viewpoint approach,” *International Journal of Cooperative Information Systems (IJCIS)*, vol. 13, no. 4, pp. 337–368, 2004.

[3] C. Peltz, “Web services orchestration and choreography,” *Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[4] R. M. Dijkman, M. Dumas, and C. Ouyang, “Semantics and analysis of business process models in bpmn,” *Inf. Softw. Technol.*, vol. 50, no. 12, pp. 1281–1294, 2008.

[5] C. Ouyang, E. Verbeek, W. M. Van Der Aalst, S. Breutel, M. Dumas, and A. H. Ter Hofstede, “Formal semantics and analysis of control flow in ws-bpel,” *Science of Computer Programming*, vol. 67, no. 2, pp. 162–198, 2007.

[6] R. Seguel, R. Eshuis, and P. Grefen, “Generating minimal protocol adaptors for loosely coupled services,” in *Web Services (ICWS), 2010 IEEE International Conference on*. IEEE, 2010, pp. 417–424.

[7] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “Qos-aware middleware for web services composition,” *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311–327, 2004.

[8] W. Tan, Y. Fan, and M. Zhou, “A petri net-based method for compatibility analysis and composition of web services in business process execution language,” *Automation Science and Engineering, IEEE Transactions on*, vol. 6, no. 1, pp. 94–106, 2009.

[9] Z. Shan, A. Kumar, and P. Grefen, “Towards integrated service adaptation a new approach combining message and control flow adaptation,” in *Web Services (ICWS), 2010 IEEE International Conference on*. IEEE, 2010, pp. 385–392.

[10] J. Rao and X. Su, “A survey of automated web service composition methods,” in *Semantic Web Services and Web Process Composition*. Springer, 2005, pp. 43–54.

[11] D. A. D’Mello, V. Ananthanarayana, and S. Salian, “A review of dynamic web service composition techniques,” in *Advanced Computing*. Springer, 2011, pp. 85–97.

[12] A. Alamri, M. Eid, and A. El Saddik, “Classification of the state-of-the-art dynamic web services composition techniques,” *International Journal of Web and Grid Services*, vol. 2, no. 2, pp. 148–166, 2006.

[13] S. Rinderle-Ma, M. Reichert, and P. Dadam, “Correctness criteria for dynamic changes in workflow systems - a survey,” *Data & Knowledge Engineering*, vol. 50, no. 1, pp. 9–34, 2004.

[14] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. van der Aalst, “Process flexibility: A survey of contemporary approaches,” in *Advances in Enterprise Engineering I*. Springer, 2008, pp. 16–30.

[15] B. Weber, M. Reichert, and S. Rinderle-Ma, “Change patterns and change support features–enhancing flexibility in process-aware information systems,” *Data & knowledge engineering*, vol. 66, no. 3, pp. 438–466, 2008.

[16] B. Zimmermann and M. Doehring, “Patterns for flexible BPMN workflows,” in *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM, 2012, p. 7.

[17] S. Rinderle-Ma, M. Reichert, and P. Dadam, “Flexible support of team processes by adaptive workflow systems,” *Distributed and Parallel Databases*, vol. 16, no. 1, pp. 91–116, 2004.

[18] M. Reichert and P. Dadam, “Enabling adaptive process-aware information systems with adept2,” 2009.

[19] D. Zhang, D. Cao, L. Wen, and J. Wang, “An efficient approach for supporting dynamic evolutionary change of adaptive workflow,” in *Progress in WWW Research and Development*. Springer, 2008, pp. 684–695.

[20] T. Bergemann, A. Hausotter, and A. Koschel, “Keeping workflow-enabled enterprises flexible: Wfms abstraction and advanced task management,” in *Grid and Pervasive Computing Conference, 2009. GPC’09. Workshops at the*. IEEE, 2009, pp. 19–26.

[21] W. M. van Der Aalst, M. Pesic, and H. Schonenberg, “Declarative workflows: Balancing between flexibility and support,” *Computer Science-Research and Development*, vol. 23, no. 2, pp. 99–113, 2009.

[22] G. Cugola, C. Ghezzi, and L. S. Pinto, “Dsol: a declarative approach to self-adaptive service orchestrations,” *Computing*, vol. 94, no. 7, pp. 579–617, 2012.