Toward synchronization between decentralized orchestrations of Composite Web Services

Walid Fdhila
LORIA - INRIA Nancy - Grand Est
F-54500 Vandœuvre-lès-Nancy, France
Email: fdhilawa@loria.fr

Claude Godart
LORIA - INRIA Nancy - Grand Est
F-54500 Vandœuvre-lès-Nancy, France
Email: godart@loria.fr

***Abstract*—Web service paradigm and related technologies have provided favorable means for the realization of collaborative business processes. From both conceptual and implementation points of view, the business processes are based on a centralized management approach. Nevertheless, it is very well known that the enterprise-wide process management where processes may span multiple organizational units requires particular considerations on scalability, heterogeneity, availability and privacy issues, that in turn, require particular consideration on decentralization. In a previous work [10], we have described a flexible methodology for splitting a centralized process specification into a form that is amenable to a distributed execution. The approach is based on the computation of very basic dependencies between process elements. In this paper, we extend this approach to support advanced patterns such as *Loops*, *Multiple instances* and *Discriminator*, and incorporate the necessary synchronization between the different processing entities. We also detail our interconnection mechanism and explain how to handle control and data dependencies between activities of the different partitions through asynchronous message exchanges. The proposed methodology preserves semantics of the centralized process with a peer-to peer interactions among the derived decentralized processes.**

I. Introduction

With the emergence of the open standards, Web-based applications have become an intuitive support for Business-to-Business (B2B) and Business-to-Costumer (B2C) processes[27]. Particularly, Service Oriented Architectures (SOA) have the potential to enhance these by allowing autonomous and distributed business processes to interact with each other. Despite the decentralized nature of the context of B2B and B2C interactions, the conception and implementation of a typical business process rely on a centralized execution setting[1] which fail to address issues such as high availability, failure resilience and scalability. The relevant research literature on business management confirms that the decentralization is a critical need for several reasons[20][15][8][29][6]:

- scalability which is one of the pressing needs since many concurrent processes or instances of the same process are executed simultaneously, and a centralized architecture can cause a performance bottleneck,
- mutually equitable business relationships where no organization holds the control of the overall process,
- fault tolerance where different parts of a process can be executed even if some components fail,
- decentralization, since the systems are inherently distributed, not lend themselves to centralized control,
- distributing the data to reduce network traffic and improve transfer time as well as concurrence. In addition, especially in processes where large data volumes are transported during the orchestration, overall performance also may benefit from having the execution engines in proximity to the target services, by jointly selecting the providers of the process activities and the providers to store instance data in a way that allows to minimize data transfer during process execution.

In order to deal with the shortcomings of centralized process executions, we have been investigating decentralized orchestrations [10]. In our prior work, we proposed a process decentralization technique to split a composite web service specification into a semantically-equivalent set of partitions. The approach implements both control and data dependencies of the centralized specification with P2P interactions. Data are transferred directly between partitions using asynchronous messaging. This reduces data over the network by sending them directly from their point of generation to their point of consumption. The main advantage of the developed approach is the flexibility that it provides in terms of concepts and structures that it manipulates. This, in turn, allows the extension of the algorithms to the different needs of decentralization. In sharp contrast to previous works, our operation of decentralization computes the abstract process constructs, *i.e.*, workflow patterns[25]. This methodology separates the implementation details from the high-level reasoning that provides a more complete and generic solution to the decentralization problem. The technique uses a dependency table that resumes direct dependencies between process activities. Next, it generates automatically transitive dependency tables resuming the transitive dependencies between activities having some common properties, *i.e.*, activities invoking the same service or provider. It generates the corresponding sub-processes by specifying their mutual interconnections.

The approach we proposed for decentralization implements only business processes composed of basic patterns ($AND-split$, $OR-split$, $AND-join$,...). However, most

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8275
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8275

of today's processes are complex and may need advanced patterns to implement some repetitive blocs of activities or to instantiate them with multiple instances. For this purpose, we are trying in this paper to extend the previously developed approach to support advanced patterns. The techniques we propose, are complementary to our previous work thereby decentralizing processes with *Loops*, *Multiple instances* and *Discriminator* constructs, and then, provide a more complete solution to the problem. Nevertheless, the flexibility introduced by the derived decentralized processes on the other hand raises new requirements like synchronization between them. For this purpose, we will revise our previously proposed mechanism to synchronize the derived partitions and explain how we translate the connectivity and communication between activities of the initial process to those between activities belonging to different sub-processes. This revision includes messages contents, the interaction patterns used for synchronization and the advantage of patterns replication for message exchanges minimization. It also separates control synchronization from data synchronization and shows how the decentralized derived processes work together through a running example.

The remainder of this paper is structured as follows. Section 2 presents a motivation example. In section 3, we give an approach overview as well as the required definitions. Section 4 explains the different steps to partition *Loops*, *Multiple instances* and *Discriminator* patterns. Synchronization process is detailed in section 5. In section 6, we describe the related work, the issues involved in process decentralization as well as a comparison with our approach. Finally, section 7 summarizes the ideas explained in the paper and outlines some future directions.

II. MOTIVATING EXAMPLE

We present a brief overview of the problem along with our solution mechanism using the following running example. Lets consider a credit approval business process *CreditAppr* depicted in figure 1. The composite service *CreditAppr* involves four services: *CollectInfo*, *AssessRisk*, *Decide*, and a *Notify* service. A customer makes a new credit request to the *CreditAppr* composite service. The latter invokes the *CollectInfo* service to know more about the customer. Once the data are available, *CreditAppr* sends them to *AssessRisk* who considers whether the risk of the credit is low or high. The risk evaluation is then sent to the *Decide* service who decides even to approve the credit or not. If the assessment is low then the lowest rate is calculated and a reply is sent to the costumer following approval. If the assessment is high risk then the customer is asked to apply through an alternative process. Customer notification is achieved through the *Notify* service. It should be noted that control dependencies and data transfer between services are managed centrally by *CreditAppr*. The latter acts as an intermidiary between all the services and the client. This bottleneck may cause degradation of performance due to large delay between the request and response from and to the client. It may also cause additional traffic in terms of exchanged messages.

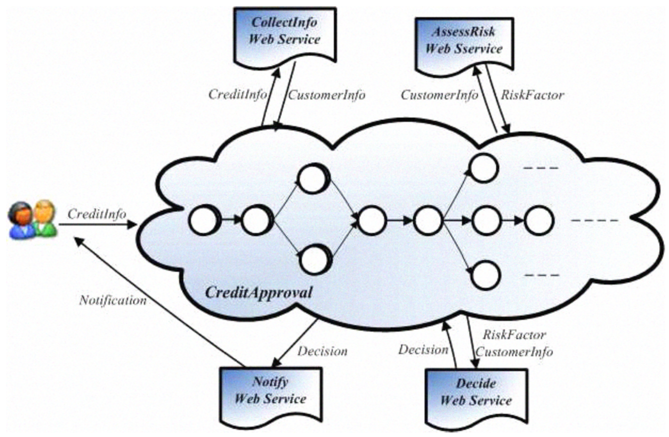

Fig. 1. Centralized Credit Approval process

Figure 2 depicts a possible decentralized execution setting for the same process. The *CreditAppr* is partitioned into four components that are executed by four distributed engines. Together, the four engines perform the role of the centralized *CreditAppr*. The data produced by a service is routed directly to the service that must consume them. For example, the *riskfactor* generated by *AssessRisk* is routed directly to *Decide* as depicted in Figure 2. In contrast to centralized architecture, decentralized orchestrations are more cooperative. This may lead to increased parallelism and reduced message overhead since fewer messages are sent. The time needed to exchange messages between a partition and its corresponding web service is quite small, since partitions are collocated with their relative web services.

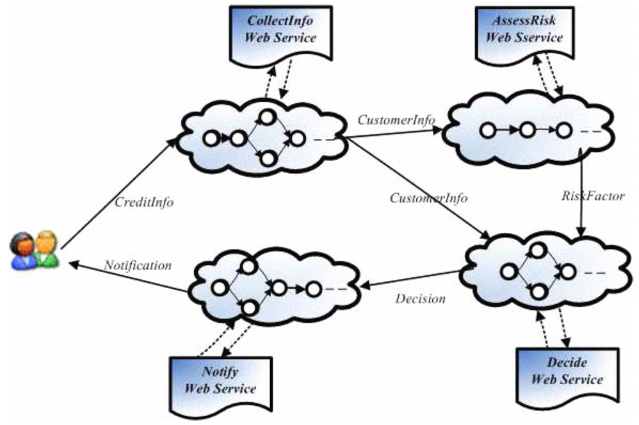

Fig. 2. Decentralized Credit Approval process

III. APPROACH OVERVIEW

In this Paper, we extend our methodology [10] for decentralizing a process specification characterizing a web service composition to support advanced patterns and detail our interconnection mechanism toward synchronization between derived processes. The developed technique consists in partitioning a composite web service into small partitions each of which has common criteria. The approach is known to be flexible that it doesn't rely on specific decentralization criterion. A criterion characterizes a common property for a set of activities. For instance, if we partition a process upon *Providers*, each partition will contain only activities invoking the same provider. The criterion choice is an important task for decentralizing efficiently a process. As an example, consider the case of partitioning a process according to the geographically collocated providers, which may result in considerable decrease in the number of messages exchanged between the geographically dispersed providers as we already mentioned in the example introduced in Figure 1. To better understand our decentralization and synchronization mechanisms, we adopt the *web services* perspective as a criterion. As a result, each partition includes only activities invoking operations of the same service. These partitions are executed independently at distributed locations (preferably collocated with the web services) and can be invoked remotely. They directly interact with each other using asynchronous messaging without any centralized control.

The developed approach doesn't presume any particular process modeling language, but simply assumes that the basic elements of a process can be specified in an abstract way to be translated to an executable process language. Throughout this paper, we use the graph based formalism [17] just for clarification reasons to guide the reader through the decentralization and synchronization steps. By definition, a process which specifies a web service composition defines the relationship between service invocations. This relationship may characterize either the control or data flow structure. Our approach takes into consideration both control and data dependencies between the process activities.

The proposed technique assumes that Processes to be decentralized are structured [16]. This means that different activities are structured through control elements such as AND-split, OR-split, AND-join, OR-join..., and for each split element, there is a corresponding join element of the same type. Additionally, the split-join pairs are properly nested. This assumption seems to be reasonable, since many works have already proposed solutions to map an arbitrary process into a structured one [18], and because most of workflow tools support structured processes.

A. Definitions

In the following, a *process* $\mathcal{P}$ is represented by a directed acyclic graph where nodes are activities and edges are data or control dependencies. Activities are depicted with boxes with the activity name inside and the web service it refers to (see Figure 9). The arcs between boxes describe the dependencies.

1) An *activity* $a \in \mathcal{A}$ consists of a one-way or a bidirectional interaction with a service via the invocation of one of its operations. In conversational compositions, different operations of a service can be invoked with the execution of different activities. The set of activities that refer to the same service s_i is denoted $\mathcal{A}_{s_i}$. A control edge characterizes the mapping relationship while a data edge characterizes the mapping relation of the output and the input values of two activities. Next, we assume that activities have identities such as $a_i{:}s_j$ where a_i denotes the activity name and s_j the invoked service
2) The *preset* of an activity a_i, denoted $\bullet a_i$, is the set of activities which can be executed just before a_i and related to it by a control or data dependency.
3) The *postset* of an activity a_i, denoted $a_i\bullet$, is the set of activities which can be executed just after a_i and related to it by a control or data dependency.

B. Interaction patterns

To preserve the initial centralized process semantics, the derived decentralized processes should interact with each other to exchange either control or data information. Service interaction patterns [28] aim at filling this gap by proposing small granular types of interactions that can be combined to the derived decentralized processes. In this paper, we use four patterns to express advanced conversations by passing control and data messages in asynchronous way.

1) *Send:* The send pattern represents a one-way interaction between two participants seen from the perspective of the sender. It is realized by a send task.
2) *Receive:* The receive pattern also describes a one-way interaction between two participants, but this time seen from the perspective of the receiver. In terms of message buffering behavior of the receiver, two cases can be distinguished. Messages that are not expected are either discarded or stored until a later point of time, when they can be consumed.
3) *One-To-Many Send:* A participant sends out several messages to other participants in parallel.
4) *One-From-Many Receive:* Messages can be received from many participants. In particular, one participant waits for messages to arrive from other participants, and each of them can send exactly one message.

IV. Patterns decentralization

A. Loop pattern decentralization

A *Loop* is a point in a process where one or more activities can be executed repeatedly [25]. It allows for the repeated sequential execution of a specified activity or a sub-process zero or more times providing a nominated condition evaluates to true. We distinguish two types of cycles: arbitrary and structured. The former have more than one entry or exit point, however in the latter the looping structure has a single entry and exit point. In this paper, we consider only structured cycles since we assumed that processes to decentralize are structured. Figure 3 depicts an example of a process including a *Loop* for three sequential activities invoking different services $a_2{:}s_2$, $a_3{:}s_3$ and $a_4{:}s_4$.

The sequence of activities is repeated while the condition *Cond* is evaluated to true. A given instance of this sequence is enabled only if the previous one is terminated. Once *Cond* is evaluated to false, a_5:s_3 can be enabled.

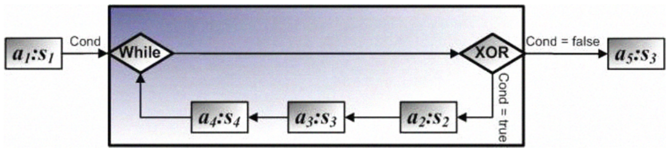

Fig. 3. process including a While pattern

Next, we refer to the process example in Figure 3 to explain our decentralization solution for *Loops*. The resulting decentralized partitions are depicted in figure 4. The process is partitioned according to services, which means that each partition includes only activities invoking the same service. We notice that only partitions including a set of activities inside the *Loop* of the main process, have a derived *While* construct with the same condition. Consequently, only P_{S1} is a *Loop* free since a_1 is not in the main *Loop*. We assume that the *Loop* condition *Cond* is known directly after a_1 execution. This condition should be sent to partitions which need *Cond* to execute their local derived *Loops* P_{S2}, P_{S3} and P_{S4}. For this purpose, we use the previously introduced interaction patterns *One-To-Many Send* and *Receive*. Once *Cond* is received by P_{S2}, P_{S3}, P_{S4}, a simple exclusive choice is enabled to choose either to execute or skip the *Loop*. If *Cond* is evaluated to false, then a dummy activity with a zero execution time is enabled. In this case, P_{S2} and P_{S4} terminate, while P_{S3} continues and handles a_5. Otherwise, if *Cond* is evaluated to true, then each of P_{S2}, P_{S3} and P_{S4} loops would be enabled as follows:

1) P_{S3} and P_{S4} will be blocked on the *receive(sync)* activity waiting for a synchronization messages, while P_{S2} executes a_2.
2) Once a_2 terminates, a synchronization message is sent to P_{S3} using the *Send* pattern to enable a_3 execution. Then, P_{S2} blocks in *receive(cond)* activity.
3) Once a_3 terminates, a synchronization message is sent to P_{S4} to enable a_4 execution. Then, P_{S3} blocks in *receive(cond)* activity.
4) a_4 executes, then calculates the new value of *cond* and sends it to P_{S2} and P_{S3}.
5) If *cond* is evaluated to false, then P_{S4} and P_{S2} terminates, and P_{S3} enables a_5. Otherwise, repeat the four first steps.

The *send(sync)* and *receive(sync)* activities are used to synchronize activities of different partitions by exchanging control messages. The resulting partitions are structured and preserve the semantics of the centralized process. Our technique for decentralizing *Loops* is not specific to the introduced example. For instance, if we consider a *Loop* on a given sub-process. In this case, we begin by partitioning the sub-process itself using our *Loop* free process partitioning

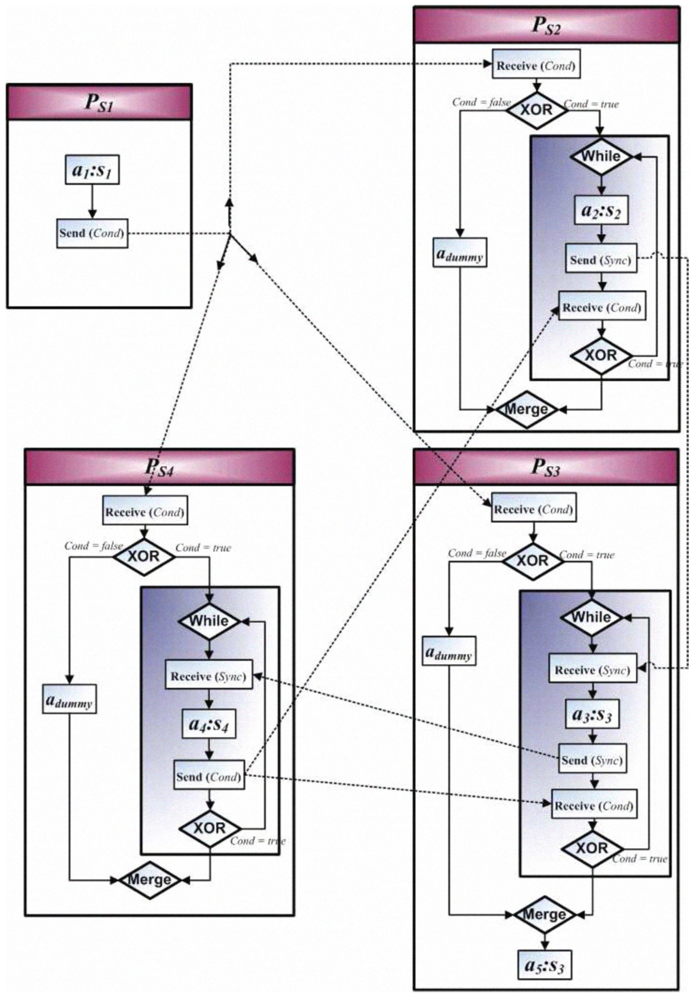

Fig. 4. Decentralized While pattern

technique [10]. Then we encapsulate the derived partitions into *Loops* and add the corresponding synchronization activities using the interaction patterns.

B. Multiple instances decentralization

The *Multiple instances* pattern provides a means of creating multiple instances of a given task [25]. As a result, within a given process instance, multiple instances of a set of activities can be created. These instances are independent of each other and run concurrently. Each of the instances of the multiple instances task that are created must execute within the context of the process instance from which they were started. For example, assume an order process in which an incoming order contains a number of order lines. For each of these order lines, a check activity needs to be executed. We distinguish two types of multiple instances pattern: with and without synchronization. In the latter, there is no requirement to synchronize the instances upon completion. However, in the former, it is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered.

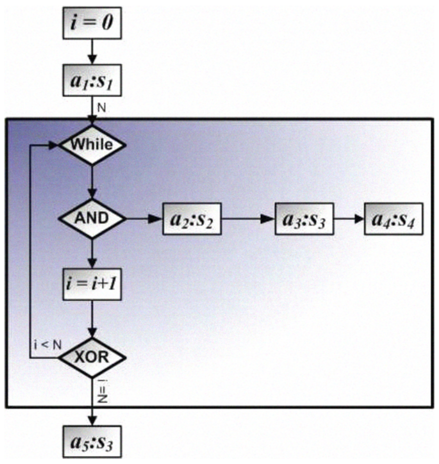

Fig. 5. Multiple instances pattern

An example of the *multiple instances without synchronization* pattern implementation is shown in Figure 5. The process is composed of five activities invoking four services. After the completion of a_1:s_1 the number of required instances N is determined, and N instances of the sequence a_2:s_2, a_3:s_3 and a_4:s_4 are enabled. These instances run concurrently. In this example, no synchronization between instances is needed and then, a_5:s_3 can be enabled immediately after the multiple instances have been enabled. As a result, a_5 instance can terminate while a_2, a_3 and a_4 instances of the multiple instances activity are still running. If we consider the case where multiple instances with synchronization pattern is used, a_5 would be enabled only after the completion of all instances of the multiple instances activity. To achieve synchronization in the same example, it is possible to implement a_4 such that once it is completed, it sends an event to some external event queue. Activity a_5 can be proceeded by another activity that consumes the events from the queue and triggers a_5 only if the number of events in the queue is equal to the number of instances of activity a_4. However, this solution is not supported by all workflow engines. We show later how to deal with *multiple instances with synchronization* patterns.

Figure 6 illustrates our decentralization technique for the *multiple instances without synchronization* pattern using the process example introduced in Figure 5. The technique is quite similar to that used for *Loops*, except that the multiple instances of the sequence a_2, a_3 and a_4 run concurrently. Assume that A_i is the ith instance of the sequence a_2, a_3 and a_4. In contrast to *Loops*, A_{i+1} doesn't need to wait until the completion of A_i to be enabled, but runs concurrently.

We assume that the instances number N is known directly after a_1 execution during run time. Then N should be sent to partitions having a derived *Multiple instance activity*, namely P_{S2}, P_{S3} and P_{S4}. According to N value, P_{S2}, P_{S3}, P_{S4} choose either to execute or skip their *multiple instances activities*. If N is evaluated to $zero$, then a dummy activity with a zero execution time is enabled. In this case, P_{S2} and P_{S4} terminate, while P_{S3} continues and handles a_5. Otherwise, if N is greater than $zero$, then each of P_{S2}, P_{S3} and P_{S4} *multiple instances activities* would be enabled as follows:

1) P_{S3} and P_{S4} will be blocked on the *receive(sync)* activity, while P_{S2} enables a_2 execution. In the same time, P_{S2} current instances number i is incremented. If i is less than N, another instance of a_2 is enabled. As a result, N instances of a_2 would run concurrently.
2) For each a_2 instance termination, a synchronization message is sent to P_{S3} and a new a_3 instance is enabled. This leads to the increment of P_{S3} local instances number i. If i is equal to N, then a_5 would be enabled even while other instances of the *multiple instances activities* are still running.
3) For each a_3 instance termination, a synchronization message is sent to P_{S4} and a new a_4 instance is enabled. Then, P_{S4} local instances number i is incremented.

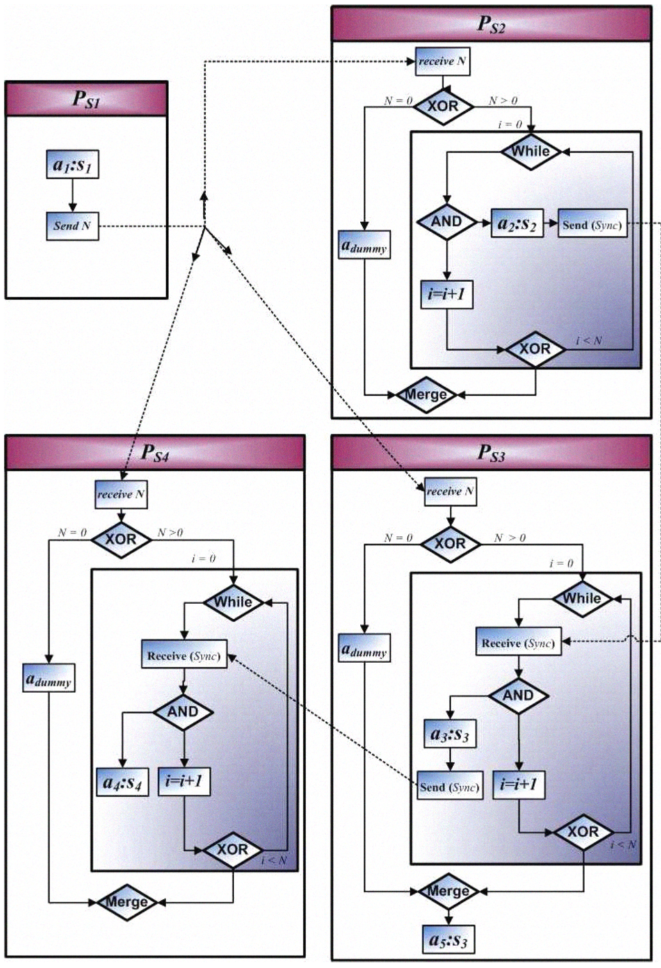

Fig. 6. Decentralized Multiple instances pattern

Our technique for decentralizing *multiple instances* pattern is also not specific to the introduced example. For instance, if we replace a_2, a_3 and a_4 by a given sub-process to run multiple instances of it. Then we partition the sub-process itself using our basic partitioning technique [10]. Next, we encapsulate the derived partitions into *multiple instances* patterns and add the corresponding synchronization activities.

In [25], the authors presented some centralized implementations to *synchronized multiple instances with a prior design or run time knowledge* patterns. With prior design knowledge, they just replicate the multiple instance activity with a *parallel split* pattern (AND_{split}). Once all activities instances are completed, they synchronize them with a *synchronizing* pattern (AND_{join}). This could be automatically decentralized using our basic algorithms. With a prior run time knowledge, they proposed many implementations such as using *Loops* to activate instances sequentially or using a combination of AND_{split} and XOR_{split}. The latter assumes a maximum number of possible instances. According to these implementations, we can use either our technique to partition *Loops* or our basic partitioning algorithms [10].

C. Discriminator decentralization

The *discriminator* (known also as 1-out-of-M join) is a point in a process model that waits for one of the incoming branches to complete before activating the subsequent activity [28]. From that moment, it waits for all remaining branches to complete and ignores them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again. For instance, When handling a cardiac arrest, the *check_breathing* and *check_pulse* tasks run in parallel. Once the first of these has completed, the *triage* task is commenced. Completion of the other task is ignored and does not result in a second instance of the *triage* task. It should be noted, that all branches must either flow from the Parallel Split to the Structured Discriminator without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

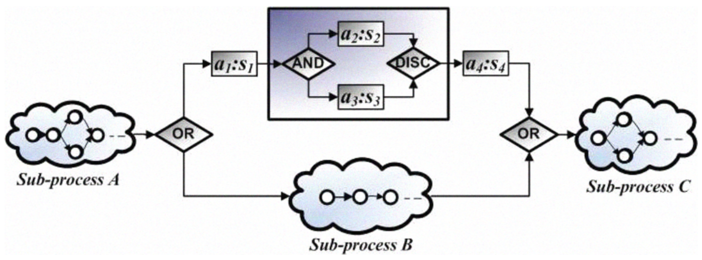

Fig. 7. Discriminator pattern

An process example implementing a *discriminator* pattern is depicted in Figure 7. The process is composed of four sub processes namely A, B, C and the sub-process in a box representing the *discriminator* pattern. Once a_1 terminates, it enables both a_2 and a_3 concurrently. Assuming a_2 terminates first, the *discriminator* fires and enables a_4, and possibly sub-process C. When a_3 terminates it would be ignored.

Assuming that the process example involves four services s_1, s_2, s_3 and s_4, then partitioning it would result in four partitions (see Figure 8). A_{si} (respectively B_{si}, C_{si}), represents the derived partition of the sub-process A (respectively B, C), related to the service s_i. After A_{s1} completion, it takes a decision about the branches to enable through the OR-$split$. The decision is forwarded to the other partitions having the same OR-$split$ using a pattern identifier [10] (this forwarding is not depicted in the example). Assume that the branch including a_1 is enabled and the one including B_{s1} is skipped. In this case, B_{s2}, B_{s3} and B_{s4} would also be skipped. Once a_1 terminates, a *one-to-many send* activity is launched and two synchronization messages are sent respectively to P_{s2} and P_{s3} enabling a_2 and a_3 concurrently. The first which terminates, send a synchronization message to P_{s4} and hence, a_4 would be enabled. The second would be ignored when it completes.

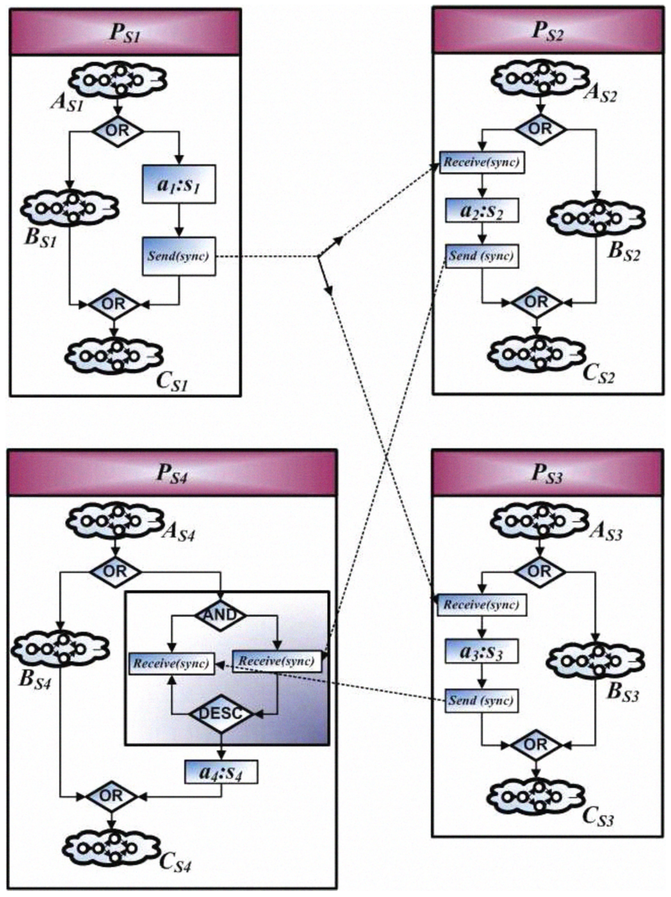

Fig. 8. Decentralized discriminator pattern

The proposed algorithm is generic and not specific to this example. It even can handle *N-out-of-M join* pattern (a generalization of the *discriminator*) [28], by communicating the number of branches to be enabled N to the *discriminator*. Due to lack of space, our formal algorithms for partitioning *loops*, *multiple instances* and *discriminator* patterns are not

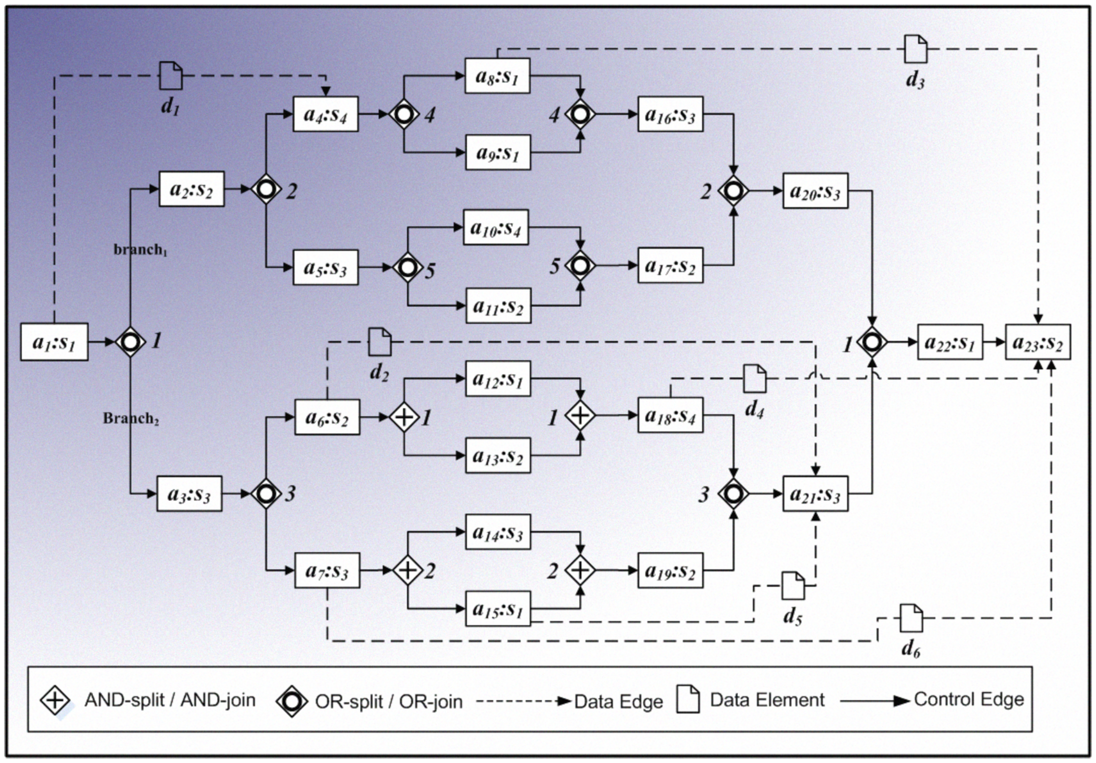

Fig. 9. Process example

presented in this paper.

V. Synchronization process

In this section, we detail how our interconnection mechanism works to synchronize between the derived partitions of a given centralized process, in run-time. First, let us consider the following process example depicted in Figure 9. The graphical description includes control and data dependencies of process activities. The process represents a collaboration between four web services namely s_1, s_2, s_3 and s_4. a_i:s_j represents an invocation activity of one of s_j operations. Each construct has an identifier such as a two corresponding constructs with the same type have the same identifier. For instance, the first $OR\text{-}split$ and the last $OR\text{-}join$ in the process example have the same identifier id =1. In this example, we use only basic patterns since we have already presented how to handle synchronization within advanced patterns in a decentralized architecture. Synchronization is achieved using *send*, *One-To-Many Send*, *Receive* and *One-from-many Receive* interaction patterns. Figure 10 depicts the derived partitions of the centralized process example as well as a part of the required inter-connections. Interconnection process is explained in [10]. In order to simplify the example, we have omitted interaction activities assuming that for each arrow connecting two partitions, there are *send* and *receive* activities respectively at the sender and receiver sides. Further, several arrows coming out from the same activity means a *One-To-Many Send*, while several arrows converging to the same activity means a *One-from-many Receive*.

A. Message exchange

Derived decentralized processes, communicate through message exchange using interaction patterns. Messages represent either control or data information. We define a message format as follow: Message ($type$, a_i:P_{Sm}, a_j:P_{Sn}, *instance-id*, $information$). Where $type$ specify either if it is control or data message (set to zero or one), a_i:P_{Sm} defines interaction activity and partition source, a_j:P_{Sn} the interaction activity and partition target, *instance-id* the instance identifier to make correlation, and $information$ is either a control decision or a data to transfer. A control decision has the form *enable*(*branch-ids*), where *branch-ids* are the set of branches to enable. For instance, in Figure 10, the message M(0, $send_1$:P_{S1}, $receive_1$:P_{S2},12, *enable*(b1)) represents a control decision taken by P_{S1} concerning $OR\text{-}split_1$. When received by P_{S2}, it enables $branch_1$ and therefore a_2. $branch_2$ and its subsequent activities will be automatically skipped.

B. Control dependency synchronization

In a centralized architecture, a decision for the $OR\text{-}split_1$ should be taken after a_1 termination, to either enable a_2, a_3 or both of them. In our decentralized architecture, the decision should be transmitted to all partitions having an $OR\text{-}split_1$ (represented by the three dashed arrows C_1 in Figure 10). Next, we explain through a running instance, how our synchronization technique minimizes the messages exchanges number compared to typical approaches. The main

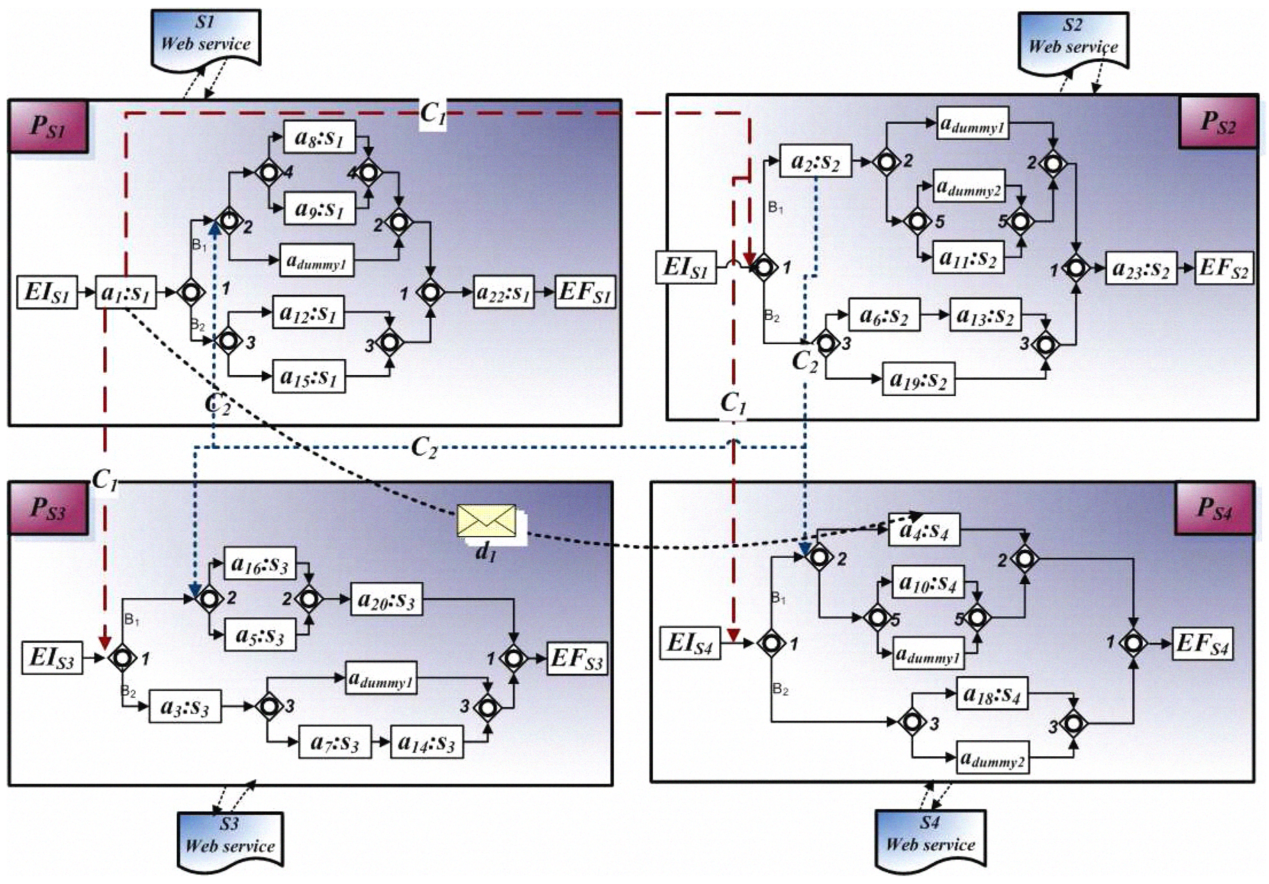

Fig. 10. Decentralized process

advantage lies in replicating patterns through the corresponding partitions. We remind that two corresponding join and split patterns are replicated only through partitions including activities between them in the centralized process. For instance, a_1 is connected to the decentralized patterns OR-$split_1$ instead of its *postset* $a_1\bullet$=$\{a_2$,$a_3\}$. To get a better understanding, assume a scenario where a_3 won't be enabled after a_1 execution.

1) Without patterns replication: First, a_1 send two messages to its postset $a_1\bullet$ to enable a_2 and skip a_3. Then a_3 send two messages to $a_3\bullet$= $\{a_6$,$a_7\}$ to be skipped, which in turn send messages to their postsets and so on until reaching a_{21}. This leads to 11 synchronization messages inter-partitions.
2) With patterns replication (Figure 10): After a_1 termination, three synchronization messages (C_1) are sent to P_{S2}, P_{S3} and P_{S4}. Messages contain a control decision taken by P_{S1} concerning OR-$split_1$ and enabling only $branch_1$ (a_2). When received, each partition would skip locally all subsequent activities of $branch_2$. Therefore, only three inter-partition messages are needed to synchronize this first part of the process execution. Now, assume that all subsequent activities of $branch_1$ are executed (until a_{20}). Thanks to our technique, P_{S1} knows in advance that it has to wait for only one control message (from a_{20}) to enable a_{22}, since it is preceded by OR-$join_1$. Hence, a_{21} doesn't need to inform a_{22} that it was skipped.

C. Data dependency synchronization

Data dependency characterizes the mapping relation of the output and the input values of two activities. We state two problems: sending data to an activity which won't be activated later and waiting for a data from an activity which won't be activated. In the first scenario, the data should be transmitted from the partition source to its destination. The message won't be consumed, but stored in the destination buffer until it expires. In the second scenario, we identified two solutions. A trivial one is that for each skipped activity which has a data dependency, a notification message should be sent by the partition it belongs to, to the activity waiting for the data. This may lead to an extra number of messages. So, we suggest to use *data routing tables DRT* for each activity followed by an OR-*split* or XOR-*split*. A *data routing table*, is a table resuming data dependencies between a set of activities. For instance, consider the process example and the case where a_3 won't be executed. Assuming that a_1 is the activity which takes the decision (OR-$split_1$), then P_{S1} should maintain a DRT_1. The latter includes the activities $\{a_8$:s_1, a_{18}:s_4, a_7:s_3, a_{23}:$s_2\}$ and the data variable names $\{d_3$, d_4, $d_6\}$. This means that only dependencies between activities inside the $< OR\text{-}split_1$, $OR\text{-}join_1 >$ and activities outside are considered. In this case, P_{S1} will send a single message to P_{S2} to notify it for the skip of d_4 and d_6. This mechanism, reduces considerably message exchanges between partitions. Also, no messages between skipped dependent activities inside $< OR\text{-}split_1$, $OR\text{-}join_1 >$ would be exchanged.

D. Process execution

During runtime, activities must respect all their preconditions and postconditions before and after execution. Preconditions are control and data connections with each of its preset activities whereas postconditions are control and data connections with each of its postset activities (i.e. in Figure 10, preconditions of a_4 are C_1, C_2 and d_1). For a given instance, a partition completes its role in the collaboration when it reaches EF activity. An instance execution terminates when all partitions reach EF for the same instance. The proposed technique for decentralization using patterns replication, make synchronization easier and minimizes message exchanges between partitions.

VI. Related Work

In recent years, several approaches and architectures for decentralized process execution have been proposed. In the context of process partitioning, [21][8] are the first works that take on the challenge of partitioning BPEL processes. These contributions use program partitioning techniques in order to reduce the communication costs between derived process fragments. It's not clear how they deal with propagating DPE across process fragments and what subset of BPEL they support.

[15][14] present a similar approach to the decentralization focusing on the P2P interactions. Their contribution take on the formalization of the decentralized interactions from a conceptual point of view. Nevertheless, they present some specific BPEL examples rather than an overall approach that can decentralize any sophisticated process. [22] presents a similar approach to the decentralization of the control flow without considering data dependencies of process activities. Similar process partitioning approaches have been applied to different needs such as the implementation of secure interactions[3] or the decentralized exception handling[7].

[19] developped a formal approach based on automata that takes as an input the existing services, the goal service, the costs and produces as an output a set of decentralized choreographers that realize the goal service using the existing services. The approach is based on *I/O-automata* representation of services and goal, and identifies appropriate choreography scheme using the notion of *universal service*, simulation relation and a path-cost computation for a graph. This work does not focus on the decomposition of global processes and it is not clear how they deal with *loops*, *discriminator* and *multiple instance* patterns.

In [32][30][31], the decentralization of processes has been studied in an abstract manner by extending the dead path elimination operation of workflow management systems. The decentralization focuses on the preservation of the centralized specification by preventing possible blocking situations.

Another approach to the decentralization concerns the implementation of additional applications to support the required interactions without embedding them into decentralized processes. ObjectFlow [11] uses a graph-based workflow definition model. Steps are executed by agents coordinated by a (potentially) distributed workflow engine which however accesses a centralized DBMS to store workflow states. In METEOR2 [23], process scheduling is distributed among various task managers. In Mentor [29], workflows are modeled using state-charts which are partitioned to each involved *processing entitiy* (PE). Each PE-specific state-chart is executed locally on the PE workstation. Another example that support the decentralized execution without partitioning centralized specifications is Self-Serv[6]. In Self-Serv, the interactions of composed services are implicitly encoded within the processes.

In the context of Web services, [13] introduce the Web Services Choreography Description Language which has not received much attention, similarly to [2]. The organization for the Advancement of Structured Information Standards puts forward the ebXML standard for business collaboration [12]. More recently, service interaction patterns have been introduced in [4], and the language *Let's Dance* was introduced in [33]. The relationship between a global public process choreography and the private orchestrations is investigated in [26] based on work on process inheritance as introduced in [5]. The equivalence of process models, using their observable behavior, is studied in [24]. The relationship between compatibility notions in process choreography and consistency of process implementations with regards to behavioral interfaces is studied in [9].

The developed techniques are often good for dealing with a particular aspect of decentralization rather than providing a generic and flexible manipulation setting required for process decentralization. The common limitation of the current decentralization approaches is their dependencies on the underlying process specification. They can deal with how the decentralized processes must be synchronized with the relevant messages of the low-level specification but they cannot address the fundamental questions about the decentralization of the combined control and data dependencies. Consequently, this becomes a major limitation for the use of these systems in different cases which are not explicitly specified in their decentralization mechanism. Also, most of the proposed approaches stop short in answering to how they handle *loops*, *multiple instances* and *discriminator* patterns in decentralized processes. The main advantage of the developed approach is the flexibility that it provides in terms of concepts and structures that it manipulates. This, in turn, allows the extension of the algorithms to the different needs of decentral-

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8275
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8275

ization. In sharp contrast to previous works, our operation of decentralization computes the abstract process constructs, *i.e.*, workflow patterns[25]. This methodology separates the implementation details from the high-level reasoning that provides a more complete and generic solution to the decentralization problem.

VII. Conclusion

This paper has presented an extension to our approach to the flexible decentralization of process specifications. The developed approach is applicable to a wide variety of service composition standards that follow the process management approach such as WS-BPEL. In contrast to previous works that take on the process decentralization approaches, our methodology separates the implementation details from the high-level reasoning that provides a more complete and generic solution. In addition to the basic constructs like *XOR*, *AND*, *OR* (split and join), sequence... It also takes into consideration advanced patterns such as *Loops*, *Multiple instances* and *discriminator*. The flexibility introduced by decentralized processes on the other hand raises new requirements like synchronization between them. In this sense, we proposed a mechanism toward synchronization through message exchange using interaction patterns. Further, we would like to implement the introduced methodology on a web service composition language to enable a quantitative evaluation of the approach in terms of message exchanges, and add security aspects between the decentralized process specifications.

References

[1] Workflow management coalition: process denition interchange v 1.0 nal. http://www.wfmc.org, 1998.

[2] S. F. Arkin, Sid Askary and W. Jekeli. Web service choreography interface (wsci) 1.0,, 2002.

[3] V. Atluri, S. A. Chun, R. Mukkamala, and P. Mazzoleni. A decentralized execution model for inter-organizational workflows. *Distributed and Parallel Databases*, 22(1):55–83, 2007.

[4] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service interaction patterns. In *Business Process Management*, pages 302–318, 2005.

[5] T. Basten and W. M. P. van der Aalst. Inheritance of behavior. *J. Log. Algebr. Program.*, 47(2):47–145, 2001.

[6] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[7] G. Chafle, S. Chandra, P. Kankar, and V. Mann. Handling faults in decentralized orchestration of composite web services. In *ICSOC*, pages 410–423, 2005.

[8] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW (Alternate Track Papers & Posters)*, pages 134–143, 2004.

[9] G. Decker and M. Weske. Behavioral consistency for b2b process integration. In *CAiSE*, pages 81–95, 2007.

[10] W. Fdhila, U. Yildiz, and C. Godart. A flexible approach for automatic process decentralization using dependency tables. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*, pages 847–855, Los Angeles, CA, USA, 2009. IEEE Computer Society.

[11] M. Hsu and C. Kleissner. Objectflow: Towards a process management infrastructure. *Distributed and Parallel Databases*, 4(2):169–194, 1996.

[12] S. S. A. Jean-Jacques Dubray and M. J. Martin. *ebXML Business Process Specification Schema Technical Specification v2.0.4. OASIS.* 2006.

[13] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10, 2004.

[14] R. Khalaf, O. Kopp, and F. Leymann. Maintaining data dependencies across bpel process fragments. *Int. J. Cooperative Inf. Syst.*, 17(3):259–282, 2008.

[15] R. Khalaf and F. Leymann. E role-based decomposition of business processes using bpel. In *ICWS*, pages 770–780, 2006.

[16] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *CAiSE*, pages 431–445, 2000.

[17] F. Leymann and D. Roller. *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, 2000.

[18] R. Liu and A. Kumar. An analysis and taxonomy of unstructured workflows. In *Business Process Management*, pages 268–284, 2005.

[19] S. Mitra, R. Kumar, and S. Basu. Optimum decentralized choreography for web services composition. In *IEEE SCC (2)*, pages 395–402, 2008.

[20] F. Montagut, R. Molva, and S. T. Golega. The pervasive workflow: A decentralized workflow system supporting long-running transactions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 38(3):319–333, 2008.

[21] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA*, pages 170–187, 2004.

[22] W. Sadiq, S. W. Sadiq, and K. Schulz. Model driven distribution of collaborative business processes. In *IEEE SCC*, pages 281–284, 2006.

[23] A. P. Sheth, K. Kochut, J. A. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting state-wide immunisation tracking using multi-paradigm workflow technology. In *VLDB*, pages 263–273, 1996.

[24] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In *Business Process Management*, pages 129–144, 2006.

[25] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[26] W. M. P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. In *CAiSE*, pages 140–156, 2001.

[27] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. PRS Prentice Hall, 2005.

[28] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Verlag, first edition, November 2007.

[29] D. Wodtke, J. Weißenfels, G. Weikum, and A. K. Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In *ICDE*, pages 556–565, 1996.

[30] U. Yildiz and C. Godart. Centralized versus decentralized conversation-based orchestrations. In *CEC/EEE*, pages 289–296, 2007.

[31] U. Yildiz and C. Godart. Synchronization solutions for decentralized service orchestrations. In *ICIW*, page 39, 2007.

[32] U. Yildiz and C. Godart. Towards decentralized service orchestrations. In *Proceedings of the 2007 ACM Symposium on Applied Computing , SAC*, pages 1662–1666, 2007.

[33] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's dance: A language for service behavior modeling. In *OTM Conferences (1)*, pages 145–162, 2006.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8275
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8275