A Attached Papers

A.1 The Kaleidoscope of Fragmentation of Service Compositions

Authors:

Tilburg: Michele Mancioppi

USTUTT: Olha Danylevych

USTUTT: Dimka Karastoyanova

USTUTT: Frank Leymann

Tilburg: Mike P. Papazoglou

The Kaleidoscope of Fragmentation of Service Compositions[*]

Michele Mancioppi[1], Olha Danylevych[2],
Dimka Karastoyanova[2], Frank Leymann[2], and Mike P. Papazoglou[1]

[1] European Research Institute in Services Science (ERISS),
Tilburg University, The Netherlands
{m.mancioppi,mikep}@uvt.nl
[2] Institute of Architecture of Application Systems,
University of Stuttgart, Stuttgart, Germany
{olha.danylevych,dimka.karastoyanova,leymann}@iaas.uni-stuttgart.de

Abstract. Service compositions enable the creation of value added services by reusing existing ones. All the foremost modeling languages for service composition are process-based, e.g. BPEL, BPMN and WS-CDL. The fragmentation of a process-based service composition is the act of dividing its elements (activities, data flows, control flows, message flows, etc.) into fragments according to some criterion.
Fragmentation techniques greatly differ in which types of process-based service compositions they are applicable to, why they are applied, how they define the fragments, etc. The state of the art lacks consistent terminology and definitions for the properties of the fragments of process-based service compositions and the criteria for classifying the different fragmentation techniques. This paper tackles this issue by investigating classification criteria for fragments and fragmentation techniques based on the "seven Ws" (why, what, when, where, who, which, and how), which are then applied to some of the existing fragmentation approaches.

Key words: SOA, fragmentation, fragments, service composition

1 Introduction

By enabling the creation of novel, value added services through the reuse of existing ones, service compositions are one of the cornerstones of Service Oriented Architecture (SOA). The predominant paradigms of service composition are service orchestration and service choreography (respectively orchestration and choreography in short) [1]. Orchestrations are centralized. One service – the orchestrator – executes the logic of the composition by invoking the other services and aggregating their results. Choreographies are instead completely decentralized. The participants in a choreography play their pre-defined roles according to how proceeds the choreography enactment, i.e. the overall made of the executions of the single participants.

[*]The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483.

The majority of modeling languages for specifying service composition are *processes-based*. A process-based service composition is specified as tasks (e.g. invocation of services, manipulation of data), executed by actors, and sequenced using constructs like control-, message- and data flows. Examples of process-based modeling languages for service compositions are Business Process Execution Language (BPEL) [2] and Business Process Modeling Notation (BPMN) [3] for orchestrations, and Web Service Choreography Description Language (WS-CDL) and BPEL4Chor [4] for choreographies. Alternatives to process-based service compositions are the declarative ones, e.g. orchestrations modeled with DecSerFlow [5], and the rule-based one, e.g. [6]. This work focuses on process-based service compositions, called service compositions in the remainder for brevity.

Service compositions must be changed over time to suit evolving requirements and environments. One broad category of changes applied to service compositions hinges on their *fragmentation*, i.e. creating *fragments* that group the service composition's elements (activities, control flows, data flows, etc.) according to criteria like "each fragment contains all the activities performed by the same actor". Fragmentation of service compositions is instrumental for accomplishing a variety of goals including, for example, enabling distributed execution of service orchestration by dividing them in parts that are executed separately [7], performing abstraction of models by removing their less significant elements [8], and facilitate the reuse of existing fragments [9, 10].

Unfortunately the state of the art lacks consistent terminology and definitions of what fragmentation and fragments of service composition are, and this hinders the comparison, analysis and reuse of the available fragmentation techniques. Our work tackles this issue by providing classification criteria for service composition fragments and fragmentation techniques based on the seven "Ws": why, what, when, where, who, which, and how. The classification criteria are then applied to two of the fragmentation techniques for service compositions available in the literature, namely [11, 7].

The paper is structured as follows. Section 2 establishes the basic terminology and definitions that are used throughout the work. Section 3 treats the criteria for the classification of fragmentation techniques for service compositions, the application of which is exemplified in Section 4. Section 5 discusses the related work. Finally Section 6 concludes the paper by presenting our final remarks and directions for future work.

2 Definitions

This section introduces the terminology that will be adopted in the remainder of the paper.

The **service composition paradigm** (**composition paradigm** in short) is the "type" of service composition, e.g. service orchestration or service choreography. The **abstraction level** of a service composition can be either **model** or **instance**. The model (also called "type" in the related work on Process-Aware Information Systems [9]) is the specification of the structure of the service com-

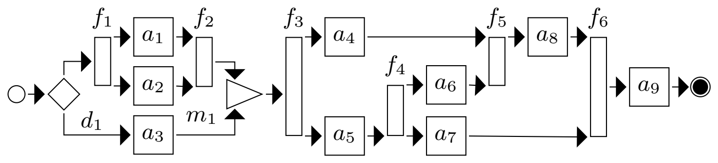


Fig. 1: The running example

position, i.e. a workflow model. An instance is the execution of a model, and it can be formalized as a "copy" of the executed model coupled with the **history**, i.e. the sequence and timestamps of actions (execute activity, traverse control flows, etc) that have been executed so far [9].

A **modeling language**, e.g. BPEL or BPMN, specifies the syntax of service composition models and the semantics for the execution of their instances. Modeling languages provide a number of **constructs** that can be combined in the models. For example, BPEL specifies several types of activities, event handlers, compensation handlers, etc. The constructs are instantiated as **elements** that compose the model. For example, the activity Invoke Amazon WS is an element of the service orchestration model Buy Books On Amazon. In other words, a service composition model can be seen as an aggregation of elements instantiated using the constructs provided by the adopted modeling language.

A subset of the elements of a service composition is called **fragment**. No assumptions are made in the remainder about the elements that make up a fragment and how they relate to each other, except that they are at least one (i.e. fragments can not be empty). **Fragmentation** is the act of creating fragments out of one service composition by applying a **fragmentation technique**. A fragmentation technique is a method to perform fragmentation according to some **fragmentation criteria**, i.e. the rationale underpinning the fragmentation technique. The fragmentation criteria may be described in natural language, e.g. "the resulting fragments group the activities according to who executes them", or formally, for example using Category Theory. Fragmentation technique combine the following two steps:

Fragment Identification identifies which elements belong to which fragment;
Fragment Severing removes the elements comprised in a fragment from the service composition, possibly substituting them in the service composition with other elements that were not initially comprised herein.

Figure 1 depicts the running example of this paper: a process made of start and end nodes, activities, fork and merge constructs, adapted from [11]. Figure 2 exemplifies the Fragment Identification and Fragment Severing phases of a fragmentation of the running example. The Fragment Identification results in the two fragments F_1 and F_2, delimited by the dashed lines. The fragmentation criteria used to identify the two fragment are not relevant to the exposition, and thus omitted. The Fragment Severing is divided in two steps. Step 1 divides the two fragments by removing the control flow connecting m_1 and f_3. Step 2 adds an end node and a control flow to the fragment F_1, and a start node and a control flow to the fragment F_2.

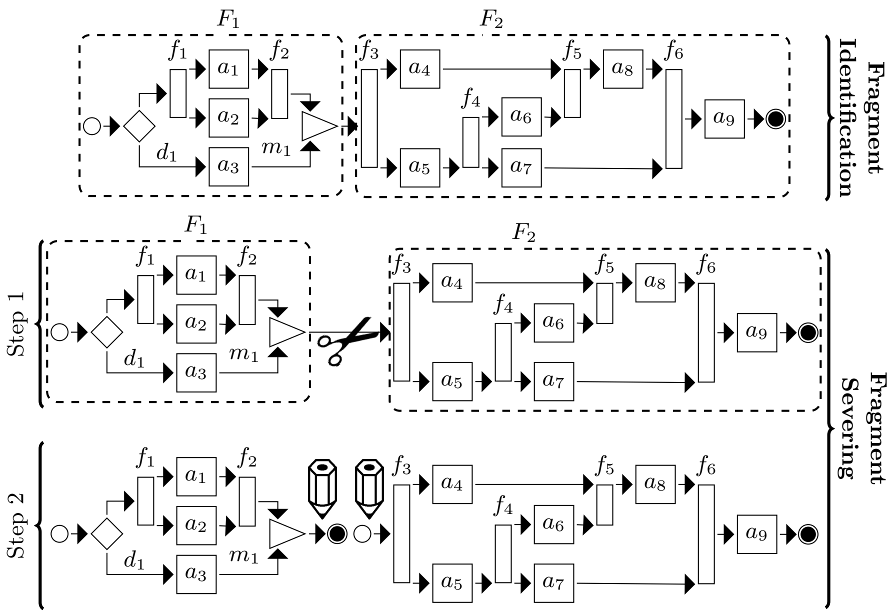


Fig. 2: Sample Fragment Identification and Severing of the running example

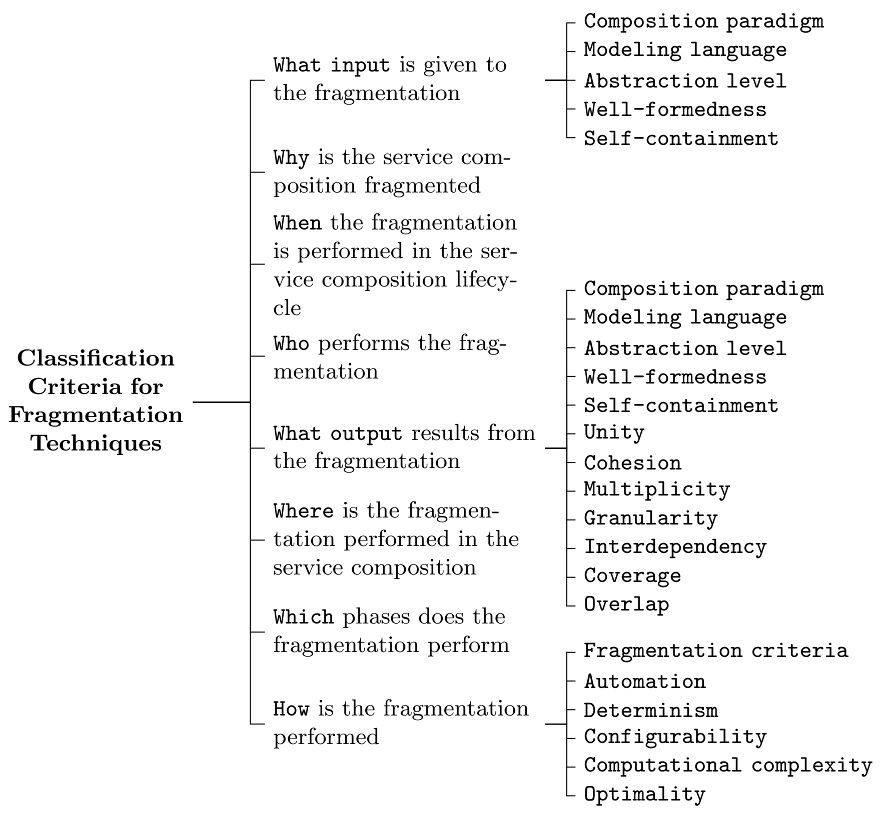
Not every fragmentation technique performs both Fragment Identification and Fragment Severing. For example, Single-Entry Single-Exit (SESE) decomposition performs only Fragment Identification, identifying the fragments without actually applying any modifications to the service composition. Conversely, the fragmentation technique for BPEL presented in [7] assumes that the fragments have been previously identified (i.e. no Fragment Identification), and performs the Fragment Severing so that the fragments can be executed in a distributed manner while preserving the semantics of the original orchestration.

It should be noted that Fragment Identification and Severing are not necessarily executed only once during a fragmentation, or in this order. A fragmentation technique may perform several Fragment Identification phases in a row, followed by a number of Fragment Severing ones, or any other appropriate combination. The ordering of the Fragment Identification and Severing phases in a fragmentation technique is called **fragmentation lifecycle**.

3 Classification Criteria for Fragmentation Techniques

This section presents the classification criteria we identify for fragmentation techniques for process-based service compositions. They are formulated on the basis of an extensive literature review. Each criterion is presented as one of the following questions, each based on one of the seven Ws:

- `What input` is given to the fragmentation?
- `Why` is the service composition fragmented?



Classification Criteria for Fragmentation Techniques
What input is given to the fragmentation
Composition paradigm
Modeling language
Abstraction level
Well-formedness
Self-containment
Why is the service composition fragmented
When the fragmentation is performed in the service composition lifecycle
Who performs the fragmentation
What output results from the fragmentation
Composition paradigm
Modeling language
Abstraction level
Well-formedness
Self-containment
Unity
Cohesion
Multiplicity
Granularity
Interdependency
Coverage
Overlap
Where is the fragmentation performed in the service composition
Which phases does the fragmentation perform
How is the fragmentation performed
Fragmentation criteria
Automation
Determinism
Configurability
Computational complexity
Optimality

Fig. 3: The classification criteria for fragmentation techniques

– `When` the fragmentation is performed in the service composition lifecycle?
– `Who` performs the fragmentation?
– `Where` is the fragmentation performed in the service composition?
– `What output` result from the fragmentation?
– `Which` phases does the fragmentation perform?
– `How` is the fragmentation performed?

The overall list of the criteria and sub-criteria is presented in Figure 3. The criteria `What input`, `What output` and `How` are further refined into sub-criteria, e.g. the `How` criterion is divided into `Automation`, `Determinism`, etc.

Each criterion is investigated specifically in the remainder. For brevity and generality we do not provide formal definitions based on a particular modeling language, but the criteria are easily applicable to any given one.

3.1 What input is given to the fragmentation

The `What input` criterion investigates the characteristics of the service compositions in input to the fragmentation technique. The several interesting aspects of the input service compositions are captured by the sub-criteria presented in the remainder of the section.

The criterion `Composition paradigm` specifies the paradigm of service composition (e.g. service orchestration or service choreography, see Section 2).

`Modeling language` denotes which modeling language is used to specify the service compositions (see Section 2).

The `Abstraction level` specifies whether fragmented service compositions is a model or an instance (see Section 2).

`Well-formedness` reports whether the service composition is required to be well-formed or not. The particular definition of well-formedness depends on the adopted modeling language. Generally it implies that the service composition satisfies all the syntactical constraints set by its modeling language.

`Self-containment` is when the service composition is sufficient in itself with respect to its intended use. For example, an executable business process is self-contained if and only if it contains all the data necessary for its execution.

3.2 Why is the service composition fragmented

The `Why` criterion specifies the goals that motivate the fragmentation of the service composition. An exhaustive list of the motivations for fragmenting service compositions is beyond the scope of this work, but some examples are:

Support the modeling: the fragmentation technique supports the modeling of service compositions by, for example, enabling autocompletion during the modeling [12] and facilitating the reuse of existing fragments [10].

Analysis: the fragmentation technique can be used to analyze the control- and data flows in the service composition. For example, fragmentation of workflows can be used to prove their soundness [11].

Abstraction: the fragmentation technique is used to create an abstract, simplified view of a service composition by identifying and removing some of its elements that are not central to its logic [8].

Optimize non-functional characteristics: the service composition is fragmented so that its fragments can be executed in a distributed fashion achieving better Quality of Service (QoS) [13], e.g. in terms of completion time and throughput.

3.3 When the fragmentation is performed in the service composition lifecycle

The `When` criterion regards in which phases of the service composition lifecycle the fragmentation takes place, e.g. Design or Execution. The particular lifecycle to be considered depends on the fragmented service composition.

When the fragmentation takes place in the lifecycle is closely related with the `Abstraction level` aspect of the criterion `What input` (see Section 3.1). The fragmentations of service compositions at different abstraction levels occur at different phases of the lifecycle. For example, consider the lifecycle for service orchestrations proposed in [14], made of Design/Verification, Deployment, Execution/Monitoring and Evaluation. The fragmentation of a model can take place, for example, in the Design/Verification and Deployment phases. Instead, the fragmentation of an instance must take place during the Execution/Monitoring and Evaluation phases.

3.4 Who performs the fragmentation

The fragmentation of a service composition may be performed by different actors. The value of the `Who` criterion is not necessarily a human, but it might also be a system (e.g. an application server or a development tool) in case of automatic fragmentation techniques (see the `Automation` sub-criterion of `How` in Section 3.8).

The `Who` criterion is very connected with `When` (see Section 3.3) because the available alternatives for the `Who` criterion are the actors involved in the service composition lifecycle phases in which the fragmentation can be performed (e.g. the business process designer at Design time).

3.5 What output result from the fragmentation

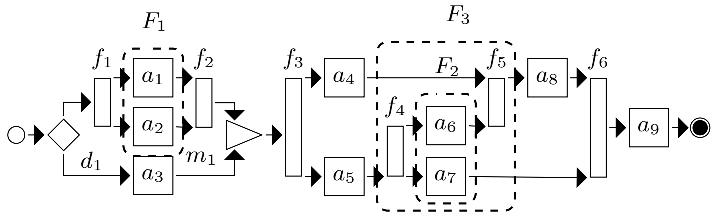
The outcome of a fragmentation is a number of fragments. Not all fragments are the same. Their properties greatly vary across the different fragmentation techniques. This section outlines the criteria to classify the properties of fragments, which are exemplified using the fragmentations presented in Figure 4 (the fragments are delimited by the dashed lines). The examples have been crafted for exposition purposes and do not reflect the application of any particular fragmentation technique.

The criteria that characterize the properties of the fragments can be grouped according to the type of the property they capture, namely: (1) properties of the single fragments, and (2) properties of all the fragments cumulatively. The two groups of criteria are discussed separately in the remainder.

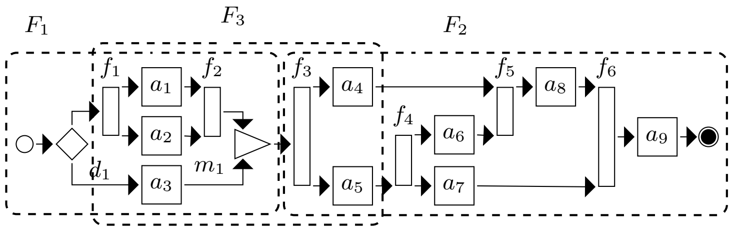
Some of the criteria to classify the fragments are duplicated from the sub-criteria of `What input`, namely `Composition paradigm`, `Modeling language`, `Abstraction level`, `Well-formedness`, and `Self-containment`. The semantics of the sub-attributes under `What output` is exactly the same as their homonyms' under `What input`, except that they apply to the fragments instead of the service composition. Their description is here omitted for reasons of space. With respect to the division in properties of single fragments or of all fragments cumulatively, criteria fall in the first category.

Criteria based on properties of the single fragments: These criteria for classifying fragments regard their properties of each one considered separately.

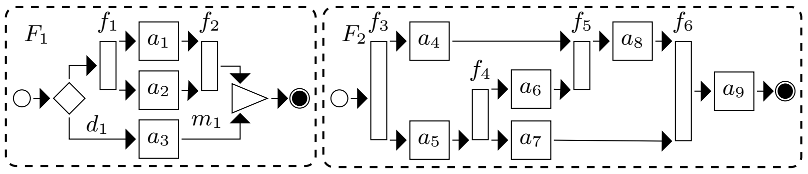
Generally, unity is defined as "the state of being united or joined as a whole" [15]. In the case of a fragment, `Unity` specifies whether it is still "physically" comprised in the service composition. `Unity` is *preserved* if the fragment is an "high-lighted area" of the service composition (e.g. Figure 4a, Figure 4b and Figure 4d). Otherwise, `Unity` is *disrupted* in fragmentation techniques that



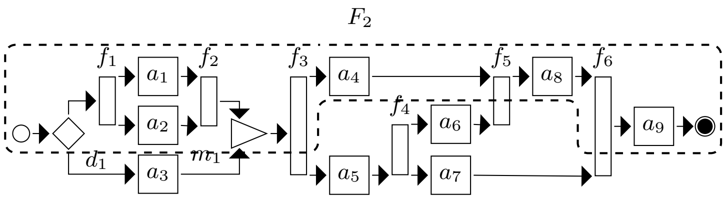
(a) Fragmentation example 1



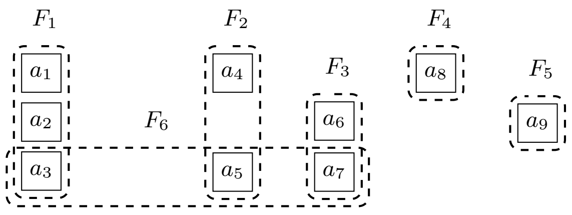
(b) Fragmentation example 2



(c) Fragmentation example 3



(d) Fragmentation example 4



(e) Fragmentation example 5

Fig. 4: Possible fragmentations of the running example in Figure 1

create fragments that are independent from the original service composition, e.g. if the fragments themselves are service composition models or instances (for example Figure 4c). Fragmentation techniques that perform only Fragment Identification (see Section 2 and Section 3.6) preserve `Unity`. In fact, `Unity` can only be disrupted through Fragment Severing.

`Cohesion` is a measure of how much the elements in the fragment "belong together" [16]. Cohesion is a well-known concept in the Software field, and has recently been under research scrutiny in relation with business processes, e.g. [16]. The cohesion of the different fragments is evaluated quantitatively, and it may vary across the different fragments produced in one fragmentation. In this case, classifications may represent the average of the cohesion.

Criteria based on properties of all the fragments cumulatively: The following criteria focus on properties that arise from comparing the fragments with each other and with the service composition.

`Multiplicity` specifies how many fragments result from the fragmentation. For example, the values of the criterion `Multiplicity` may be specified as multiplicities in regular grammars (0, 1, more than 1, or a specified number n) or ranges, e.g. between m and n, extremes included.

Not all fragments have the same size. The `Granularity` of the fragments is quantitatively expressed as their size relative to the service composition's. Should the granularity of the different fragments vary, its average and standard deviation could be used to provide an overview of the overall granularity of the fragmentation.

`Interdependency` defines which kind of dependencies (if any) exist among the fragments. Different fragmentation techniques are likely to define different types of dependencies, e.g. control- and data dependencies [17] and coordination protocols [7] in the case of fragmentation techniques for enabling distributed execution of the service compositions.

The extent to which the fragments cumulatively comprise all the elements of the original service composition is treated by the `Coverage` criterion. *Full* coverage is achieved if every element of the service composition is contained in at least one of the fragments, and no fragment contains elements not originally present in the service composition (e.g. Figure 4b). *Partial* coverage occurs when the union of the fragments contains a proper subset of the elements of the service composition (e.g. Figure 4a, Figure 4d and Figure 4e). The coverage is *extended* if the fragments collectively contain every element in the service composition, plus others that were not part of the service composition. An example of extended coverage has been presented in Section 2, where nodes and control flows were added during the Fragment Severing to fragments whose union comprises all the elements. Finally, the coverage is *partial-extended* if the fragments do not collectively contain all the elements of the service composition, but others have been added that are not part of the service composition. Note that "null" coverage is not possible: it would mean that none of the fragments contains any of the elements of the service composition, which is inconsistent with the definition of fragment (assumed not to be empty) given in Section 2.

Finally, the criterion `Overlap` denotes the extent to which the fragments share elements with each other. The alternatives are defined as pair-wise relations between fragments. *Full overlap* is when the two fragments comprise exactly the same elements. Since fragments are defined as sets of elements, *full overlap* is an identity relation. *Nesting* is the case in which one fragment is a proper subset of another, e.g. F_2 and F_3 in Figure 4a. *Partial overlap* is when two fragments have in common at least one element, but neither fragment is a subset (proper or improper) of the other, for example F_1 and F_3, and F_2 and F_3 in Figure 4b. Finally, fragments that do not share any element are *disjoint*, for example F_1 and F_2 in Figure 4c. The classification of a fragmentation technique should list all the cases that may occur in fragmentations. If all the alternatives listed above are possible, the classification may report *any* for brevity.

3.6 Which phases does the fragmentation perform

The `Which` criterion specifies the fragmentation lifecycle (see Section 2). It is outside the scope of our work to indicate how to specify fragmentation lifecycles, but the general consensus appears to be on state diagrams.

3.7 Where is the fragmentation performed in the service composition

The `Where` criterion treats the *scope* of the fragmentation, i.e. the regions of the service composition that are processed during the fragmentation. When the fragmentation takes into account the entirety of the service composition, its scope is *global*, and *partial* otherwise.

`Coverage` and `Where` and related. On one hand, not all the elements processed during a fragmentation end up in some fragments (i.e. global scope does not imply full coverage). On the other hand, an element that is not processed cannot be included in any fragment.

3.8 How is the fragmentation performed

The `How` criterion classifies the fragmentation techniques according to aspects that pertain to the algorithms they employ.

`Fragmentation criteria` describes the rationale underpinning the fragmentation technique (see Section 2). Fragmentation techniques that do not perform fragment identification may leave the fragmentation criteria unspecified.

`Automation` refers to whether the fragmentation technique is manual, automatic, or semiautomatic.

`Determinism` reports whether the fragmentation technique always produces the same fragments as output given the same service composition in input.

`Configurability` is the capability of the fragmentation technique to be customized in terms of scope, properties of the resulting fragments (e.g. upper- and lower bounds for granularity), etc.

`Computational Complexity` is the evaluation of the complexity of the fragmentation technique according to the size of the input (i.e. the number of elements in the service composition). An example of computational complexity is $O(n)$, i.e. the fragmentation requires at most linear amount of time with respect to n, i.e. the size of the input. Alternatively, complexity classes may also be used as values, e.g. `P`, `NP` or `EXP-TIME`. The `Computational Complexity` can be seen as a measure of the scalability of the fragmentation technique, defined for example as "the ability to accommodate increasing input, to process growing volumes of work gracefully, and/or to be susceptible to enlargement" [18].

`Optimality` specifies whether the fragmentation technique always results in the optimal fragmentation (for some definition of optimum that is specific to the technique) or in a sub-optimal one. The particular definition of optimum may depend on several factors, like the intended use of the fragments and the type of service composition, and it is situational to the goal of the fragmentation (and thus related to the criterion `Why`, see Section 3.2). For example, our previous work [17] investigates a fragmentation technique that is built on metrics for achieving optimality of, among others, concurrency and message overhead during the the distributed execution of the fragments.

4 Applying the Classification Criteria

This section exemplifies the application of the classification criteria defined in Section 3 to two of the fragmentation approaches in the literature, namely [7, 11]. The approaches have been selected for their heterogeneity and the completeness of the information that is available on them. The outcome of the classification is presented in Table 1.

Fragmentation Technique 1: Khalaf [7] proposes a fragmentation technique for BPEL models to enable their distributed execution while preserving the original semantics. The fragments are assumed to have been previously identified (i.e. no Fragment Identification), and the approach focuses on the Fragment Severing.

The resulting fragments are self-contained, well-formed BPEL processes that communicate with each other over message exchanges to realize the control- and data-flow dependencies between the activities in original process. There is no overlap between the fragments. The coverage is partial-extended: on one hand, elements are added to the fragments to make them well-formed and self-contained. On the other hand, the loops and scopes of the original BPEL process whose activities are spread across multiple fragments are removed, and substituted by coordination protocols.

Criterion	**Technique 1**	**Technique 2**
`What input`		
`Composition paradigm`	Service orchestration	Service orchestration
`Modeling language`	BPEL	Workflow Graph
`Abstraction level`	Model	Model
`Well-formedness`	Yes	Yes
`Self-containment`	Yes	Yes
`Why`	Distributed execution	Analysis
`When`	Deployment or Execution time	Modeling time
`Who`	Human process designer with tool support or a distributed process engine	Analysis tool
`What output`		
`Composition paradigm`	Service orchestration	Service orchestration
`Modeling language`	BPEL	SESE workflow fragments
`Abstraction level`	Model	Model
`Well-formedness`	Yes	Yes
`Self-containment`	Yes	No
`Unity`	No	Yes
`Cohesion`	Unspecified	Unspecified
`Multiplicity`	2+	1+
`Granularity`	Variable	Variable
`Interdependency`	Message exchanges, coordination protocols	Process Structure Tree
`Coverage`	Full	Partial
`Overlap`	Disjoint	Nested or Disjoint
`Where`	Global	Global
`Which`	Fragment Severing	Fragment Identification
`How`		
`Fragmentation criteria`	Unspecified	Group adjacent elements in fragments with one single entry and one single exit
`Automation`	Yes	Yes
`Determinism`	Yes	Yes
`Configurability`	No	Fragment granularity
`Computational complexity`	Unspecified	Linear to the input's size
`Optimality`	Unspecified	Unspecified

Table 1: Comparison of the fragmentation techniques presented in Section 4

Fragmentation Technique 2: Vanhatalo et al. [11] propose a fragmentation technique to verify soundness of a business process model specified as a workflow. The business process model is sound if it does not contain deadlocks or lacks of synchronization. The fragmentation technique is deterministic and has time complexity linear with respect to the size of the model (i.e. the number of its elements).

The approach performs only Fragment Identification. In fact, the fragments remain parts of the original process model, and thus unity is preserved. Each resulting fragment has only one inbound and outgoing edge, i.e. control flows respectively "entering" and "exiting" the fragment (hence SESE, for "single-entry, single-exit"). Some of the disjoint fragments are connected through control-flow constructs (thus interdependent). The fragments are well-formed, but are not self-contained (for example, none of them contains start or end nodes). The coverage is partial, because start- and end node are never included in any fragment (though all the other elements are). The fragment vary in granularity, and their overlap is either nested or disjoint.

The fragments are organized in the Process Structure Tree (PST), i.e. a tree in which the fragments are nodes, and the "child of" relation is nesting (a fragment is child of another in the tree is the first is nested into the latter in the model). Parent and child fragments have dependencies because of the control flows connecting them, because the control flows that enter and exit the child fragment are part belong the parent. Therefore, the PST effectively plots the dependencies between fragments.

5 Related Work

The nature of fragments, which we cover in the sub-criteria of `What output`, has recently been investigated by Eberle et al. [19]. Their work suggests the `Unity` property of the fragments, and it focuses on the local knowledge represented by a fragment, suggesting that the fragments should be defined according to whom performs their activities (which would be a fragmentation criterion according to our terminology).

Fragmentation of service composition is akin to the practice of Software Maintenance, to which the 7 Ws have been applied different extents. Why Software Maintenance is performed has lead to the taxonomy of its different types by Chapin et al. [20]. Buckley et al. [21] instead investigate the when, where, what and how of Software Maintenance. Our work has several points of contact with [21], in particular with respect to the "Time of change" (our `When`), the "Artifact" (our `What processes`), and the "Granularity" (our `Where`).

Mens et al. [22] propose a taxonomies of model transformation in the scope of Model-Driven Architecture (MDA). Given the fact that fragmentations can be realized through transformations, it should not come as a surprise that our works share a number of similarities. They define what is in input to and what in output from a transformation (though in less detail than our `What processes` and `What output` criteria), why is a transformation applied, some of the criteria we list under `How` (namely `Automation` and `Computational complexity`), and which mechanisms can be used for model transformation (e.g. graph transformations and functional programming) (which can be seen as a combination of our `Determinism` and `Configurability`). Additionally, they investigate the success criteria (e.g. testability, verifiability, traceability and change propagation) and quality criteria (e.g. usability, usefulness, scalability and formality) for transformation languages or tools. We have not studied yet these aspects of fragmentation approaches, but the results of [22] appear to be readily portable to our domain.

6 Conclusions

Fragmentation techniques are important tools for changing service compositions in response to evolving requirements. However, the lack of a consistent taxonomy for classifying the different fragmentation techniques and the properties of the fragments they produce has hindered their comparison and reuse.

At the best of our knowledge, this work presents the first attempt to organically investigate the characteristics of fragmentation techniques for service compositions and the properties of the resulting fragments. We have advanced classification criteria based on the 7 Ws (why, what, when, where, who, which, and how). They provide (1) a basis for classifying existing fragmentation techniques, and (2) a "checklist" of what authors should explicitly specify about the novel fragmentation approaches they introduce. We have also shown the application the classification criteria to two the fragmentation techniques in the state of the art.

The future work foresees an investigation of the precise definitions for the different types of fragments (part, slice, partition, piece, segment, etc.) on the basis of the classification criteria we have provided. Moreover, we would like to perform a more extensive classification of the available fragmentation techniques in the state of the art, which would facilitate the identification of research gaps and possibilities. Finally, we are investigating how to adapt our criteria for the merge of service compositions and their fragments.

References

1. Peltz, C.: Web services orchestration and choreography. IEEE Computer **36**(10) (2003) 46–52
2. OASIS: Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS (April 2007)
3. OMG: Business Process Modeling Notation Version 1.2. OMG Recommendation, OMG (February 2008)
4. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS, IEEE Computer Society (2007) 296–303
5. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: WS-FM. Volume 4184 of Lecture Notes in Computer Science., Springer (2006) 1–23

6. Pu, K.Q., Hristidis, V., Koudas, N.: Syntactic rule based approach to web service composition. In Liu, L., Reuter, A., Whang, K.Y., Zhang, J., eds.: ICDE, IEEE Computer Society (2006) 31
7. Khalaf, R.: Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective. Doctoral thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (March 2008)
8. Polyvyanyy, A., Smirnov, S., Weske, M.: Process model abstraction: A slider approach. In: EDOC, IEEE Computer Society (2008) 325–331
9. Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: CAiSE. Volume 4495 of Lecture Notes in Computer Science., Springer (2007) 574–588
10. Ma, Z., Leymann, F.: BPEL fragments for modularized reuse in modeling BPEL processes. Networking and Services, International conference on **0** (2009) 63–68
11. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: ICSOC. Volume 4749 of Lecture Notes in Computer Science., Springer (2007) 43–55
12. Betz, S., Klink, S., Koschmider, A., Oberweis, A.: Automatic user support for business process modeling. In: Proceeding of the Workshop on Semantics for Business Process Management at the 3rd European Semantic Web Conference 2006, Budva, Montenegro (June 2006) 1–12
13. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. In: OOPSLA, ACM (2004) 170–187
14. Decker, G., Kopp, O., Barros, A.P.: An introduction to service choreographies. it - Information Technology **50**(2) (2008) 122–127
15. Abate, F.R., ed.: The Oxford Dictionary and Thesaurus. Oxford University Press (October 1996)
16. Vanderfeesten, I.T.P., Reijers, H.A., van der Aalst, W.M.P.: Evaluating workflow process designs using cohesion and coupling metrics. Computers in Industry **59**(5) (2008) 420–437
17. Danylevych, O., Karastoyanova, D., Leymann, F.: Optimal stratification of transactions. In: ICIW, IEEE Computer Society (2009) 493–498
18. Bondi, A.B.: Characteristics of scalability and their impact on performance. In: Workshop on Software and Performance. (2000) 195–203
19. Eberle, H., Unger, T., Leymann, F.: Process fragments. In: OTM. Volume 5870 of Lecture Notes in Computer Science., Springer (November 2009) 398–405
20. Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., Tan, W.G.: Types of software evolution and software maintenance. J. of Software Maintenance **13**(1) (2001) 3–30
21. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. J. of Software Maintenance **17**(5) (2005) 309–332
22. Mens, T., Czarnecki, K., Gorp, P.V.: A taxonomy of model transformations. In Bézivin, J., Heckel, R., eds.: Language Engineering for Model-Driven Software Development. Volume 04101 of Dagstuhl Seminar Proceedings. (2004)

A.2 Optimal Stratification of Transactions

Authors:

USTUTT: Olha Danylevych

USTUTT: Dimka Karastoyanova

USTUTT: Frank Leymann

Published in:

- Proceedings of the 4^{th} International Conference on Internet and Web Applications and Services. ICIW 2009. May 24-28, 2009 - Venice/Mestre, Italy.

2009 Fourth International Conference on Internet and Web Applications and Services

Optimal Stratification of Transactions

Olha Danylevych, Dimka Karastoyanova, Frank Leymann
University of Stuttgart
Institut for Architecture of Applicationsystems
Universitaetstr. 38, 70569 Stuttgart, Germany
{olha.danylevych, dimka.karastoyanova, frank.leymann}@iaas.uni-stuttgart.de

Abstract

The performance of applications is influenced by the way its operations are grouped into global transactions. This in turns influences the performance of business processes which utilize these applications as implementations of process activities/steps. Stratified transactions, as produced by the stratification approach presented in this paper, is a way to manage a global transaction by combining the more elemental transactions coordinated using the two-phase commit protocol and queued transactions. The stratification approach can be applied for optimally fragmenting workflow-based service compositions and support the out- and in-sourcing scenarios. This paper formally models global transactions and investigates the mechanisms for building an optimally stratified transaction relying on formally defined evaluation criteria. We investigate the applicability of local search algorithms to the optimization of transaction stratification. In particular we consider hill-climbing, simulated annealing, and a novel hybrid method combining both approaches.

1. Introduction

Service compositions are often implemented with process-based approaches. In a Web Service environments BPEL is the de-facto standard. Nowadays BPEL processes are executed in a non-distributed fashion. Unfortunately, this does not reflect the needs of distribution inherent to the multi-organization scenarios that are pervasive in the current practice of Service Oriented Architecture. Therefore, distributed execution of business processes is a desirable result to be achieved.

In the literature there are approaches enabling the execution of parts of processes in a distributed environment. Some of those approaches, such as [3], focus on the design and implementation of distributed process execution engines. Others approaches, like [2], split the processes into multiple partitions, relying on coordination protocols to preserve the operational semantics of the global process. Unfortunately, the available approaches do not yet address issues like performance optimization. In fact, performance is usually disregarded for the sake of reducing the complexity of the scenarios to be dealt with.

In this work we present a method to achieve optimal distribution of work units deployed on different process engines/nodes according to criteria such as response time and concurrency. Our approach can be used in scenarios involving out-sourcing, in-sourcing, optimization of process and resource utilization (of both IT and human resources), etc. We assume known IT infrastructure- and organizational models. The

We model the overall process, called the *global transaction*, as a workflow made of atomic tasks called *basic transactions*. Inside the global transaction, the basic ones are coordinated by the means of coordination protocols such as the *two phase commit* (2PC) protocol [11]. However, 2PC protocols are not suitable for long-running transactions because of their high-abort probability and the the inefficient lock of resources, which are released only upon the protocol's completion. The exclusive lock of resources results in increased operational costs because of, for instance, the reduced throughput of executed transactions. Additionally, the high demands of 2PC protocols in terms of messages exchanges impacts the efficiency of the overall process.

For the purpose of optimizing the performance of the global transaction, we employ a *stratification* of basic transactions. The rationale of the stratification is to split the global transaction into groups of "smaller" transactions that are combined to preserve the (non-functional) properties and semantics of the global transaction, while improving its performance. The result is a so-called *stratified transaction*. In the general case, multiple stratified transactions can be obtained from the same global transaction, each characterized by a different trade-off in terms of overall execution time, response time, cost, etc.

The contributions of this paper include: the formal representation of the stratified transaction model, the stratifica-

978-0-7695-3613-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICIW.2009.79

tion algorithm, criteria to evaluate the effectiveness of the stratification and an approach based on local search to optimize it. The paper is structured as follows: Section 2 introduces the formal model for resources, transactions and their properties. Section 2.3 introduces dependence and stratification graphs that are the backbones of the stratification process. The criteria for the evaluation of the stratification is treated in Section 2.5. The algorithms that can be used for optimal stratification are presented and evaluated in Section 3. Finally, conclusions and future work are presented in Section 4.

2 Modeling Stratified Global Transactions

This Section presents our stratification approach and the assumptions we rely on. We model basic transactions as atomic entities characterized by properties such as execution time, recovery cost, invocation cost, etc. The properties of basic transactions are assumed to be known, and they are tightly related to the properties of the resources it manipulates. In the scope of our work, the term "resource" is a generalisation abstracting databases, message-queueing systems, application servers, and, more generally, the enterprise computing facilities. Similarly to the properties of the basic transactions, the properties of the resources are also known a-priori.

At run-time, basic transactions can be executed if and only if the conditions related to some of their properties are fulfilled, e.g. a basic transaction may be executed only after a specified *start time*. Basic transactions communicate with each other using mechanisms like, for instance, message passing. However, the exact mechanisms realizing this communication are outside of the scope of our work and they do not influence our approach. Exceptions, e.g. "server failure" and "internal failure", may occur during the execution of a basic transaction, leading to its failure. The likelihood of the execution failure of a basic transaction is represented by an *abort probability*. In the present work, we assume that the abort probability is given.

In the global transaction, the basic transactions are related with each other through different kinds of dependencies, such as data- and control dependencies. In our work we abstract from the particular types of dependencies by assuming that there is a inherent order according to which the basic transactions have to be executed inside the global transaction to respect these dependencies. This order is represented as a *dependency graph*, which we treat as a simplified control flow based on SPLIT and AND-JOIN constructs. As an assumption, we assume that there are no cyclical dependencies in the global transaction.

Our stratification approach is applied at modeling time. It adds an additional layer, called *stratification graph*, to the structure of the global transaction, making it a *stratified transaction*. The stratification graph is a hyper-graph laid on top the pre-existing dependence graph connecting the basic transactions. The nodes of the stratification graph are *strata* that group basic transactions and internally coordinate them using 2PC protocols. The edges of the stratification graph represent connections between two strata. The connections between strata come from the pre-existing dependencies that connect the basic transactions comprised in the different strata.

The strata communicate with each other using reliable messaging mechanisms. Each stratum is associated with a persistent message queue responsible for the consumption of incoming messages. We assume that the messaging middleware connecting the strata preserves the order of the messages in the input queues of each stratum, and handles run-time failures such a incorrect formatting and corruption of messages.

When executing a stratified transaction, a stratum can start if and only if all the required messages, implied by the dependencies incoming to the basic transactions it contains, are received. Intuitively, this corresponds to an AND-JOIN rule in the dependence graph. The execution of stratum completes as soon as all its basic transactions are done and the 2PC coordination protocol is successful.

This Section is structured as follows: the properties of the resources are modeled in Section 2.1). The modeling properties of the basic transactions is explained in Section 2.2. The dependence graph connecting the basic transactions is formalized in Section 2.3. Section 2.2 provides a list of possible transaction properties (this list serves as basis for further evaluation of stratification alternatives and does not has the ambition to be complete). Stratification graphs are modeled in Section 2.4. Finally, Section 2.5 evaluates our stratification approach in terms of message overhead introduced by the coordination among and within strata, the additional time costs because of the communication among strata, and a method to evaluate different prioritized criteria in the stratified transaction.

2.1 Properties of the Resources

We focus on two properties associated with resources: *type*, e.g. messages queues or databases, and *location*.Different granularities in the specification of the the type of the resource can be used, for instance, to differentiate among different kinds of databases (e.g. relational and xml-based, or MySQL v5 and PostreSQL). The rationale for the differentiation among different types of resources comes from the observation that usually the synchronisation costs (e.g. in terms of time and message overhead) for resources of different kinds are higher than the ones for the resources of the same type. The location of resources is expressed in terms of the particular physical machines on which they are deployed. The location is important because it is generally cheaper to synchronize resources located on the same machine than to synchronize resources spread over different locations.

Formally, $Res = \{r_1, r_2, ..., r_m\}$ denotes in the remainder a set of resources. Let L be a set of all possible locations in the system[1]. The possible locations associated to the resources in Res are described by the function $\mathcal{L} : Res \rightarrow \mathcal{P}(L)$.

2.2 Properties of the Basic Transactions

There is a number of properties of basic transactions that could be taken into acount, such as but not limited to invocation costs/recovery costs (transactions with high invocation/recovery costs should not be combined with those transactions with high abort probability), start time(the earliest possible time for starting the execution of a basic transaction, due time (the time by which a basic transaction should be completed), expected average/max execution time, commit probability (the probability that the basic transaction eventually commits), the abort probability (the probability that the basic transaction is aborted when run, for instance calculated using statistics or based on information about reliability of the used resources). A detailed description of these properties can be found in [6]. For the sake of brevity, in the remainder we focus on three properties of a transaction: costs related to the time of the execution and types and locations of the resources the transaction uses.

Let $\mathcal{T}$ be the set of basic transactions comprised in the global transaction.

Time costs. The function φ depicts the expenses of transaction the execution of which execution is completed at time τ. Each transaction t is related to a time-cost function f_t as follows:

$$\varphi : \mathcal{T} \rightarrow C \quad with \quad C = \{f : R^+ \rightarrow R^+\} \tag{1}$$

Then $\varphi(t)(\tau) = f_t(\tau)$ returns the costs for the transaction t if it has finished execution at time τ. Depending on the urgency of the basic transaction, f_t may be a polynomial, exponential or step-wise defined function, e.g. a constant until the due time and then exponential after the deadline is met.

The set of resource types used by a basic transaction are considered as a property of the latter; the same applies to the locations of the resources. The rationale for this is that the synchronisation costs grows with the amount of different employed resource types, and with the number of different locations of the resources. The property of a transaction to manipulate a set of resource types $\mathcal{A}_{resTypes}$ is described as:

$$\mathcal{A}_{resTypes} : \mathcal{T} \rightarrow \mathcal{P}(TypesRes) \tag{2}$$

where $TypeRes$ is the set of the resource types in the system, and $\mathcal{P}(TypeRes)$ its power set. The function $\mathcal{A}_{resLocs}$ denotes the property of a transaction defined as set of locations where the its employed resources are:

$$\mathcal{A}_{resLocs} : \mathcal{T} \rightarrow \mathcal{P}(L) \tag{3}$$

2.3 Dependence Graphs

The dependencies between the transitions in the set $\mathcal{T} = \{t_1, ..., t_n\}$ are modeled as the dependency graph G_a. A dependency graph is a directed graph, whose nodeset contains basic transactions in the system, and its edges in represent the dependencies between the transactions. Formally:

$$G_a = (V_a, E_a),\ with\ V_a \equiv \mathcal{T}\ and\ E_a \subseteq V_a \times V_a$$

The dependency between the basic transactions t_j and t_i is denoted by $t_i \rightarrow t_j$.

$$\forall t_i, t_j \in V_a : t_i \rightarrow t_j \Leftrightarrow e \in E\ with\ e = (t_i, t_j)$$

The dependence graph represents a partial order over the basic transactions. The existence of the edge $e = (t_i, t_j)$ in the dependence graph means that the transaction t_i has to be executed and completed before the execution of t_j can start. Notice that the dependence graph might be disconnected, i.e. there is no guarantee that given any two transitions, there is a path connecting one to the other or vice-versa.

2.4 Stratification Graphs

The idea of stratification is to partition the basic transactions into disjunct subsets, i.e. the strata. The strata are then composed in the stratification graph that sits on top of the dependence graph. Each stratum is a node in the stratification graph. By definition, an edge between two strata exists when there is at least one pair of basic transactions in them that are connected in the dependence graph. In a nutshell, each edge in the stratification graph corresponds to a set of edges in the dependence graph. Each stratum is associated with a persistent message queue that receives and manages the messages incoming to the stratum.

Figure 1 presents a dependence graph partitioned into strata, and its corresponding stratification graph. Let $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ be the set of basic transactions in one global transaction, and $G_a = (V_a, E_a)$ be the corresponding dependence graph. The basic transactions are partitioned into the strata $\mathcal{S} = \{s_1, s_2, ..., s_p\}$ in order to fulfill the following conditions:

$$\forall i \in [1, |\mathcal{S}|] : s_i \in \mathcal{P}(\mathcal{T})\ \ with\ |s_i| \neq 0$$

$$\bigcup_i s_i = \mathcal{T} \wedge \forall i, j \in [1, |\mathcal{S}|],\ i \neq j :\ \ s_i \cap s_j = \emptyset$$

[1] $\mathcal{P}(M)$ denotes the power set of M, empty set excluded.

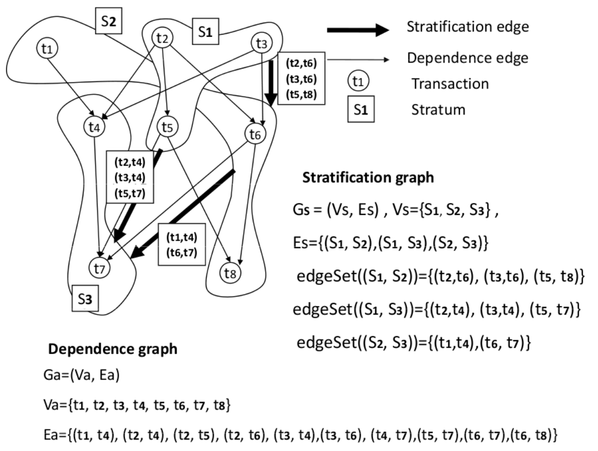

S2
S1
t1
t2
t3
(t2,t6)
(t3,t6)
(t5,t8)
Stratification edge
Dependence edge
t1 Transaction
S1 Stratum
t4
t5
t6
(t2,t4)
(t3,t4)
(t5,t7)
(t1,t4)
(t6,t7)
t7
t8
S3
Stratification graph
Gs = (Vs, Es) , Vs={S1, S2, S3} ,
Es={(S1, S2),(S1, S3),(S2, S3)}
edgeSet((S1, S2))={(t2,t6), (t3,t6), (t5, t8)}
edgeSet((S1, S3))={(t2,t4), (t3,t4), (t5, t7)}
edgeSet((S2, S3))={(t1,t4),(t6, t7)}
Dependence graph
Ga=(Va, Ea)
Va={t1, t2, t3, t4, t5, t6, t7, t8}
Ea={(t1, t4), (t2, t4), (t2, t5), (t2, t6), (t3, t4),(t3, t6), (t4, t7),(t5, t7),(t6, t7),(t6, t8)}

Figure 1. An example of stratification.

Simply put, the conditions reported above mean that the union of the strata gives the whole global transaction, and the strata do not overlap.

The process of stratification consists in the mapping of the dependence graph G_a to its stratification graph G_S:

$$Stratification : G_a \to G_S$$

where the stratification graph is defined as:

$$G_S = (\mathcal{S}, E_S \subseteq \mathcal{S} \times \mathcal{S})$$

where $\mathcal{S}$ represents the set of the strata.

In the remainder we consider as equivalent the stratum and the corresponding set of basic transaction it contains. The edges of the stratification graph G_S built on top of the dependence graph $G_a = (V_a, E_a)$ are constrained by the following condition:

$$(s_i, s_j) \in E_S : \ (i \neq j) \ \Leftrightarrow \\ \exists\, t_x \in s_i \ \exists\, t_y \in s_j : \ (t_x, t_y) \in E_a$$

It is generally possible to build multiple different stratification graphs out of a given dependence graph, each one corresponding to a different way of partitioning basic transactions among the strata.

Acyclic structure stratification graphs. Stratification graph is built on top of the dependence graph. The dependence graph is acyclic and defines the execution order of transactions. Given the fact that strata can be executed if and only if all the dependencies of its basic transitions are satisfied (see Section 2), in order to avoid deadlocks when executing the stratified transaction, the stratification graph must necessarily be acyclic.

Different strategies can be used to ensure that the stratification graph is acyclic. These techniques are analogous to the ones used for deadlock recognition and prevention. Getting inspiration from deadlock prevention techniques, basic transaction can be removed from a stratum on a cycle if it is a "root" or "leaf"-transaction; two transactions t_x and t_y can belong to the same stratum s if and only if all nodes on the way from t_x to t_y belong to the same stratum s. These mechanisms are described in detail in [6].

2.5 Evaluation of a Stratification

As reported in Section 2.4, it is generally possible to partition the same global transaction in different stratified transactions, each one characterized by a different stratification graph. Different stratified transactions will represent different trade-offs of distribution and concurrency, and will require different overhead of coordination among strata.For the sake of brevity, in the remainder we focus on the message costs of the 2PC needed to coordinate the basic transactions within a given stratum and the costs related to exceeding the deadlines of basic transactions. The interested reader will find in [6] a number of other criteria such as response- and execution time of the stratified transactions, invocation- and recovery costs, message costs and degree of concurrency.

Costs of 2PC-Messages. The "Costs of 2PC-Messages" function estimates the impact in terms of message exchanges of the 2PC protocols that internally coordinate the strata. It depends on the amount of messages needed, the locations of involved resources and the variety of resource types. The amount of messages needed for synchronisation depends on expected number of "rounds" (cycles) of 2PC protocols. We take the WS-AtomicTransaction [11] 2PC protocol as reference model. For the purposes of this Section, adopting a different 2PC protocol matters only in the average number of messages that are exchanged in one execution. We call "round" a complete set of messages needed for synchronisation within a stratum. In the throughout we assume that every round comprise four message exchanges: "Prepare Commit", "Acknowledge", "Commit/Abort". Each round can be completed with either a success (commit), or a failure (abort). In case of an abort, a new synchronisation round is started. The expected amount of synchronisation rounds $E(Rounds_s)$ depends, among other things, on the number of basic transactions comprised in the stratum s and their abort probability p_{abort}. The probability c that a 2PC synchronisation round within a stratum is successful is calculated as the multiplication of the success probabilities

$$Prob(t_i\ successful) = 1 - Prob_{abort}(t_i)$$

associated to every basic transaction t_i within the stratum.

The expected amount of synchronisation rounds for a stratum is thus:

$$c := Prob(Rounds_s = 1) = \prod_{t_i \in s} Prob(t_i\ successful)$$

$$Prob(Rounds_s = m) = (1-c)^{m-1} \cdot c$$

$$E(Rounds_s) = \sum_j Prob(Rounds_s = j) \cdot j$$

While the costs of a 2PC protocol may also depend on its configurations (e.g. the choice of the coordinator), further refinement of this metric is beyond of scope of this paper. By using the expected value of number of rounds we can estimate the message complexity needed for running a 2PC protocol. The amount f_{2PC} of messages needed for the synchronisation of one given stratum is calculated as:

$$f_{2PC} : \mathcal{S} \to N \ , \ \ f_{2PC}(s_x) = 4 \cdot |s_x| \cdot E(Rounds_x)$$

Given a stratified transaction, the cost function C_{2PC} for the message exchanges in 2PC protocols because of the stratification is defined as:

$$C_{2PC} : G_S \to R$$

$$C_{2PC}(G_S) = \sum_{s_i \in S} f_{2PC}(s_i) \cdot (\mathcal{F}_{ResTypes}(s_i) + \mathcal{F}_{ResLocs}(s_i))$$

Where $\mathcal{F}_{ResTypes}(s)$ and $\mathcal{F}_{ResLocs}(s)$ are defined as:

$$\mathcal{F}_{ResTypes}(s) = |\bigcup_{t_j \in s} \mathcal{A}_{resTypes}(t_j)|$$

$$\mathcal{F}_{ResLocs}(s) = |\bigcup_{t_j \in s} A_{resLocs}(t_j)|$$

Timeliness Costs. The "Timeliness Costs" of each basic transaction is defined on the time of its completion. Executions of basic transactions that complete later than the due time cost more, for instance because of the extended locking of resources. The timeliness cost of the overall stratified transaction is obtained by aggregating timeliness costs of the single basic transactions. A stratum is completed when all its basic transactions are successfully completed. $\tau_{finish}(s)$ denotes the time of the successful completion of the stratum s.

$$TC : \mathcal{S} \to R^+ \ , \ \ TimeCosts : G_S \to R^+$$

$$TC(s) = \sum_{t_i \in s} \varphi(t_i)(\tau_{finish}(s))$$

$$TimeCosts(G_S) = \sum_{s_i \in G_S} TC(s_i)$$

Overall evaluation. The overall evaluation of a stratification alternative is based on criteria such as the "Costs of 2PC-Messages" and "Timeliness Cost" criteria presented in this work. For the purpose of the evaluation, the criteria are weighted according to priorities that are provided by the modeler. Different evaluation criteria may conflict with each other, e.g. trying to reduce the "Costs of 2PC-Messages" might lead to an increased "Timeliness Cost" because of more coarse-grained strata. The input of correct priorities is paramount to the choice of the stratification alternative. However, a methodolgy to correctly specify the priorities is outside the scope of the present work. Given the criteria $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_k\}$, the overal evaluation is defined as :

$$\mathcal{F}(G_S) = \sum_i prio(\mathcal{E}_i) \cdot \mathcal{E}_i(G_S) \ , \ \ prio : \mathcal{E} \to R$$

3 Choosing the Optimal Stratification Alternative

The problem of finding the optimal stratification among all the possible alternatives is a generalization of the well-known *problem of graph partitioning* [9]. Likewise the graph partitioning problem, the optimal stratification problem is (at least) NP-Complete.However, differently from the graph partitioning problem, the optimal stratification problem aims at satisfying multiple conditions, i.e. the different prioritized evaluation criteria. Therefore, the known mechanisms for the graph partitioning problem are not sufficient for finding the optimal stratification. Appropriate mechanisms should implement a more generic approach to support the multiple prioritized evaluation criteria. Besides the evaluation criteria, the stratification might be subject to constraints (requirements) such as limitations to the amount of basic transactions in each stratum, the amount of strata, the number of edges in stratification graph, etc.

The application of evolutionary programming to achieve optimal stratification is treated in [6]. In this work we focus on local search optimisation algorithms, namely hill-climbing [10], simulated annealing [10], and a novel hybrid approach combining the two. In general, methods of local search are iterative and based on the idea that at each single step one element of the search space (in our case one of the stratification alternatives) is observed, evaluated and checked whether it satisfies the end condition. If the current element is not "good enough", its "neighborhood" is created by generating elements slightly different from the current one (in our case, we might move few basic transitions from a stratum to another). Using strategies such as "first best" [10], a neighbor element is choosen for the next step of the search. Hill-climbing is not applicable for the problems with many local optima. According to the particular case in input, simulated annealing may either lose the focus (and behave as a random search) or behave as hill-climbing.

In this paper we propose a hybrid approach (sketched in Figure 2) for finding the optimal stratification that combines the strengts of both hill-climbing and simulated annealing. The hybrid approach works as hill-climbing in the first phase. As soon as the search is trapped into a local optimum (e.g. the algorithm does not leave a certain element in a given number of iterations), a second phase based

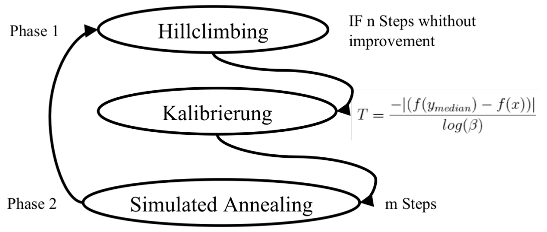

Phase 1
Hillclimbing
IF n Steps whithout improvement
Kalibrierung
$T = \frac{-|(f(y_{median}) - f(x))|}{log(\beta)}$
Phase 2
Simulated Annealing
m Steps

Figure 2. A schema of the hybrid approach

on simulated annealing is executed instead. Simulated annealing allows the search process to move out of the local optimum. A *calibration step* is needed to configure the simulated annealing according to the current state of the search. Simulated annealing is executed for a pre-configured number of steps, after which method moves into phase 1 (see figure 2). The decision about the parameters n and m as well as the congifuration of the SA phase and callibration are essential for the results of the search.

To evaluate the effectiveness of the optimization approaches, we implemented a prototype to apply the stratification approach and evaluated each of them on random generated global transactions (see [6] for the details). The hybrid approach showed the best results among the local search algorithms in the case of dependence graph with many edges (i.e. $|E| \approx |V|^2$). In this cases, hill-climbing showed the worst results. This could be explained by the size of the search space structure: the amount of possible stratification alternatives is higher as there are less dependency edges, and thus there are more local optima hill-climbing can get stuck into. In the case of dependence with few edges ($|E| \approx O(|V|)$) though, hill-climbing gives better results than the other local search approaches. However, the evolutionary programming gave the best results in all test cases. The evolutionary algorithm always outperforms local search methods if there are considerable computation time/resources. However, if relatively good results are sufficient and there are restrictions in computation resources and time, the hybrid approach is a better recommendation.

4 Conclusions

In this paper we presented an approach, called stratification, for fragmenting at modeling time a global transactions composed out of basic transactions that use multiple resources. The basic transactions are grouped into strata, that are coordinated to preserve the semantics of the original global transaction. Given the fact that multiple stratifications are generally possible, an optimal stratification is calculated based on the properties of the transactions and the resources they manipulate. Our stratification can be applied to service composition scenarios in a Web Service environment, increasing the concurrency and distribution of basic transactions over multiple sites, as well as scenarios involving out-sourcing and in-sourcing. Future work foresees the application of the stratification at run-time, i.e. on the fly, using execution statistics of the properties of transactions and their resource to perform the optimization. While in our existing work we deal with adaptation of processes modeled as global transactions, the adaptation of stratified transactions is still to be addressed. Furthermore, we envision the adoption of other optimisation approaches such as tabu search, and to investigate the optimal prioritization of evaluation criteria by experimenting with larger number of stratified transactions.

Acknowledgments.

The research leading to these results has received funding from the European Community's 7th Framework Programme under the Network of Excellence S-Cube - Grant Agreement no. 215483.

References

[1] The Open GROUP: Distributed Transaction Processing - The XA Specification. X/Open Company Ltd, February 1992

[2] Khalaf R., Leymann F., Role-based Decomposition of Business Processes using BPEL, Int'l Conf. on Web Services, ICWS 2006, industry track, IEEE Computer Society, Chicago, IL, USA, September 2006

[3] Martin, D., Wutke D., Leymann F.: A Novel Approach to Decentralized Workflow Enactment. 12th IEEE International EDOC Conference (EDOC 2008). Munich, Germany, September 15 - 19, 2008.

[4] Leymann, F., Transaktionsuntersẗtzung fuer Workflows. Informatik in Forschung & Entwicklung, 12(1), 1997.

[5] Leymann, F., Roller, D., Production Workflow. Concepts and Techniques. Prentice Hall PTR. Upper Saddle River. New Jersey 07458, 2000.

[6] Danylevych, O.: Stratifizierte Transaktionen, Diplomarbeit Nr. 2663, Universitaet Stuttgart, Institut fuer Architektur von Anwendungssystemen, Deutschland, 2008.

[7] Gray, J., Reuter, A. Transaction Processing: concepts and techniques. San Francisco, California. Morgan Kaufmann Publishers, 1993.

[8] Couloris, G., Dollimore, J., Kindberg, T., Distributed Systems. Concepts and design. Addison-Wesley Publishers Limited, 1994.

[9] Garey, M. R., Johnson, D. S., Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co. New York, NY, USA, 1990.

[10] Weicker, K. Evolutionäre Algorithmen. Teubner-Verlag, 2002.

[11] OASIS Consortium: WS-AtomicTransaction version 1.1. OASIS Standard, April 2007

A.3 Towards Identification of Fragmentation Opportunities for Service Workflows with Sharing-Based Independence

Authors:

UPM: Dragan Ivanovic

UPM: Manuel Carro

UPM: Manuel Hermenegildo

Submitted to:

- 2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2010).

Sharing-Based Independence-Driven Fragment Identification for Service Orchestrations[⋆]

Dragan Ivanović[1], Manuel Carro[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] IMDEA Software, Spain

Abstract. In the evolving world of Service-Oriented Computing (SOC), fragmentation and merging of service compositions are motivated by a number of design, performance and security concerns, such as composition reuse, dynamic load balancing, cross-organizational boundary process execution and resource access protection. The emphasis is on application of automatic design- and run-time methods for fragmentation that can automatically identify and separate orchestration fragments while preserving correctness and semantics of the original, non-fragmented, orchestration. While most of the current fragmentation approaches are founded on control-flow-based transition view of orchestrations (e.g. using Petri nets), we present an approach that takes into account the structure and use of data comprising state of an executing orchestration, and employs sharing-based independence analysis over orchestration components to identify fragmentation opportunities. We illustrate applicability of the sharing-based approach on fragment identification, with reflections on security and resource access control, and show how the existing analysis tools for logic programs can be used as the underlying technology for this kind of analysis.

1 Introduction

Service-Oriented Computing (SOC) enables interoperability of functionally specialized components with low coupling. In that context, service compositions are mechanism for expressing business processes (i.e. wokflows) that include other services, and are exposed as services themselves. Composition notations and languages, such as BPMN [Obj09], WS-BPEL [JEA$^+$07], XPDL [Wor08], Let's dance [ZBDtH06] or DecSerFlow [vdAP06] allow process modelers and designers to view a composition from the point of business logic and processing requirements related to parallelism and data flow. The now fashionable service mash-ups are also tools for building (usually simplified) customized orchestrations from known service components in a user-centric way.

[⋆] The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483. Manuel Carro and Manuel Hermenegildo were also partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo was also partially supported by EU projects FET IST-15905 *MOBIUS*, FET IST-231620 *HATS*, and 06042-ESPASS.

These service compositions, unlike the back-end worker services, are coarse-grained software components that normally implement higher-level business logic, and allow streamlining and control over mission-critical business processes inside an organization and across organization boundaries. However, the centralized manner in which these processes are designed and engineered does not necessarily reflect the desirable properties in their run-time environment. To enable distributed orchestration execution, optimize network traffic, ensure privacy, or create a basis of reusable orchestration components, various fragmentation approaches have been proposed [WRRM08,BMM06,TF07].

This proposal focuses on the identification of fragments on the basis of a notion of *independence* between orchestration activities. This is done by considering how these activities handle resources such as data that represents the state of an executing orchestration (i.e. process variables), external participants (such as partner services and external participants), resource identifiers, and mutual dependencies. The underlying idea is that the fragments may adjust and vary the arrangement of orchestration activities, while preserving the essential properties, such as correctness and transactional integrity. A quantitative measure of independence of orchestration fragments can also be related with the degree of cohesion and coupling of orchestration components.[3] Greater independence degree between fragments means lower coupling, and thus higher maintainability and easier evolution. Conversely, a greater dependence may lead to situations where a change in an activity leads to changes in dependent / depended activities as well.

Based on data representation of different resources, we concentrate on data independence between orchestration activities, using techniques based on sharing and groundness analysis [MH92,MS93,MH91], taking into account different resources, such as variables holding the state of an running composition, references to external services, and, in general, references to (maybe external) resources which induce some degree of dependence between the identified fragments.

2 Overview of the Independence-Based Approach

Independence analysis, which we are formulating in this paper based on the notion of *sharing*, is a general property that can be applied to both upper and lower layers of software architecture, and consequently, to various parts of the service stack.[4] We will focus on the service composition layer, were we model the relationships between different entities used within a orchestration as *data structures* subject to sharing analysis, and the orchestration activities as *goals* in a logic program [SS94].

Figure 1 gives an example of the idea behind the general approach. A part of a orchestration (leftmost) between points 1 and 2 is a sequence of invocations of partner services *A* and *B*. If these two activities do not access the same resources, we can isolate two fragments (middle) guided by a notion of independence. These fragments can be *split* with the certainty that no communication between them will be necessary.

[3] Note that this is a general S.E. concept which still seems completely applicable to S.O.C.

[4] In fact is a technique general enough to be applied to any phenomena which can be modeled in a programming language, and has a plethora of applications in e.g. the field of programming languages.

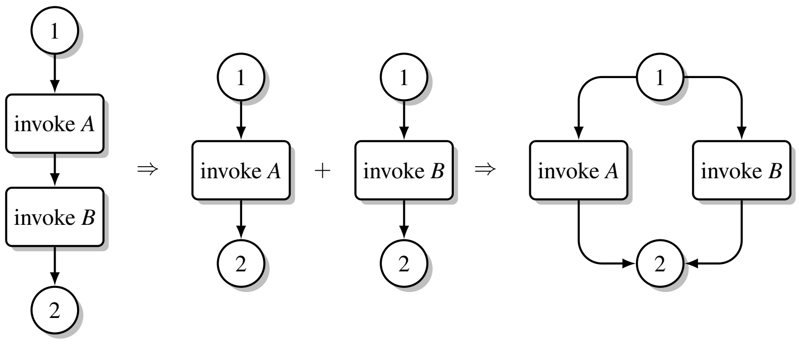


Fig. 1. An illustration of sharing-based splitting

Our approach will ensure that independence between goals imply independence between the activities they stand for. We model control dependencies as a special case of data dependencies on outgoing transition indicators from wokflow activities. The entities which have an impact on dependence and that we want to model typically include:

- Variables (e.g. in BPEL) that hold (part of) the state of the executing processes. This is a classical case of data-based depenency found in many programming languages. The analysis needs to take into account that these variables are usually complex XML-like structures.
- Services used in the composition (partners in BPEL parlance). Accesses to partners need to respect their business protocol, which can be as simple as a synchronous *request-reply* cycle, or as complex as an asynchronous exchange of messages with multiple stateful callbacks and conversations going on in the course of such exchange. We model dependencies on partner services by mapping these dependencies to a to a sharing problem between new, *invented* variables that are appropriately used in the model whenever the original orchestration accesses these external services.
- Access to databases or other shared resources, either within local activities in the composition, or within external (partner) services. We can map resource identifiers and database connection specifiers to new logical variables referenced in activities that use resources/databases.

2.1 Underlying Analysis Techniques

In our approach we use abstract interpretation-based [CC77,NNH05,Bru91] tools to perform sharing-analysis in logic programs. Different abstract domains for sharing (and freeness / groundness) have been designed over the years [MH91,LS02,BS92,HK03]. They differ on the accuracy of the approximation and on the resources (memory and time) they require, but all of them produce program-point information which can later be used to deduce dependencies which can be used to rewrite / annotate the program for a variety of purposes —including, but not exclusively, parallel execution. The choice of a domain impacts the accuracy and complexity of the analysis, and the behavior of

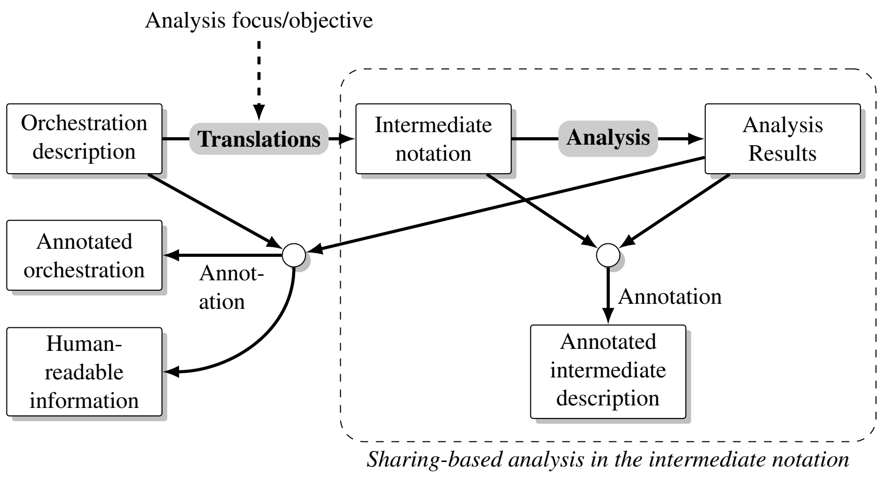

Analysis focus/objective
Orchestration description
Translations
Intermediate notation
Analysis
Analysis Results
Annotated orchestration
Annot-ation
Human-readable information
Annotation
Annotated intermediate description
Sharing-based analysis in the intermediate notation

Fig. 2. Overview of the sharing-based fragmentation analysis and its possible outcomes and applications.

the domain operations can certainly be taken into account when encoding the original orchestration in order not to loose precision and to capture as faithfully as possible the original behavior. Selecting the right analysis domain and the right orchestration encoding is a very interesting challenge which we plan to explore in a future, and in which precision and complexity play relevant roles.

While the results of sharing / freeness analysis can be used for a variety of program transformations and annotations, in this paper, and for illustration purposes, we will mainly use its application to extract independence conditions, which were primarily designed to determine the possibility of safe parallel execution. Even if our ultimate aim is not necessarily performing parallel execution, the independence / non-interference conditions which a syncronization-free parallel execution requires happen to be interesting and applicable beyond parallelism, as we will see in Section 3.6. Notwithstanding, other fragmentation approaches may find it valuable the direct use of sharing / freenes information without the information filtering that applying them to dependency analysis and later to annotations for parallelism unavoidable brings about.

2.2 The Big Picture

A picture depicting the overall scheme of our approach is shown in Figure 2. A orchestration description, in an adequate abstract or executable notation, is first translated into an intermediate notation, which in our case is a logic program that we generate so that it captures relevant information and is amenable to sharing / freenes analysis. In our case these analyses are performed by the tools provided by the Ciao / CiaoPP suite [HBC$^+$08,HPBLG05].

The intermediate notation is an abstract model of a orchestration, which is more concrete than e.g. Petri-net based orchestration models [OVvdA$^+$07,vdAtHW03], as for our purposes we need to faithfully capture some operations on data, sometimes of certain complexity (see Figure 4), but maybe less concrete than an executable implementation in e.g. BPEL, since we do not necessarily need to completely mimic the actual operational semantics. Depending on the focus and objectives of a particular analysis (identification of fragments, security concerns, resource management) different intermediate versions of the actual orchestrations can be generated.

The analysis stage operates on the intermediate notation and its results can be used to either transform it into an annotated program (where e.g. independence between tasks / activities can be clearly seen) or, alternatively, be mapped back to a tool which can manipulate the original orchestration code [KL06] or be presented in a format which is easy for a designer or service engineer to understand and apply to the service design being undertaken.

In this paper we focus on some of the steps in Figure 2. More concretely, we will, by means of illustrative examples:

– Informally present how the translation from some not-yet-identified orchestration language would proceed (by looking at the results of some translations) in order to capture relevant information for some types of fragmentation purposes.
– Study how this intermediate representation can be analyzed and the results of this analysis used to annotate the abstract version of the orchestration.
– Use this annotated orchestration version in order to identify fragments composed of one or more activities which present independence, and interpret this independence in the light of the purpose of the fragmentation.

As we previously discussed, other very interesting tasks, such as using sharing / freeness analysis to detect directly fragmentation opportunities for several scenarios, instead of going through the dependency / annotation path are left as future work.

It is worth noting that sharing / freeness analysis is non-trivial and doing it manually is error prone and inefficient and, especially important, incompatible with the requirements of automated service adaptation and evolution. Automatic sharing analysis, on the other hand, although hard and computationally intensive, can be employed automatically on demand, and, if done through a correct analysis (which is for example the case if using abstract interpretation) guarantees the safety of its results: although it may sometimes miss an opportunity for split, being a conservative approximation it ensures that any two asks inferred to be independent at runtime will definitely be so.

3 Sharing-Based Fragment Identification

Before we proceed to technical details, to illustrate the sharing-based approach to orchestration fragment identification, we take the example that is schematically displayed in Figure 3. That is a simple example of a orchestration A that consists of five activities ($A_1 ... A_5$), one of which (A_4) is a structured activity itself, with two sub-activities ($A_{4,1}$ and $A_{4,2}$). The solid line arrows drawn between activities represent dependency links akin to those found in BPEL's `flow` construct. The mechanism of dependency links is able to express a wide class of orchestration patterns, with some exceptions [vdA03,vdABtHK00].

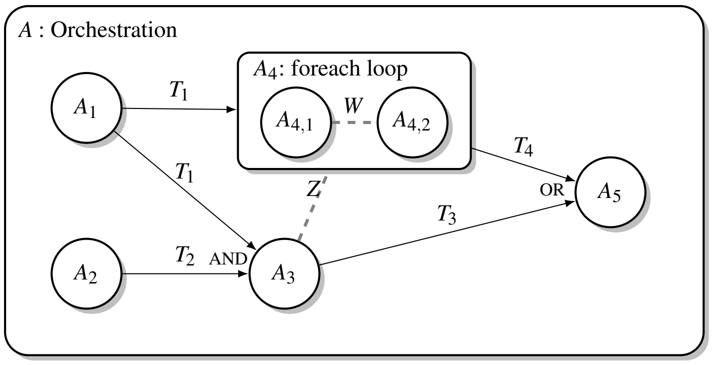


Fig. 3. A orchestration example for sharing-based analysis. Arrows represent control flow dependencies. The dashed lines are additional dependencies that exist in the alternative scenario.

Outgoing links (i.e. the transitions) from activity A_i are marked as T_i, and can have Boolean true or false value. Join conditions for activities with more than one incoming link can be arbitrary logical conditions over the incoming links, although the simple AND and OR joins are the most common case. Circular dependencies are not allowed.

The arrows in the figure essentially depict control-flow dependencies. For a while, we will ignore the dashed dependency lines marked with Z and W, and will come back to them when we look at the impact of additional data dependencies on the analysis. Each activity in the orchestration is responsible for setting its outgoing transition upon successful completion. An activity with incoming dependencies can begin executing only when the status of all incoming links is known. Thus, the links behave as three-state logical variables. If the join condition for activity A_i is not met, either the orchestration execution fails with an exception, or the failure is silently propagated by setting the outgoing transition T_i to false. The latter case corresponds to BPEL-style *dead path elimination* [KKL07].

The orchestration description in Figure 3 is translated into an intermediate notation which is directly amenable to sharing analysis. Let us note that encoding full data dependencies in Petri-net based approaches [MWL08,MaBP02] needs a very specific and cumbersome encoding, as as done in [FMB00]. Indeed, encoding data dependencies in Petri Nets is an area which has not received much attention, in contrast with data-flow program analysis. Additionally, taking into account structured subactivities and complex join conditions requires the introduction of several additional auxiliary nodes wich hinder the understanding of the resulting model by e.g. an engineer.

3.1 Representing Orchestration in Intermediate Logic Program Notation

Since we are interested in modeling only those aspects of orchestration behavior that are relevant for fragment identification based on sharing analysis, we can omit non-substantial part of orchestration logic that does not influence data dependencies. We model an orchestration as a Horn clause, with the body that is an ordering of individual

```
:- module(_, [a/2],
          [assertions]).
:- entry a/2:
          ground*var.

a([X], [Y,T5]):-
       a_1([], [], [T1]),
       a_2([], [], [T2]),
       a_3([T1,T2], [X], [T3]),
       a_4([T1], [X], [T4]),
       a_5([T3,T4], [], [Y,T5]).
a_1(_, _, [[]]).
a_2(_, _, [[]]).
a_3(_, _, [[]]).
a_4(_, [[]], [[]]).
a_4(P, [[N|L]], [T4]):-
       a_4_1([], [], [_T4_1]),
       a_4_2([], [N], [_T4_2]),
       a_4(P, [L], [T4]).
a_4_1(_, _, [[]]).
a_4_2(_, _, [[]]).
a_5(_, _, [[],[]]).
```

Fig. 4. Logic notation for the orchestration A in Fig. 3

activities. If an orchestration A has $n > 0$ activities, then the clause has the shape:

$$A \leftarrow A_1, A_2, \ldots, A_n\,, \tag{1}$$

where A_i, $i = 1..n$, stands for the logical goal associated with the ith activity, of the form:

$$a_i(\overline{P}_i, \overline{R}_i, \overline{W}_i)\,. \tag{2}$$

Arguments of a_i are lists of (logical) variables. $\overline{P}_i$ is the set of incoming transitions on which A_i depends. $\overline{R}_i$ is the set of data inputs read by A_i, and $\overline{W}_i$ is the set of values computed by A_i. $\overline{W}_i$ always includes T_i, the computed value of the outgoing transition from A_I. In the concrete (Ciao) Prolog notation, we represent $\overline{P}_i$, $\overline{R}_i$ and $\overline{W}_i$ with lists.

Ordering of activities in (1) needs to respect two constraints: (i) the *link dependency constraint*, which mandates that an activity A_i cannot depend on outgoing transition of any activity A_j where $j \geq i$ (i.e. $T_j \notin \overline{P}_i$); and (ii) the *read-write dependency constraint*, which stipulates that an activity A_i cannot use as an input the result of any activity A_j where $j \geq i$ (i.e. $X \in \overline{W}_j \Rightarrow X \notin \overline{R}_i$).

Since circular link dependencies are ruled out, at least one ordering that respects the link dependency constraint can always be found and statically decided for a given orchestration. In many cases, on top of that we can easily impose the read-write dependency constraint when translating orchestration from a concrete orchestration language into the intermediate logic program form. Here we show an approach when in the concrete language activities operate by updating values of named variables.

Suppose, for instance, that x is a symbolic name for a variable within the scope of an orchestration that is read and/or written by its activities. In an ordering of activities that respects the link dependency constraint, at any position $i = 1..n$, we model the value of x after A_i with logical variable X_i that is either X_{i-1} if A_i does not update x, or the updated value $Y \in \overline{W}_i$. We assume that X_0 is a free variable unless it is a part of the message with which the orchestration was invoked. To ensure the read-write dependency constraint, when activity A_i reads the current value of X, we put X_{i-1} into $\overline{R}_i$.

For the goal A that corresponds to the whole orchestration, we use the format analogous to (2). Since orchestration is autonomous piece of code, we omit its $\overline{P}$, and use its $\overline{R}$ and $\overline{W}$ to model elements of the request and the response message, respectively.

Figure 4 shows the translation of the orchestration into the intermediate notation of a logic program. The `entry` declaration declares that the predicate `a/2` is the entry point to the orchestration. Here, it takes tow arguments: an input message with data element `X`, and the output message with data element `Y` together with the outgoing transition (which comes from the final activity in the orchestration). The first two arguments are `ground` on entry, and the latter two are `var`.

Lines 6-11 on Figure 4 describe the orchestration in terms of (1) and (2). In this case we use ordering (A_1,A_2,A_3,A_4,A_5), but we could have swapped lines 7 and 8 (A_1 and A_2) and lines 9 and 10 (A_3 and A_4) at will without breaking the ordering constraints.

We also need to supply rules for individual activities, and these are given in lines 12-22 on Figure 4. For simple activities, the rule for a_i is a fact where the first two arguments are ignored (signified with an "_", and the third (corresponding to $\overline{W}_i$) is a list of ground terms for each computed value. Throughout translation, we use the empty list symbol "`[]`" (or *nil*) when we need a ground term. That gives us the rules for a_1, a_2, a_3, $a_{4,1}$, $a_{4,2}$, and a_5. We are allowed to ignore $\overline{P}_i$ and $\overline{R}_i$, as well as to substitute nil for computed values in $\overline{W}_i$ because we are not concerned with modeling actual computations, just with giving appropriate information for sharing-based independence analysis.

Rules for structured activities, such as looping and branching, are slightly more complex. Activity A_4, which is a foreach construct, has a base-case rule that exits the loop (line 15 on Figure 4), and a tail-recursive rule that involves activities $A_{4,1}$ and $A_{4,2}$ inside the body of the loop (lines 16-19). Shape of the rules depends on the original semantics of the orchestration. For instance, if the body of the loop updated an orchestration variable with symbolic name z, then $\overline{R}_4$ would need to include value of z before the loop, and $\overline{W}_4$ would need to include the value of z after the loop, in order for the tail recursion (line 19) to correctly thread the updates. If, however, z was local to the body of the loop (as in the example discussed in Section 3.5), that would not be necessary.

3.2 Obtaining Annotated Orchestration

On the intermediate logical program representation of the orchestration from Figure 4, we apply several automated analysis algorithms from the CiaoPP analysis suite []. First, the set sharing and freeness analysis (`shfr`) is performed to detect aliasing between different variables appearing in the program. Second, on the basis of sharing analysis, we use the non-strict independent AND-parallelism analysis (`nsiap`) [] to infer dependence or independence of goals on previous execution of other goals. Third, we use the information given by the independence analysis and feed it to the of the UUDG code annotation algorithm [CCH08] that inserts AND-parallelizing primitives. The result is an annotated orchestration in form of a logic program, which is used as a basis for fragment identification, explained in the following subsection.

The annotated version of the logic representation of the orchestration (Figure 4) is shown on Figure 5. In the main body of the orchestration (lines 1-8), which used to be just an ordering of $n = 5$ activity goals, there are some modified goals of the form "A_i&>H_i" (start A_i in parallel as task H_i), as well as some additional goals of the form "H_i<&" (wait for task H_i to finish). Another kind of annotation can be seen in the body of the foreach loop (lines 11-14): "A_i&A_j" (start A_i and A_j in parallel, and wait for both to finish), which boils down to "A_i&>H_i, A_j&>H_j, H_i<&, H_j<&" [CCH07].

```
1   a([X], [Y,T5]) :-
2           a_2([], [], [T2]) &> H2,
            a_1([], [], [T1]),
4           a_4([T1], [X], [T]4) &> H4,
            H2 <&,
6           a_3([T1,T2], X, [T3]),
            H4 <&,
8           a_5([T3,T4], [], [Y,T5]).

10  a_4(_, [[]], [[]]).
    a_4(P, [[N|L]], [T4]) :-
12          a_4_1([], [], [_T41]) &
            a_4_2([], [N], [_T42]) &
14          a_4(P, [L], [T4]).
```

Fig. 5. A digest of the annotated orchestration code from Fig. 4

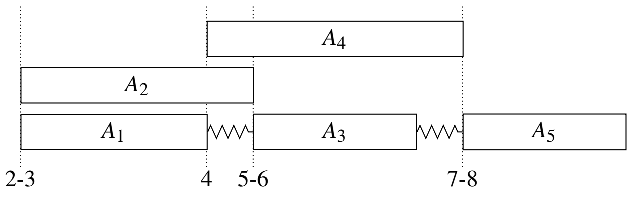


Fig. 6. Graphical representation of the parallelizing annotations for the orchestration.

These parallelizing annotations inserted by the UUDG algorithm are derived from the independence analysis results (which are in turn derived from the sharing and freeness analysis), and are known to start activities as early as possible, and to wait for their completion as late as possible, without breaking dependencies (up to relabeling of local/remote tasks). The original idea of the parallelizing annotations is operational parallelization of logic program execution, and indeed the annotations can be used to inform runtime fragmentation in a distributed orchestration enactment milieu. Here, we use the annotations as start and finish markers for safe reordering of orchestration activities in the process of obtaining fragments.

The UUDG algorithm rearranges mutually independent activities to ensure earliest start / latest finish, if such rearrangement does not break the dependencies. Because we model all dependencies essentially as data dependencies (transitions are a special case), the link and the read-write dependency constraints ensure that all compliant orderings may differ only in relative ordering of mutually independent activities. Therefore, we can conclude that the initial choice of ordering of activities in (1), as long as it complies with the two constraints, does not have substantial impact on the annotations, because in the fragmentation identification (next subsection) we do not distinguish between "start A_i, start A_j" and "start A_j, start A_i".

Figure 6 schematically shows the parallelization of a from Figure 5. The blocks correspond to execution of activities, and the numbers correspond to lines of the annotated code in Figure 5. The relative size of blocks is not significant, but how they overlap. The bottom thread, starting with A_1 corresponds to the body of a. First, A_2 and A_1 may be executed simultaneously. After completion of A_1, A_4 is immediately scheduled for par-

$$\begin{tikzcd}
A \arrow[r, dashed, "\text{Fragment}", "\text{identification}"'] \arrow[d, "\text{Independence analysis + Annotation}"'] & A_f \\
A^* \arrow[r, "\text{Activity}", "\text{rearrangement}"'] & A_f^* \arrow[u, "\text{Annotation collapsing}"']
\end{tikzcd}$$

Fig. 7. The fragment identification workflow

allel execution (line 4). The next action to perform is to wait for the termination of A_2 which is needed to to start A_3, since there is a dependency due to the shared transition variable `T2`. Finally, A_5 can start as soon as both A_4 and A_3 finish.

3.3 Static (Design-Time) Fragment Identification

We use the annotated version A^* of the orchestration A for identifying fragments. As already mentioned, the annotation algorithm based on data-sharing dependencies [CCH07] rearranges the initial representation by introducing parallelization primitives `&>`, `<&`, and `&` that indicate the earliest possible start and the latest possible join of activities in the orchestration, in a deterministic and optimal way under the detected dependencies. Here we show a technique that manipulates these annotations to obtain a fragmented version of an annotated orchestration.

At design time, we perform fragmentation typically to separate from the overall orchestration a fragment of activities that fulfill some criterion, such as: activity type, complexity, user preference, resources used, control or data dependencies, etc., or a combination thereof. Whatever criteria is used, we assume that it generates a non-empty subset $\mathscr{A}$ of activities from orchestration A. We can thus define fragmentation as the problem of producing a fragmented orchestration:

$$A_f \leftarrow F_-, F, F_+ \,, \tag{3}$$

which is a dependency-safe rearrangement of activities in A, such that: (a) F contains the minimal subset of A that includes all activities in $\mathscr{A}$, and (b) goals related to any activity that is not in F appear either in F_- or F_+, but not in both. F_- or F_+ can be empty.

To achieve the goal of fragment identification, we follow the workflow shown on Figure 7. We have already explained how the annotated orchestration is obtained, and collapsing it back to the notation without annotations is simply a matter of converting each “A_i`&>`H_i” into “A_i” and removing each “H_i`<&`”. Therefore, we will concentrate here on rearranging activities in the annotated orchestration:

$$A^* \leftarrow A_1^*, A_2^*, \ldots, A_N^* \; (N \geq n) \tag{4}$$

where A_i^* $(i = 1..N)$ is either A_k, A_k`&>`H_k, or H_k`<&`, for some $k = 1..n$, to obtain its fragmented version:

$$A_f^* \leftarrow F_-^*, F^*, F_+^* \tag{5}$$

When rearranging A^* into A^*_f, we must take care not to break dependencies. In the sequence of N goals that constitute the body of A^*, it is safe to move a goal of the form “H_k<&” to the left, but not past the corresponding “A_k&>H_k”. Indeed, the annotation algorithm ensures that if “H_k<&” appears at position j in A^*, no goal to the left of j (i.e. at a position $i < j$) depends on A_k. For the same reason, it is safe to move a goal of the form “A_k&>H_k” to the right, but not past the corresponding “H_k<&”. Of course, if we place “A_k&>H_k” next to “H_k<&”, we obtain an (immovable) “A_k”.

The algorithm to achieve (5) operates on a working copy of A^*, with i and j being at each step the leftmost and the rightmost position of goals associated with activities in $\mathscr{A}$ in the sequence. In the first phase, we move goals that are not associated to activities in $\mathscr{A}$ to either side of F^*.

(i) For each “A_k&>H_k”, $A_k \notin \mathscr{A}$, positioned to the left of i, with the corresponding “H_k<&” positioned to the right of j, either move the former to position j, or move the latter to position i.
(ii) Move each “A_k&>H_k”, $A_k \notin \mathscr{A}$, positioned between i and j, rightwards as far as possible towards position j.
(iii) Move each “H_k<&” positioned between i and j with the corresponding $A_k \notin \mathscr{A}$, leftwards as far as possible towards position i.

In the second phase, we try to compact the goals associated to $\mathscr{A}$, which include “A_k” and “A_K&>H_k” with $A_k \in \mathscr{A}$, and the corresponding “H_k<&”:

(iv) Going from left to right, move each “H_k<&” associated to $\mathscr{A}$ and positioned to the right of i, leftwards as far as possible without passing another goal associated to $\mathscr{A}$.
(v) Going from right to left, move each “A_k&>H_k” associated to $\mathscr{A}$ and positioned to the left of j, rightwards as far as possible without passing another goal associated with $\mathscr{A}$.

After applying the above algorithm, the resulting A^*_f can be separated into three parts: goals at positions from 1 to $i-1$ constitute F^*_-; goals at positions between i and j constitute F^*; and goals at positions $j+1$ to N constitute F^*_+. That means that F^*_- or F^*_+ can be empty. The final F_-, F and F_+, which constitute A_f, are obtained by collapsing F^*_-, F^* and F^*_+, respectively.

The algorithm guarantees that all activities from $\mathscr{A}$ will be concentrated in F^* and F, but it may happen that some other activities need to be included in the fragment, because otherwise the dependencies would be violated. In that case, the algorithm ensures that a minimal number of such activities is included. However, in extreme cases, it may happen that the fragment is identical with the initial workflow (both F_- and F_+ are empty), even if $\mathscr{A}$ is a proper subset of activities in A, and therefore fragmentation does not have any effect. The reader can easily verify that is the case when $\mathscr{A} = \{A_1, A_2, A_5\}$.

The examples of fragmentation using this algorithm are shown in Table 1. The case $\mathscr{A} = \{A_1, A_5\}$ illustrates inclusion of additional activities into the fragment, and these activities in F that are not in $\mathscr{A}$ can themselves be seen as sub-fragments of F, as illustrated on Figure 8. We also see that the algorithm can produce more than one possible fragmentation to choose from (e.g. the case $\mathscr{A} = \{A_2, A_3\}$).

Another important point is that although the fragmented orchestration has the form (3), that does not mean that this approach is limited to cases of sequential fragment

$\mathscr{A}$	F^*_-	F^*	F^*_+
$\{A_1\}$	(*empty*)	A_1	A_4&>H_4, A_2, A_3, H_4<&, A_5
$\{A_2,A_3\}$ [1]	A_1	A_2, A_3	A_4, A_5
$\{A_2,A_3\}$ [2]	A_1, A_4	A_2, A_3	A_5
$\{A_1,A_5\}$	A_2	A_1, A_4&A_3, A_5	(*empty*)
$\{A_3,A_5\}$	A_2&>H_2, A_1, A_4, H_2<&	A_3, A_5	(*empty*)
$\{A_2,A_4\}$	A_1	A_2&A_4	A_3, A_5
$\{A_4\}$	A_2&A_1	A_4	A_3, A_5

Table 1. Examples of sharing-safe fragmentation of the annotated orchestration.

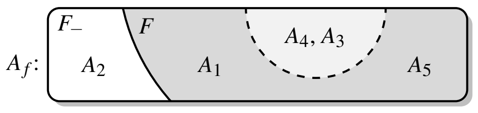


Fig. 8. A schematic representation of fragment identification for $\mathscr{A} = \{A_1,A_5\}$

execution. The ordering of activities in the intermediate logic representation (1), as we said, is not an operational sequencing. In fact, if we are interested in parallelization opportunities, we can see from the examples in Table 1 for $\mathscr{A} = \{A_2,A_3\}$ that the fragment F consisting of A_2 and A_3 can be executed in parallel with A_4, which is either in F_- (case [1]) or in F_+ (case [2]).

The identified fragment can be formalized as a new, synthetic activity within the orchestration, with its own rule following the structure of (1). For uniformity, we model the goal corresponding to the fragment F using the form (2) for activity goals:

$$f(\overline{P}_f,\overline{R}_f,\overline{W}_f)\,, \tag{6}$$

where $\overline{P}_f$ and $\overline{R}_f$ contain all incoming link dependencies and input data, respectively, used by activities in F, and $\overline{W}_f$ contains the computed values (including transitions) from activities in F used by the rest of the orchestration. An example of how the fragment is represented as a synthetic activity (or a sub-orchestration) in the intermediate logic program form for the case $\mathscr{A} = \{A_2,A_4\}$ is shown on Figure 9.

```
a([X], [Y,T5]):-
        a_1( [], [], [T1]),
        f([T1], [X], [T2,T4]),
        a_3([T1,T2], [X], [T3]),
        a_5([T3,T4], [], [Y,T5]).

f( [T1], [X], [T2,T4]):-
        a_2( [], [], [T2]),
        a_4( [T1], [X], [T4]).
```

Fig. 9. Representing fragment as a synthetic activity.

```
:- module(_, [a/2],
          [assertions]).
:- entry a/2:
         ground*var.

a([X], [Y,T5]):-
       a_1([], [], [T1]),
       a_2([], [], [T2]),
       a_3([T1,T2], [X], [Z,T3]),
       a_4([T1], [X,Z], [T4]),
       a_5([T3,T4], [], [Y,T5]).
a_1(_, _, [[]]).
a_2(_, _, [[]]).
a_3(_, _, [[],[]]).
a_4(_, [[]], [[]]).
a_4(P, [[N|L]], [T4]):-
       a_4_1([], [], [W,_T4_1]),
       a_4_2([], [N,W], [_T4_2]),
       a_4(P, [L], [T4]).
a_4_1(_, _, [[],[]]).
a_4_2(_, _, [[]]).
a_5(_, _, [[],[]]).
```

Fig. 10. Modified example in the intermediate notation with additional data dependencies

```
a([X],[Y,T5]) :-
       a_1([], [], [T1]) & a_2([], [], [T2]),
       a_3([T1,T2], [X], [Z,T3]),
       a_4([T1], [X,Z], [T4]),
       a_5([T3,T4], [], [Y,T5]).

a_4(_, [[],_], [[]]) :- !.
a_4(P, [[N|L],Z], [T4]) :-
       (a_4_1([], [], [W,_T4_1]), a_4_2([], [N,W], [_T4_2]))
       & a_4(P, [L,Z], [T4]) .
```

Fig. 11. A digest of annotation results for a and a_4 with additional data dependencies.

3.4 Dynamic (Run-Time) Fragment Identification

In a dynamic orchestration enactment scenario, the annotations can straightforwardly inform the enactment engine(s) on opportunities for distributing execution of fragments without violating control and data dependencies.. In practice, and if parallelism and speedups is really sought, a very relevant information is the actual feasible degree of parallelization — i.e., how many parallel activities can be launched in a give parallel architecture before the associated overhead is too large.

In the annotated version of a in Figure 5, a_1 and a_2 are started in parallel (lines 2 and 3). As soon as a_1 is finished, a_4 is started, thus potentially running in parallel with a_1. Next, a_3 is started as soon as a_2 finishes, thus possibly running in parallel with a_4. Finally, a_5 can be started after finish of both a_3 and a_4.

Fragmentation opportunities for a_4 are even more straightforward, as the analysis clearly indicates that both sub-activities $a_{4,1}$ and $a_{4,2}$, as well as the rest of the `foreach` loop can be delegated to different orchestration enactment threads. In terms of a maximal number of fragments of a_4 that are independent at any moment, it can be easily verified that it is equal to $2N+1$, where N is the length of the input list. Since at most two activities of a can run simultaneously, the maximum number of basic (simple activity) simultaneously executing fragments within a is thus $2N+2$.

3.5 Additional Data and Conversational Dependencies

Since we have used data sharing to model control-flow dependencies, introduction of the additional data dependencies between orchestration activities is straightforward and does not require changes in the analysis itself. That is a major advantage over e.g. Petri net based approaches [TF07], which have to be significantly restructured to take data into consideration, for instance by moving towards colored Petri nets with special coordination and data link mechanisms [KL06,LWC$^+$02,DLC$^+$07]. Of course, as a general rule, adding data dependencies in general decreases opportunities for fragmentation.

To illustrate effects of additional data dependencies, we modify the example by introducing the alternative scenario which includes two additional data dependencies represented with dashed lines in Figure 3. We therefore assume that A_3 calculates some value Z that A_4 uses, and, in the scope of the structured activity a_4, we assume that $A_{4,1}$ calculates some value W that $A_{4,2}$ uses. The introduction of these dependencies in the logic program representing orchestration is displayed in Figure 10.

Figure 11 shows the annotated orchestration with the additional dependencies. Opportunities for rearranging activities in A^* are now much smaller, because essentially the only thing we can do is to swap A_1 and A_2. Also, in the body of the foreach loop A_4, the two activities have to be put strictly in sequence $A_{4,1}$, $A_{4,2}$.

The same dependency mechanism can be used to ensure preservation of conversational protocol between the orchestration and its partner services under fragmentation. To ensure sequencing of messages sent to or received from a partner, we can introduce the appropriate ghost logical variables for the two unidirectional message channels, which can be collapsed to a single bidirectional channel in case of strictly synchronous messaging. Every message dispatch and reception is then modeled in the same way as when modeling updating of an orchestration variable, by placing the appropriate input ghost variable(s) into the $\overline{R}$ set of a messaging activity, and the corresponding "updated" ghost variable(s) into its $\overline{W}$ set.

3.6 Applications to Resource Sharing and Security

In this section we want to highlight that the same techniques we have presented so far (independence analysis based on sharing analysis) can be straightforwardly use to uncover at least two situations which can hinder some cases of fragmentation.

On one hand, the use of shared external resources (such as, e.g., other services or databases) being accessed out-of-order can make the resulting fragmented process not to have the same behavior as the original one. While one may anyway want to assign these accesses to different fragments, being aware of their existence is of utmost importance in order to explicitly synchronize their acesses. This, which is always cumbersome for a human, can be automated with some ease when the fact that the same resource is accessed in two different points. It is however more involved in the cases where a reference (e.g., a URL) to the accessed services is passed around as part of a data structure. We want to note that passing data structures with many references around is the norm in RESTful services. Detecting that this is the case (i.e., there is variable sharing in two different points assigned to different fragments) can in principle be done with the techniques we have proposed.

On the other hand, it is very possible that when fragmenting for e.g. outsourcing one wants to prevent information leak: pieces of information which are handed over to some party must not, in any case, be seen by some other party. The only wayt to ensure that this is not the case is to make sure that there is no way for this to happen in the composition code. The possibility that this information leaking happens can be uncovered by studying variable sharing as well: if the data structures which are sent to the different fragments are determined as not having sharing, then information definitely cannot leak from one party to the other using this data structure. If there is a possible sharing, the composition control and data flow has to be studied more closely to determine if this is the case.

4 Conlusions and Future Work

Sharing-based independence analysis can be used to detect fragmentation opportunities in the context of orchestrations (service orchestrations). By representing a orchestration in the intermediate notation of a logic program, we are able to apply analysis techniques that detect data-sharing dependencies between orchestration activities, where the state of the executing orchestration, resources accessed, and partners are modeled by logical variables. Depending on the focus and objective of the analysis, such as reuse, security, resource access control, etc., orchestration can be represented in different ways that model specific notions of sharing.

In particular, the analysis can be used to identify orchestration fragments by using automatic parallelizing annotations on top of the dependency analysis. At design time, such fragment identification can be user or design criteria driven to produce sets of fragments arising from separating a selection of activities from a orchestration design. At runtime, fragment identification based on parallelization annotations informs a distributed orchestration enactment system when to distribute execution of independent orchestration parts.

Our future work will be aimed at correspondence between the original orchestrations represented in frequently used abstract and executable notations (Petri net based, BPEL, etc.), intermediate representation of a orchestration and the annotated results of the analysis, with the goal to enable automated fragmentation analysis of orchestrations in these notations and presentation of results in a form that is suitable for design and implementation needs.

References

[BMM06] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards Distributed BPEL Orchestrations. *ECEASST*, 3, 2006.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[BS92] N. Baker and H. Søndergaard. Definiteness analysis for clp(∇). Technical report 92/25, Univ. of Melbourne, 1992.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[CCH07] A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 138–153, The Technical University of Denmark, August 2007. Springer-Verlag.

[CCH08] A. Casas, M. Carro, and M. Hermenegildo. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 651–666. Springer-Verlag, December 2008.

[DLC$^+$07] Xinguo Deng, Ziyu Lin, Weiqing Cheng, Ruliang Xiao, Lina Fang, and Ling Li1. Modeling Web Service Choreography and Orchestration with Colored Petri Nets. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2007.

[FMB00] F. C. Filho, P. Maciel, and E. Barros. Using petri nets for data dependency analysis. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'2000), 8-11 October 2000, Nashville, TN*, volume 4, pages 2998–3003, 2000.

[HBC$^+$08] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Jose Meseguer Pierpaolo Degano, Rocco De Nicola, editor, *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

[HK03] J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, January 2003.

[HPBLG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[JEA$^+$07] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, BEA, Intalio, Individual, Adobe Systems, Systinet, Active Endpoints, JBoss, Sterling Commerce, SAP, Deloitte, TIBCO Software, webMethods, Oracle, 2007.

[KKL07] Oliver Kopp, Rania Khalaf, and Frank Leymann. Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes. Technical Report 2007/04, Institut für Architektur von Anwendungssystemen, Universitätsstraße 38, 70569 Stuttgart, Germany, November 2007.

[KL06] R. Khalaf and F. Leymann. E Role-based Decomposition of Business Processes using BPEL. In *IEEE International Conference on Web Services (ICWS'06)*, 2006.

[LS02] Vitaly Lagoon and Peter Stuckey. Precise pair-sharing analysis of logic programs. In *Principles and Practice of Declarative Programming*, pages 99–108. ACM Press, 2002.

[LWC$^+$02] Dongsheng Liu, Jianmin Wang, Stephen C. F. Chan, Jiaguang Sun, and Li Zhang. Modeling workflow processes with colored petri nets. *Comput. Ind.*, 49(3):267–281, 2002.

[MaBP02] Massimo Mecella and Francesco Parisi Presicce an Barbara Pernici. Modeling E -service Orchestration Through Petri Nets. In *Technologies for E-Services*, volume

2444 of *Lecture Notes in Computer Science*, pages 109–134. Springer Verlag, July 2002.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. Technical report 93/7, Univ. of Melbourne, 1993.

[MWL08] Daniel Martin, Daniel Wutke, and Frank Leymann. A Novel Approach to Decentralized Workflow Enactment. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 127–136, Washington, DC, USA, 2008. IEEE Computer Society.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. Second Ed.

[Obj09] Object Management Group. *Business Process Modeling Notation (BPMN), Version 1.2*, January 2009.

[OVvdA$^+$07] Chun Ouyanga, Eric Verbeekb, Wil M.P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, April 2007.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[TF07] Wei Tan and Yushun Fan. Dynamic workflow model fragmentation for distributed execution. *Comput. Ind.*, 58(5):381–391, 2007.

[vdA03] W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, Jan/Feb 2003.

[vdABtHK00] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced Workflow Patterns. In *CoopIS*, pages 18–29, 2000.

[vdAP06] Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[vdAtHW03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business Process Management: A Survey. In Wil van der Aalst, Arthur ter Hofstede, and Mathias Weske, editors, *International Conference on Business Process Management (BPM)*, volume 2678 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

[Wor08] The Workflow Management Coalition. *XML Process Definition Language (XPDL) Version 2.1*, 2008.

[WRRM08] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features - Enhancing flexibility in process-aware information systems. *Data Knowl. Eng.*, 66(3):438–466, 2008.

[ZBDtH06] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *OTM Conferences (1)*, pages 145–162, 2006.

A.4 Towards Runtime Migration of WS-BPEL Processes

Authors:

UniHH: Sonja Zaplata

UniHH: Kristian Kottke

UniHH: Matthias Meiners

UniHH: Winfried Lamersdorf

- To appear in: Proceedings of the 5th International Workshop on Engineering Service-Oriented Applications. WESOA'09. In conjunction with ICSOC-ServiceWave. November 23, 2009. Stockholm, Sweden.

Towards Runtime Migration of WS-BPEL Processes[⋆]

Sonja Zaplata, Kristian Kottke, Matthias Meiners and Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
`[zaplata|3kottke|4meiners|lamersd]@informatik.uni-hamburg.de`

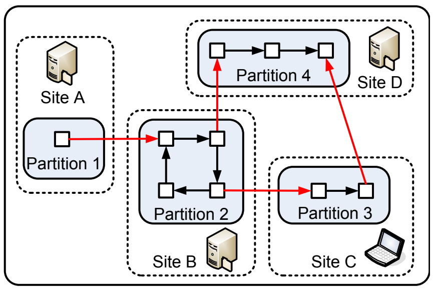
Abstract. The decentralized execution of business process instances is a promising approach for enabling flexible reactions to contextual changes at runtime. Most current approaches address such process distribution by *physical fragmentation* of processes and by dynamic assignment of resulting static process parts to different business partners.
Generalizing that in order to enable a more dynamic segmentation of such responsibilities at runtime, this paper proposes to use *process runtime migration* as a means of *logical process fragmentation*. Accordingly, the paper presents a general migration metadata model and a corresponding basic privacy and security mechanism for enhancing existing process models with the ability for runtime migration while still respecting the intensions and privacy requirements of both process modelers and initiators. In addition, the approach is conceptually evaluated by applying it to WS-BPEL processes and comparing the results to the general concept of process fragmentation.

1 Motivation

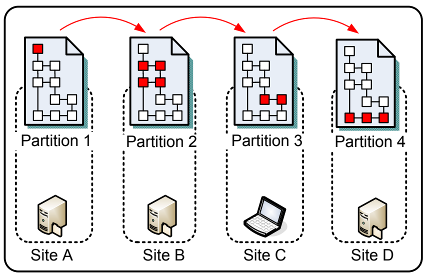
In todays networked business environments, cross-organizational collaborations composing complementary services and thus realizing new, value-added products gain increasing importance. As a technical representation of such business processes, executable workflows allow for flexible, dynamic and loosely-coupled collaboration among several business partners. The *Business Process Execution Language for Web Services (WS-BPEL)*[1] is currently one of the most relevant practical approaches. It allows for distributing resources such as employees, machines and services, whereas process control flow logic is typically executed by one single component at one single site [2].

However, due to the autonomy of participants, a single centralized process management system to control the execution of cross-organizational processes is often neither technically nor organizationally desired. As an example, required

⋆ The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).



(a) Process fragmentation



(b) Process migration

Figure 1. Process decentralization variants

services and resources often cannot be accessed by a centralized process engine because of technological differences or due to security policies [3]. Furthermore, in some cases the location where a process fragment is executed is relevant to perform the required functionality or is necessary for judicial reasons, e.g. in the context of eGovernment. Related to this, other non-functional aspects such as execution time, performance, navigation cost and capacity utilization can be optimized by load balancing and thus improve flexibility and scalability of participating systems [4]. If e.g. subsequent steps of a process are executed at a remote site, large data transfers can be avoided [2] and potential errors and resulting side effects can be handled more reliably, e.g. in the context of transaction management and compensation of interrelated activity blocks.

Most current research in the area of service oriented architectures is approaching decentralization of control flow navigation by a *physical fragmentation* of processes – splitting the overall executable process into several subparts which are then distributed to a number of available process engines (cp. figure 1(a)). In contrast to that, this paper proposes *process migration* as a means of *logical fragmentation*, fragmenting only the responsibilities for the execution of the process into a set of sub-responsibilities while preserving the original structure of the process description for all of the participating systems (cp. figure 1(b)). Such migration is the most "natural" way of executing a distributed process – as inherited by traditional human-based workflow management: A process is described in subsequent steps which are passed from one workplace to another, ensuring the specified task dependencies by sending the tasks to their respective executor when all requisite conditions are satisfied. Logical fragmentation by process migration has several advantages over physical process fragmentation:

- Process migration allows for fragmenting the responsibility to execute a process at runtime – depending on the availability of business partners or other contextual incidences. Furthermore, the granularity of fragmentation and the range of distribution can be selected on the fly by each executing participant.
- Coordination and merging of multiple process fragments is not necessary in the case of sequential execution. Global variables, scopes, errors and trans-

actions are easier to handle, because all these aspects of the process (i.e. data and control flow) are available to all executing parties. Thus, there is less communication and coordination overhead.

- Process migration is applicable to modern distributed systems including mobile devices because it does not depend on a single centralized system and allows for dynamic sharing of restricted resources [3,5].

However, process migration has also some drawbacks and still includes some interesting challenges which this paper proposes to address. First, the process description needs to implement a formal or technical model to communicate the current state of the migrated process instance. To preserve interoperability, this model should not require modifying the original business process [2]. Second, an important motivation for physical process fragmentation is given by the resulting separation of process fragments. If the process is to be fragmented for privacy reasons, process migration lacks proper security mechanisms in order to protect private information carried within the process. Third, if activities within the process should be executed in parallel, process migration alone is not sufficient, but rather process replication is needed in order to split up parallel tasks and allow load-balancing by running them on different machines.

This paper presents an approach to enhance existing processes with non-intrusive migration metadata and an overall system architecture to support run-time process migration among cooperating process execution systems. Therefore, we analyze which information has to be attached to the process at designtime in order to execute the (logical) process fragments as it was originally intended by the designer or the initiator of the process in whole. Furthermore, we present an initial privacy mechanism to protect the migrating process instance against unwanted changes and unauthorized access. Due to space limitations, parallel execution will not be fully covered in this paper, but outlined briefly. Finally, the approach presented here is applied to WS-BPEL and a respective prototype implementation, before the paper concludes with a short summary.

2 Related Work

Distributed and decentralized process execution becomes increasingly important and, consequently, many such approaches demonstrate the relevance of this research (cp. [6] for a brief overview). A first possible solution for distributing the control flow of a process is to change the service granularity. The activities which should be outsourced are wrapped, encapsulated behind a new service interface and the remaining process model is changed accordingly. A respective approach for WS-BPEL processes is presented by Khalaf and Leymann [7] providing sophisticated concepts to split and distribute specific WS-BPEL elements (e.g. scopes, loops and alternatives). Similarly, the approach of Baresi et al. [8] proposes a distributed service orchestration in WS-BPEL based on partitioning rules and process fragmentation by introducting corresponding invoke/receive activity pairs. However, process fragmentation is carried out at design time and is realized by weaving additional activities into the resulting fragments in order to realize a standard-compliant communication between them at runtime.

Another general approach is to split the original process, deploy the resulting parts at the desired system and induce *choreography* between the separated processes. A choreography-based process management system targeted at dynamic environments is represented by *CiAN* [9]. However, choreography and process fragmentation need a joint preparation phase for the physical distribution of each (sub-)process where all participating parties have to be available. Therefore, this approach is more advantageous in case of a similar recurrent execution of the same process than for spontaneous reactions to (unfrequent) ad-hoc changes.

As also criticized by Martin et al. [2] both solutions imply heavy changes in the original process model and additionally require the introduction and maintenance of new services. Thus, on the one hand, these unnecessary changes to the original process model are not motivated by the original business process, but by infrastructural constraints [2]. In consequence, the authors propose a non-intrusive approach for process fragmentation and decentralized execution. Here, fragmentation is achieved by transforming the orchestration logic represented in WS-BPEL into a set of individual activities which coordinate themselves by passing tokens over shared distributed tuple-spaces. Decentralized process execution has also been considered in Mentor [10] by partitioning a process based on activity and state charts. Addressing more dynamic environments, the approach of *MobiWork* [11] realizes mobile workflows for ad-hoc networks and is focused on the allocation of tasks to mobile participants also using process fragmentation to generate "sub-plans".

However, all presented approaches support at most dynamic allocation and assignment on the basis of a *static* fragmentation. All fragments and responsible parties are determined either at design time or once after invocation but mostly before executing the first activity of the process instance. Considering long-running processes, this flexibility may not be enough in order to also allow reactions to spontaneous contextual changes. In contrast, a *dynamically continuable* runtime segmentation implies that fragments and responsible parties are determined dynamically according to the current context and with respect to previous results and requirements of upcoming activities during the actual execution of the process instance.

A way to address such dynamic behavior is based on runtime migration of entire process descriptions. Migrating workflows as a basic concept for process automation have been introduced by Cichocki and Rusinkiewicz [12] in 1997. More recently, the framework *OSIRIS* [13] relies on passing control flow between distributed workflow engines in order to execute service compositions. Process data is kept in a distributed peer-to-peer-database system which can be accessed from each node participating in the process execution. In *Adept Distribution* [14] a similar approach to process fragmentation and decentralized execution is presented which supports dynamic assignment of process parts to so-called *execution servers*. The control of a particular process instance migrates from one execution server to another and the next participant is dependent on previous activities which are able to change the participant to execute the next partition. Related to this, Atluri et al. [4] present a process partitioning algorithm which creates self-describing subprocesses allowing dynamic routing and assignment.

Process migration has also particularly been applied to the area of mobile process execution, e.g. by Montagut and Molva [5]. Their approach relies on passing control flow between distributed WS-BPEL engines and addresses security on an application level by integrating a public/private process model in order to access applications internal to mobile devices. However, such a solution represents a choreography-like approach which only uses process migration in order to hand-over control flow – and thus also has some of the aforementioned disadvantages. The last example is the *DEMAC* middleware [3] which is able to delegate process execution (in whole or in part) to other stationary or mobile process engines. Its restriction to a proprietary process description language is, however, an obstacle to migrate existing business processes and to integrate standard process engines of external parties.

Complementing existing migration approaches, we therefore introduce a more technology-independent migration model which can also be applied to existing WS-BPEL processes. The following section presents a respective concept while considering the above-mentioned requirements for logical process fragmentation.

3 Process Migration Model and Decentralized Execution

There are at least two ways for enabling a process instance to migrate to other systems at runtime: One is to weave the migration information into the existing process model (*intrusive migration information*). This can e.g. be realized by inserting migration activities or migration scopes which determine to invoke other process engines using the remaining process description as an input parameter. Alternative paths or loops can optionally specify the distribution to potential migration partners and handle situations where migration fails. Although such an approach could be realized by using the standard WS-BPEL constructs and is thus compatible to existing systems, it only provides low flexibility as migration activities have to be planned in advance (i.e. at design time). Furthermore, this approach requires the original business process instance to be changed which often results in an unwanted mix of business logic and technical execution logic [2]. Compared to physical process fragmentation, there are thus only few advantages.

An alternative is to apply *non-intrusive migration information.* Technically this can be realized by an additional document holding the migration metadata or as a non-modifying annotation of the process description. Apart from the advantage that business logic does not have to be modified, non-intrusive process migration is possible after each activity and the decision about follow-up process engines can be made dynamically at runtime. The general methodology of non-intrusive process migration is depicted in figure 2. The development starts with the original modeling of the underlying business process which produces a process model, specified in an executable process description language such as WS-BPEL (step 1). Optionally in step 2, this process model can now be enhanced by a

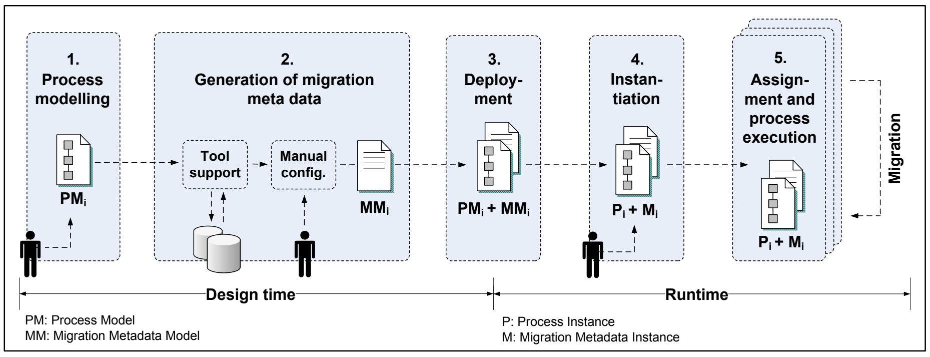


Figure 2. Process migration: methodology

migration metadata model which holds all information required for migration. In the following, process model and migration model are deployed (step 3) and can be instantiated by an application or a user (step 4). If required, parameters are passed to customize the process (i.e. normal invocation parameters) or the migration model. The latter is advantageous if the initiator is allowed to influence non-functional aspects about the way a process is executed (e.g. if the user pays for a higher service quality, the selection of migration partners is influenced accordingly). After that, the resulting process instance is executed following the guidelines of the associated migration metadata. However, if migration metadata is omitted or migration is not supported, the unaffected process can still be deployed and executed the usual (centralized) way.

The remainder of this section focuses on the second step of this methodology, i.e. the identification and description of the migration metadata model (cp. section 3.1) and the necessary enhancements to integrate basic privacy mechanisms (cp. section 3.2). An architecture to deploy and execute the migratable process is outlined in section 3.3.

3.1 Migration Model

The proposed migration model and its relationship to general process elements is depicted in Figure 3. As a starting point, we assume a common minimal process model consisting of a finite number of *activities* representing the tasks to be fulfilled during process execution, and a finite number of *variables* holding the data which is used by these activities. Activities can represent a specific task (*atomic activities*) or a control flow structure as a container for other activities (*structured activities*). Furthermore, variables can be specified on process level (*global variables*) or at activity level (*local variables*). Optionally, variables can contain an *initial value* which is assigned at design time.

A process description complying to these properties (e.g. XPDL [15] or WS-BPEL) can be enhanced by migration metadata documenting the execution state of the process (*process state*) and of each activity (*activity state*), such that the

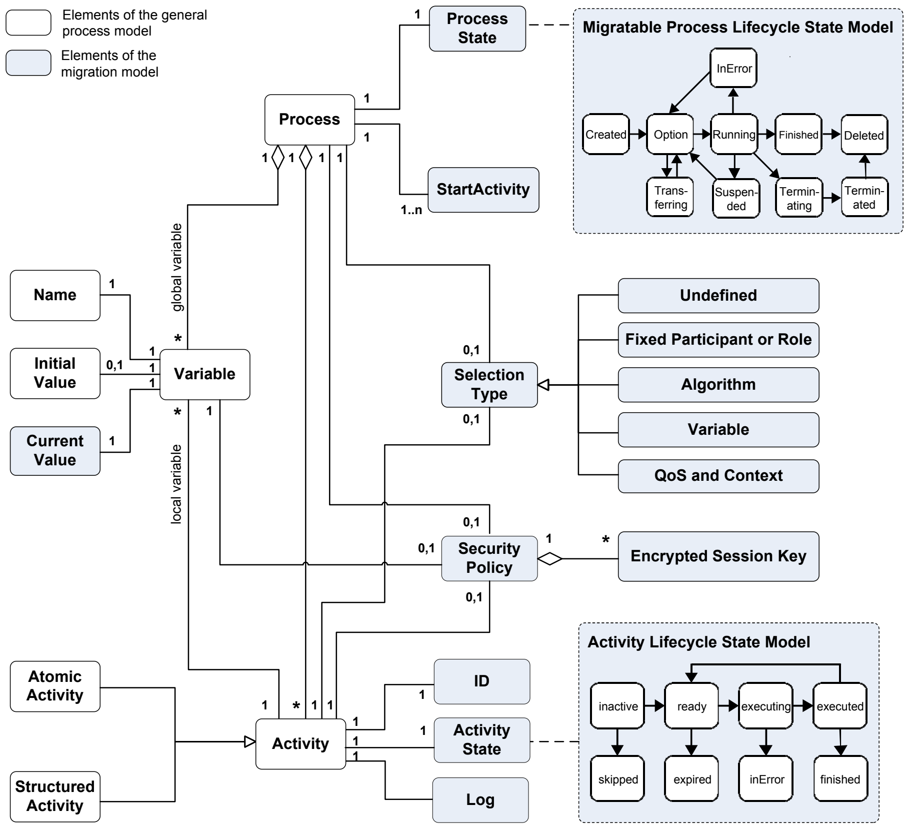


Figure 3. Overview of the migration model

progress in processing the activities is well-defined and visible for every participating device at any time during execution. The process state can take a value from the *migratable process lifecycle model* [3] as depicted in the upper right corner of Figure 3. As long as an activity can be executed at the local process engine, there is no need to search for another execution partner to accomplish this task. Consequently, the process is not transferred before all of the currently executed atomic activities are completed which preserves the process's consistency and integrity of its data. Avoiding to split up such atomic tasks, the safe state *Option* defines a stable point to transfer a process during its execution. In contrast, the process is regarded to be in the state *Running* if activities are in the state *executing*. Other states are used for the administration of the process, e.g. to keep it for logging purposes or to denote an error. The state of each activity is represented by an element of the *activity life cycle state model* which is based on the established lifecycle model presented by Leymann and Roller [16].

In addition to that, a set of activities can be referenced as *startactivities* to mark the first activity to be executed after process migration. The model allows for multiple startactivities in case the order in which the activities have to be executed is irrelevant or the activities should be processed in parallel. The indication of a start activity requires each activity to have a unique identifier (*ID*) in order to describe a pointer to this activity. Besides the state of the process and its activities, also the state of the variables have to be documented. As process migration is unable to cope with applications which keep part of their state externally, e.g. data stored in an external database, the *current value* has to be copied and attached to the migration data.

Up to this point, basic migration metadata can be generated automatically, i.e. by setting the process to the state *created* and all activities to *inactive* (cp. first part of step 2 in figure 2). If variables have been specified with an initial value, the given value is set as the current value of the variable.

However, the process modeler or the actual initiator often wants to influence the way the distributed process is executed. If the process is going to be migrated, one of the most important questions is, where the execution of the upcoming activity should be performed. Furthermore, additional data has to be transported to enable security and traceability of the process. As this could be determined by various (application-dependent) aspects, the following extensible migration model elements can be specified by the process initiator (cp. Figure 3): The *selection type* determines which strategy is used to assign an activity to a specific process engine. If the selection type is *undefined* (default) the process engine which is currently working on the process instance decides about further migrations. Thereby, it is able to shift processes to other engines which e.g. have access to required resources or which operate at a better performance. In contrast, the type *fixed participant or role* determines that a specific executing entity (e.g. a human or a concrete process engine) or a subject of a defined group of such entities (e.g. a process engine belonging to the role "bank") has to execute the process or a specified set of activities. More dynamically, as proposed by [14], the next participant can also be picked from a *variable* within the process description itself. If no such entities should be specified, but the participant should be selected as a result of a computation (e.g. picking the process engine which can execute as much of the process as possible), the respective *algorithm* is referenced. Finally, the selection can be based on specific *quality of service or context* requirements such as current workload or geographical location. Associated information about entities, algorithms or non-functional criteria can be included as an additional entry in the migration metadata or can be referenced (e.g. a URL). Attributes which are attached to process-level apply to all included elements, i.e. activities and variables. However, such attributes can be overwritten by local attributes on activity-level. This allows for specifications such as "all participants should be selected according to the quality-of-service aspect X, but the performer of activity n must be the fixed participant P". Finally, the process modeler can specify which kind of additional data should be collected during process execution, e.g. which participant has actually executed which subset of the process. These requirements and respective collected data can be described in the activity-related *log*.

Since procedures to allocate and select suitable participants depending on a given set of tasks in decentralized environments have already been established (e.g. [3,9]), the specification of selection algorithms is not part of this paper. Instead, the next subsection focuses on the required privacy of critical process parts to establish a basic model for the *security policy* of the presented migration model.

3.2 Privacy and Security Considerations

During decentralized execution of a process, its entire information is generally public to subjects which potentially belong to foreign organizations. This may not be acceptable, because the process description often contains private data (e.g. credit card information), private control flow information (e.g. existence of customer complaints), or identities of persons and companies which must not be revealed to or modified by other (external) parties. As another security risk, malicious participants could try to modify parts of the process or the migration metadata. To prevent such privacy and security threats, the access to process data can be restricted to specified subjects or roles, as e.g. determined in the above mentioned selection types *fixed participant or role*. However, process modelers must be aware of the fact that applying such policies reduces the number of potential migration partners and thus again may restrict flexibility.

Figure 4 shows the general idea of "masking" critical parts of a process description in order to ensure that only dedicated participants can execute sensitive activities and access corresponding data. The approach assumes that potential business partners can communicate with each other without being eavesdropped. Thus, a basic cryptographic key infrastructure is required, such as PKI (Public Key Infrastructure) or subject-related shared keys. However, encryption of the actual process is more complex, primarily because most process description languages (such as also WS-BPEL) allow for the definition of global variables which can be referenced in several activities – and thus might belong to more than one participant. In consequence, these parts cannot be directly encrypted with the personal key of the authorized subjects. Alternatively, the encryption of the different parts of the process (i.e. activities, variables or even the whole process) uses different *session keys* which are only used once. A corresponding *security policy* of such an element therefore contains a number of symmetric keys (e.g. pk_1 and pk_2 in Figure 4). The procedure of key distribution is based on a concept which is derived from broadcast encryption [17] where the same encrypted content is sent to all receiving parties without the need for two-way authentication or authorization. In the approach presented here, the keys necessary for decryption are sent together with the protected content. These keys prevent unauthorized access to the content, but are also themselves protected by cryptography. In case of an existing PKI the entries are encrypted with the public key pub_i of the appropriate subject (cp. Figure 4) and can be unlocked with the private key prv_i. Accordingly, an entry for each authorized subject is created and added to the migration meta data of the protected process element. As the result of this step, only the legitimate receiver is able to obtain

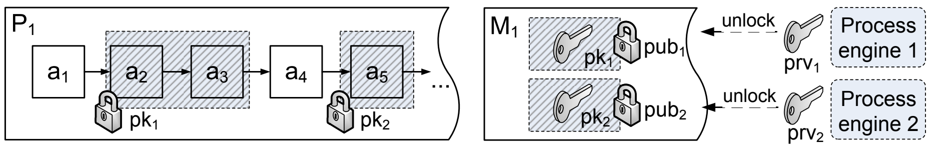


Figure 4. Process encryption and key distribution

the keys and decrypt the content and even encrypted global variables can be accessed by different authorized subjects using the same session key [18]. Neither an additionally interaction between the process initiator and the subjects nor an authentication is needed. As a positive side-effect, the use of unique session keys also increases the resistance of the cryptographic approach to attacks.

To additionally ensure the integrity of the process description, the process initiator is optionally able to generate a MAC (Message Authentication Code) for each security-related process part. Each peer provider owning the appropriate process key pk_i is thus also able to verify the integrity of this part. However, after a participant has modified a part of the process it has to generate a new MAC which confirms the integrity of this part. This possibility is indispensable because variables have to be changed by the subjects during process execution. In addition to the MACs, the process initiator can secure both the existence and the correct sequence of the process parts by a digital signature. In case of an existing PKI each subject can verify the correctness of the signature on the basis of the initiator's certificate, preventing e.g. a later modification of the process sequence. To also prevent replay attacks, an additional timestamp can be added to the signature. The integrity of the process description can be ensured by storing the digital signatures and the MACs within the migration document which finally has to be secured in a similar way as the process itself.

3.3 Execution

The architecture of a corresponding execution support is depicted in Figure 5. Considering the first layer, all potential participants have to provide a compliant interface in order to receive process descriptions from preceding process engines, e.g. represented by a WSDL description. By encapsulating the existing platform and exposing its functionality of cooperative process execution "as a service", the concept of process migration can be embedded into existing system infrastructures. Thus, the interface can be realized by using e.g. a standard web service which receives the process description (P_i) optionally supplemented with migration data (M_i) as an input parameter and returns the identifier of the process and the performer's signature in order to acknowledge its receipt. The service can furthermore be published at a public registry, so the service can be found and invoked dynamically whenever a migratable process is initiated.

If security mechanisms have been applied, a simple *privacy manager* is responsible for decrypting and encrypting the process and relevant parts of the

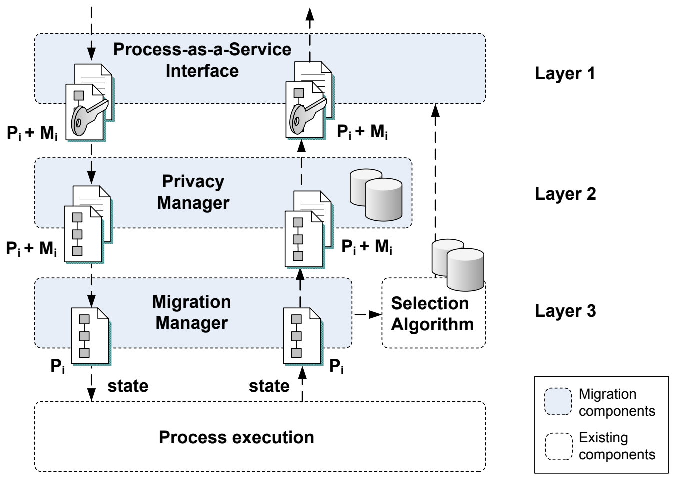


Figure 5. Runtime migration support

migration data (layer 2). Encryption of protected process parts can be realized by common procedures such as AES (Advanced Encryption Standard). In the case of WS-BPEL which is described in XML syntax, the specifications *Xml Encryption* and *Xml Signature* by the W3C can be utilized to tag encrypted parts and ensure integrity of the migrating process description. However, concerning the "masking" of processes, it has been found that encrypted parts are often causing errors during process execution because the process engine tries to interpret encrypted variables and activities but does not find expected content, e.g. encrypted variables do not match the expected data type. Thus, the privacy manager is also responsible for exchanging non-assigned encrypted parts by temporary dummy variables or activities. As encrypted process parts are not required to actually execute the assigned parts as defined by the security policy, this does not influence process execution at the local site.

The *migration manager* interprets the migration data as specified in section 3.1. It is responsible for passing the given process to the process engine, to update process states, activity states and log files subsequent to execution, and, if necessary, to determine the next process participant according to the given selection type specified for the upcoming activity – potentially making use of existing selection algorithms (layer 3). Considering the integration of the prototype system, it is desirable to completely avoid modifications on existing process execution systems. However, it shows that the underlying process engine has to implement an additional interface for receiving management instructions from the migration manager and for generating events in case of state changes. As most modern process engines already implement a general management interface (such as e.g. the *ActiveBPEL* Management API), migration manager and process engine can be sufficiently decoupled and the modification effort can be limited to a respective adapter component.

4 Migrating WS-BPEL Processes

WS-BPEL is a block-structured XML-based process description language which allows composing web services. According to the WS-BPEL 2.0 specification by OASIS [1] it is essentially comprised of two kinds of activities: Basic activities for web service interaction (*invoke, receive, reply*), basic control flow activities (*empty, wait, exit, throw, rethrow*) and activities for data manipulation (*assign*). Structured activities are used to compose the basic activities and define control-flow dependencies between them (*sequence, if then else, pick, flow, while, repeat until, for each*). Based on this characterization, the activities have been assigned to the elements of the general model in figure 3. In addition, Table 1 shows the result of the analysis which was performed in order to evaluate to which extent WS-BPEL processes can be migrated at runtime. Furthermore, the table shows a comparison to physical process fragmentation and summarizes the following discussion on advantages and disadvantages of both approaches.

Considering atomic activities, it shows that WS-BPEL has a very interactive character which makes the distribution of the control flow logic (both for migration and for fragmentation) more difficult. The *invoke* activity initiates the invocation of a web service which is specified within the process description (or references associated parts such as WSDL files) in either an abstract or a specific way. Thus, migration of a process containing an unprocessed *invoke* activity is not only possible, but even advantageous if the required service is not reachable from the current system. In case of a synchronous service call (*request-response* pattern) the receipt of the response message is part of the atomic activity. In case of asynchronous messaging, sending an associated *reply* subsequent to a migration is also not critical as the required information about the receiver (e.g. its physical address) is logged. Nevertheless, receiving a reply (*receive*) requires the specification of a specific participant because the sender of the reply has to know where to send the message. Thus, flexibility of arbitrary distribution is – in this case – limited both for migration and for physical process fragmentation.

The assignment of a variable (*assign*) is not a problem as the current value is stored within the migration metadata. The same is true for *wait*, *empty* and *exit* activities as these have a rather simple behavior. Notifications about faults are also uncritical as in case of process migration all the relevant information for fault handling (i.e. *scopes*, *fault handler*, *compensation handler*) are available to the executing party. If required, the occurrence of faults can also be documented in the log, e.g. if the control flow logic has to return to the failed activity when fault handling is finished. Considering process fragmentation, other process fragments may have to be notified in case of a fault, resulting in an increased coordination overhead.

As indicated above, migration must not happen while an atomic activity is in the state *executing.* However this does not apply for structured activities which only act as a container for other activities. As a consequence, structured activities such as *sequence* or *while* do not have to be finished in order to allow the migration of the process instance. This is another advantage over physical process fragmentation where e.g. loops often have to be distributed as a whole:

	WS-BPEL elements	Process migration	Process fragmentation
Atomic activities	invoke	possible	possible
	reply	possible (log)	coordination required
	receive	fixed participant	fixed participant
	assign	possible	possible
	wait, empty, exit	possible	possible
	throw, rethrow	possible (log)	coordination required
Structured activities	sequence	possible	possible
	if then else	possible	unnecessary fragments
	while, repeat until, for each	possible	coordination required
	pick	possible	coordination required
	flow	coordination required	coordination required
Other elements	scope	generally available	coordination required
	fault handler	generally available	coordination required
	compensation handler	generally available	coordination required
Dead path elimination	-	automatically	requires coordination
Privacy of process parts	-	artificial	fulfilled
Splitting atomic activities	-	forbidden	no known approach
Data replication	-	for parallel execution	variables, scopes, events
Design time distribution	-	possible	possible
Runtime distribution	-	during execution	once after invocation

Table 1. Migratable WS-BPEL processes and comparison to the general concept of process fragmentation

By storing the current value relevant for the evaluation of the loop condition in the migration data, migration is even possible within iterations. If the condition has to be evaluated only once (such as in the case of *if then else*) the selected branch is determined by the process's startactivity. In case of process fragmentation, fragments and responsible parties are often determined at design-time or at invocation-time. If a process's transition condition restricting access to parallel or exclusive paths is evaluated at runtime, some of the process fragments and their respective assignments of executors may never be used. Thus, process migration is more efficient because it allows integrating the current state of variables at runtime in order to make its assignments. Related to this, the execution of a necessary *dead path elimination* [16] requires further coordination if process fragments are distributed physically. In case of process migration, the dead path can be processed automatically by setting all upcoming activities (until the next join condition) to the *skipped* state. As this information is hold in the migration document, this does generally not involve communication with other participants.

The *pick* activity waits for the occurrence of an event from a set of events and then executes the activity associated with that event. If the process is fragmented physically, this is a problematic issue. Either all the necessary data has to be replicated (i.e. all event/reaction pairs) or the events have to be fragmented as well. If the reaction to an event affects other fragments, additional coordination is necessary. In case of process migration, this is not a problem as the whole spectrum of possible events and reactions is available to the responsible participant. If, furthermore, other activities are temporarily suspended because of the event, the activity states indicate where the execution must be continued. However, each process participant has to subscribe to each required event as long as it is responsible for the execution of the process instance. Thus, during migration time, there is a remaining risk that some events may be not be noticed.

The *flow* activity contains activities which should be processed in parallel. As long as the process is migrated to exactly one participant, migration within the execution of a flow is uncritical as the states of each included activity are well-defined. Nevertheless, the process cannot be transferred until all atomic activities have reached a stable state and thus may have to wait for long-running activities to be finished. Since the execution of parallel paths on a single machine cannot be considered as "real parallelism", a copy of the (entire) process can be distributed to different participants which are each responsible for the execution of one of the parallel paths. In order to synchronize parallel paths, there has to be a defined meeting point. In consequence, distributed parallel execution needs advanced coordination mechanisms for both migration and fragmentation. However, using replication instead of fragmentation allows for a local detection of shared variables and thus avoids unnecessary synchronizations.

Other interesting aspects discussed in Table 1 include privacy of process parts, specification of fixed participants and distribution flexibility. As a drawback for process migration, privacy can only be realized by artificially masking private process parts as proposed in section 3.2, whereas physical fragmentation of the process makes such mechanisms unnecessary. In consequence, the effort for developing migratable processes containing private parts is a little higher. Nevertheless, process migration allows for more flexibility in selecting the most suitable process engine at runtime while still allowing to respect the interests of the process designer by determining specific participants or selection algorithms. Thus, especially long-running distributed process instances benefit from the possibility to adapt the execution of control flow to changing conditions.

5 Conclusion and Future Work

This paper focuses on distributed process execution involving multiple engines in order to increase flexibility and to improve reactions to ad-hoc context changes. As an alternative to physical process fragmentation, a concept for realizing logical process fragmentation on the basis of process migration has been presented. Compared to physical fragmentation, process migration provides more flexibility by allowing to distribute running process instances at runtime while respecting the guidelines of the process modeler. On the other hand, privacy and security-related issues have to be considered explicitly as also addressed in this paper.

Future work includes the evaluation of other practically-relevant process description languages and the implementation of respective migration managers. A prototype system covering the proposed system architecture for XPDL and WS-BPEL processes has already been developed and shows basic applicability of the proposed concepts. Considering privacy support, WS-BPEL process designers must still be careful not to mask multi-level scopes when these are also relevant for public process parts. Therefore, a a tool to support process modelers when applying security mechanisms would be useful to facilitate the development of migration metadata and help process modelers to avoid unnecessary errors.

References

1. OASIS: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (2007)
2. Martin, D., Wutke, D., Leymann, F.: A Novel Approach to Decentralized Workflow Enactment. In: Enterprise Distributed Object Computing, IEEE (2008) 127–136
3. Zaplata, S., Kunze, C.P., Lamersdorf, W.: Context-based Cooperation in Mobile Business Environments: Managing the Distributed Execution of Mobile Processes. Business and Information Systems Engineering (BISE) **2009**(4) (10 2009)
4. Atluri, V., et al.: A Decentralized Execution Model for Inter-organizational Workflows. Distrib. Parallel Databases **22**(1) (2007) 55–83
5. Montagut, F., Molva, R.: Enabling Pervasive Execution of Workflows. In: Collaborative Computing: Networking, Applications and Worksharing, IEEE (2005)
6. Jablonski, S., et al.: A Comprehensive Investigation of Distribution in the Context of Workflow Management. In: ICPADS 2001. (2001) 187–192
7. Khalaf, R., Leymann, F.: A Role-based Decomposition of Business Processes using BPEL. In: IEEE International Conference on Web Services, IEEE (2006) 770–780
8. Baresi, L., Maurino, A., Modafferi, S.: Towards Distributed BPEL Orchestrations. ECEASST **3** (2006)
9. Sen, R., Roman, G.C., Gill, C.D.: CiAN: A Workflow Engine for MANETs. In: COORDINATION 2008, Springer (2008) 280–295
10. Muth, P., et al.: From centralized workflow specification to distributed workflow execution. J. Intell. Inf. Syst. **10**(2) (1998) 159–184
11. Hackmann, G., Sen, R., Haitjema, M., Roman, G.C., Gill, C.: MobiWork: Mobile Workflow for MANETs. Technical report, Washington University (2006)
12. Cichocki, A., Rusinkiewicz, M.: Migrating Workflows. In: Advances in Workflow Management Systems and Interoperability, NATO (1997) 311–326
13. Schuler, C., Weber, R., Schuldt, H., Schek, H.J.: Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In: ICWS. (2004) 26–34
14. Bauer, T., Dadam, P.: Efficient Distributed Workflow Management Based on Variable Server Assignments. In: CAiSE. (2000) 94–109
15. Norin, R., Marin, M.: XML Process Definition Language. Specification WFMC-TC-1025, Workflow Management Coalition (2002)
16. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. PTR Prentice Hall (2000)
17. Lotspiech, J., Nusser, S., Pestoni, F.: Broadcast Encryption's Bright Future. Computer **35**(8) (2002) 57–63
18. Bertino, E., Castano, S., Ferrari, E.: Securing XML documents with Author-X. Internet Computing, IEEE **5**(3) (2001) 21–31

A.5 Executing Parallel Tasks in Distributed Mobile Processes

Authors:

UniHH: Sonja Zaplata

UniHH: Kristof Hamann

UniHH: Winfried Lamersdorf

Submitted to:

- *Eighth International Conference on Pervasive Computing (Pervasive 2010).*

Executing Parallel Tasks in Distributed Mobile Processes*

Sonja Zaplata, Kristof Hamann and Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
[zaplata|hamann|lamersd]@informatik.uni-hamburg.de

Abstract. The practical relevance of business processes combined with an increased use of mobile devices and networks have lead to new research activities in the area of *Mobile Business Process Management (M-BPM)*. The distributed execution of such mobile processes allows for running several parallel tasks on different (mobile or stationary) devices. However, interdependencies with respect to the processes' data (i.e. data objects used in more than one parallel task) may lead to undesired results or require advanced coordination and synchronization mechanisms.
In order to address such issues, this paper presents a multi-level concept for supporting the execution of parallel tasks within distributed mobile processes. It introduces a specific model based on data replication and respective methods for detecting data dependency conflicts, assignment of application-specific data classes, decentralized coordinated execution, and synchronization of parallel process paths. In addition, it demonstrates the applicability of these concepts both by formal verification and practical integration of a respective prototype component into an existing mobile process management system.

1 Introduction

According to the paradigm of Service-Oriented Computing, recurrent business transactions as well as structured ad-hoc application tasks can be represented by appropriate business process models. The technical implementation of such executable structured business tasks is called a *service composition* or – more generally – a *process* which involves software services as well as manual or interactive human (sub-) tasks specified in an operational and executable way.

The integration of *mobile participants* opens up new ways to enhance business processes with functionalities which are either not available from stationary systems or simply uninteresting in a static environment. Examples for that include context-based services providing information about location, the perception of specific situations, or the usage of resources which are only accessible within a

* The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

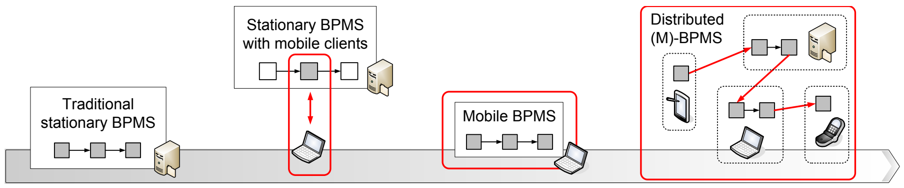

Traditional stationary BPMS
Stationary BPMS with mobile clients
Mobile BPMS
Distributed (M)-BPMS

Figure 1. Evolution of mobile business process management systems: Increasing involvement of mobile process participants

local network. However, most mobile devices still have limited resources as, e.g. restricted computing power, memory capacity, and energy resources. In consequence, many mobile devices and applications are rather specialized and thus highly heterogeneous [1,19].

Current research dealing with mobile business process management systems can be classified into three main categories (cp. Figure 1): First appearances are based on traditional workflow systems which are mostly centralized and include mobile clients as participants of single tasks only. More advanced mobile systems allow for self-contained initiation and execution of whole business processes by integrated lightweight mobile process execution engines [6]. However, due to resource limitations and heterogeneity, latest approaches propose to execute more complex processes by involving several stationary and mobile devices which are able to select the most appropriate process participant dynamically, e.g. depending on required functionality and actual context [7,10,13,20].

Considering the advantageous distributed and decentralized execution of mobile business processes, especially the simultaneous execution of several parallel process paths offers new chances for resource-restricted mobile devices: Splitting up a process into distributed paths which run on the most appropriate device could increase performance by, e.g., load balancing or enhance the cooperation between different devices with complementary capabilities. On the other hand, dependencies within the processes' data used in more than one parallel path may lead to undesired results or require additional coordination and synchronization mechanisms – a particular problem which is addressed in this paper.

An example for a process spanning several mobile devices is depicted in Figure 2. It shows an extract from a distributed mobile process which collects environmental information. Because associated tasks are long-running, need different resources (potentially provided by different types of devices), and are interconnected they should preferably be processed in parallel. In detail, the first path P_1 involves input information collected manually by a mobile user (variable *information*). When the input is finished, the process variable *finished* is set and a web service is invoked to process the input data and compute the result (variable *result*). This service invocation costs some money which is withdrawn from the budget available for executing the mobile process (variable *balance*). Activities on the second path P_2 require a device with sensors for automatically collecting environmental information – such as positioning or temperature data (variable

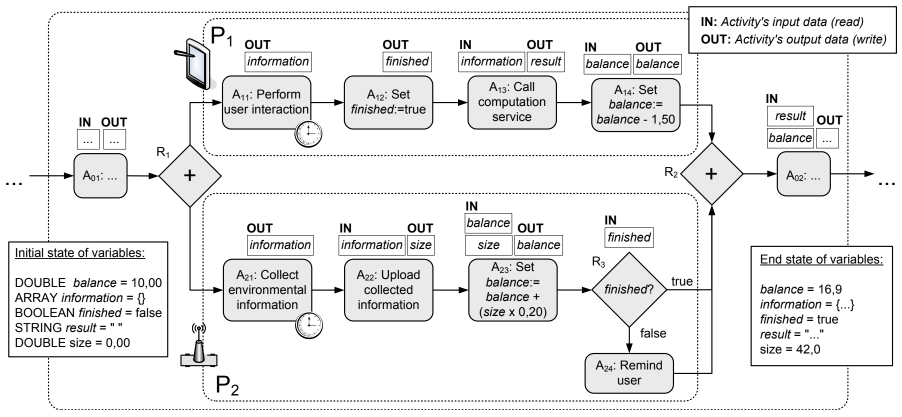


Figure 2. Example: Distributed execution of parallel mobile process paths

information) – and access to a service for sharing this information. Depending on the size of information uploaded to the service (variable *size*), the process owner is rewarded with some money, which is added to the process' budget (variable *balance*). If the user interaction on P_1 is still not finished when all the work of P_2 is done (variable *finished*), the responsible user should be reminded, e.g. by sending a short message.

In such a scenario, it is advantageous to split the responsibility for the parallel paths and select the most suitable mobile device depending on its specific capabilities and context at runtime. Such dynamic assignment can be realized by a context-aware selection procedure such as presented in [7] or [10]. If, for example, two suitable devices (e.g. a PDA and a wireless sensor) are chosen to execute one of these paths respectively, the following observations can be made:

- Parallel process paths may contain data dependencies, i.e. process variables defined in a global scope of the process are used within more than one parallel path. While read/read access of such a variable is not critical, write/read and write/write dependencies can lead to undesired results, e.g. lost updates. For example, the value of the variable *balance* is considered correct only if it holds the total amount of money considering data manipulations of both paths at the end of the process.
- The question of *correctness* and thus the need for examining conflicts caused by data dependencies is, in general, application-dependent. For example, let us assume that the variable *information* is only used within the parallel sections of the process and that, depending on the intentions of the process designer, the variable *must not* be synchronized before the subsequent activities directly using its values are finished.[1]

[1] Alternatively, the information gathered in both paths could have also been modeled as two independent variables; but in view of saving memory resources at least both variants can be justified.

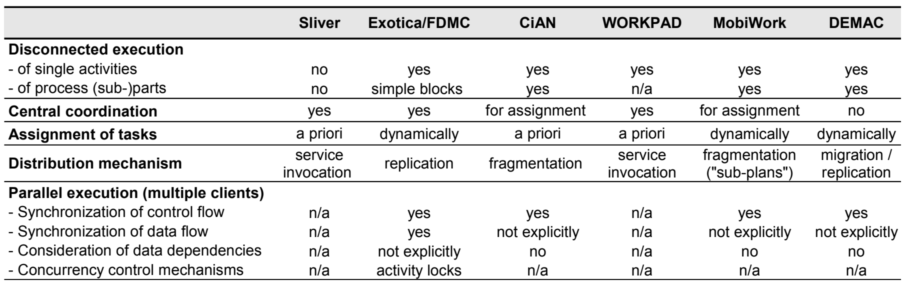
- In some cases, the interconnection of parallel process paths may be explicitly intended. For example, the variable *finished* constitutes such an interdependency between both paths. If *finished* is set to true by P_1, the reminder in P_2 is omitted. Whether parallel paths should be notified at once or only within a (given) period of time can have a considerable impact on the efforts for runtime coordination, and is again application-dependent.
- In mobile environments costs of data transfer and restriction of wireless communication range, coordination, and synchronization should be minimal. This means, in particular, that models and mechanisms supporting the execution of parallel process paths must explicitly consider that participating devices may be temporarily unavailable.

On such a background, the specific contribution of this paper is to support the correct execution of parallel process paths while respecting the desired behavior of the process as well as the specific characteristics of mobility and distribution. Accordingly, the rest of the paper is organized as follows: Section 2 analyzes existing approaches for parallel process execution. Based on that, Section 3 presents concepts of an approach to support modeling, control flow distribution and execution of parallel tasks in distributed mobile processes. Finally, these concepts are evaluated – both formally and practically – in Section 4, before the paper concludes.

2 Existing and Related Work

As motivated in Section 1 there are several approaches dealing with the management of mobile process clients and management systems. This section reviews them in more detail, especially regarding their support for parallelism. Furthermore, relevant preliminary work in the area of traditional concurrency control, data replication, and synchronization is summarized.

Figure 3 gives an overview of selected mobile process management approaches and summarizes their most important characteristics. First, approaches based on systems realizing monolithic process engines for mobile devices (e.g. *Sliver* [6]) do not explicitly integrate concepts for coordination of parallel paths because they do not consider executing these paths on different devices. In contrast, the *Exotica/FMDC* workflow management system [2] enables mobile clients to download single user tasks or simple sequential activity blocks in order to perform them without being connected to the central workflow server. The respective task and its data are replicated and locked until the mobile client reconnects. In consequence, parallelism between multiple mobile users can only involve those activities which do not share the same process variables. A choreography-based workflow management system targeted to mobile environments is represented by *CiAN* [20]. It supports distribution on the basis of process fragmentation and provides basic mechanisms for control flow synchronization. Nevertheless, parallel execution containing data dependencies and synchronization of data flow have not been explicitly considered. Related to this, the *WORKPAD* infrastructure [13] was designed to support human rescue teams in disaster scenarios, but



	Sliver	Exotica/FDMC	CiAN	WORKPAD	MobiWork	DEMAC
Disconnected execution						
- of single activities	no	yes	yes	yes	yes	yes
- of process (sub-)parts	no	simple blocks	yes	n/a	yes	yes
Central coordination	yes	yes	for assignment	yes	for assignment	no
Assignment of tasks	a priori	dynamically	a priori	a priori	dynamically	dynamically
Distribution mechanism	service invocation	replication	fragmentation	service invocation	fragmentation ("sub-plans")	migration / replication
Parallel execution (multiple clients)						
- Synchronization of control flow	n/a	yes	yes	n/a	yes	yes
- Synchronization of data flow	n/a	yes	not explicitly	n/a	not explicitly	not explicitly
- Consideration of data dependencies	n/a	not explicitly	no	n/a	no	no
- Concurrency control mechanisms	n/a	activity locks	n/a	n/a	n/a	n/a

Figure 3. Analysis of selected mobile business process management systems

still requires a central entity in order to coordinate mobile process participants. The approach of *MobiWork* [7] realizes mobile workflows for ad-hoc networks and is focused on the allocation of tasks to mobile participants. Parallel processing has thus not been discussed yet. The last example is the *DEMAC* process management system [9] which is able to delegate process execution (in whole or in part) to other mobile or stationary process engines using the concept of process migration. In case of parallel path execution, the process description can optionally be duplicated and transferred to other devices in order to continue execution of any of the parallel paths. But, nevertheless, data dependencies and synchronization of process variables have not been considered here either.

As summarized by Figure 3, most of the approaches focus on basic models and mechanisms to distribute and execute process control flow based on fragmentation or replication. None of them explicitly considers data dependencies among parallel process paths and thus integrates concepts for data synchronization and data concurrency control.

Regarding *distributed (non-mobile) workflow management systems*, current research concentrates on partitionizing processes to decentralize their execution. In fact, these approaches primarily cover the *Web Service Business Process Execution Language* (BPEL). Early systems take up the idea of replication and enforce the exchange (and thus the synchronization) of data values subsequent to each modification [15,18]. In most cases, these approaches most often assume a stable network connection between participating systems and do not consider the case of disconnected operations. In contrast to this straightforward idea, Martin et al. transform a segmentated BPEL process into an executable workflow net, which uses tuplespaces to synchronize shared variables [12]. A tuplespace is an instance of shared memory and for this reason rather inappropriate to use in mobile environments. The model of a so called pervasive workflow [14] can be realized with multiple BPEL processes, but parallel access to shared data has not been considered in this work. While Nanda et al. mention this problem in their work about how to partition orchestrations, they do not handle it further as they perceive such occurences as nondeterministic parallel programs [16], which is true but insufficient. Other approaches avoid troubles in fobidding parallel (write) access to shared variables [8] or in forbidding shared variables at all [3].

The main challenge for parallism in mobile processes is thus to deal with dependencies resulting from concurrent access to shared data. Respective problems of data replication, dependencies, and concurrency have already been investigated thoroughly in the research area of *transaction processing* providing a solid foundation for further research. Here, *correctness* is expressed as *serializability*, i.e. equivalency to any serial execution. For replicated data this criterion is extended to *one-copy serializability*, determining that the concurrent execution of transactions on replicas is *one-copy serializable*, if it is equivalent to a serial execution on non-replicated data [4].

So, a gap between both of these research areas is still open. Therefore, the contribution of this paper is to propose a concept for supporting the correct synchronization of distributed process data in the context of mobility. For that, the following section presents an approach of using *one-copy serializability* to develop a formal model of data flow dependencies as well as a problem-adequate coordination algorithm for this class of systems.

3 Distributed Execution of Parallel Process Tasks

As shown in the previous section, the overall aim on executing parallel tasks in distributed mobile processes is not only the correctness of the execution, but moreover the minimization of coordination efforts in terms of message exchange and ressource usage. Since serializability gives a generic view on correctness, the basic idea presented in this paper is to find a serial execution being equivalent to the actual execution, which is performed without coordination as far as possible. *Dependency conflicts* will be introduced as access to variables which nevertheless require the disadvantageous exchange of messages between involved devices.

The first level of this novel multi-level approach provides therefore a method to help process designers to detect unintended dependency conflicts within the process model already at design time. On the second level, *data classes* are introduced to allow process designers to specify the intended data dependencies more precisely and, thus, reduce the need for coordination between parallel paths which involve this data. These two steps involving the process designer are optional, but recommended in order to minimize coordination efforts at runtime and with this to accelerate the overall execution time. The following levels describe parallel execution at runtime, presenting procedures for distributing and synchronizing control and data flow as well as an optimistic approach to resolve remaining dependency conflicts.

First, a formal model is developed as a foundation in order to analyze a process' behaviour. To ease understanding, the introduced abstract meta model is limited to express the characteristics needed in this paper only.

3.1 Abstract Meta Model and Data Replication

To examine parallelism in more detail, the following generic meta model is introduced. It represents relevant parts of the process structure and abstracts from concrete process modeling languages and different ways to model parallelism.

Definition 1 (Precedences). *Let A_i and A_j be activities of a process. If A_i is connected to A_j so that A_i must be executed directly before A_j, then we write $A_i \lessdot A_j$ (direct precedence). $\lessdot$ must not contain any cycles.*[2] *The precedence relation $<$ is the transitive closure: $< := \lessdot^+$*

Definition 2 (Alternativity). *Let A_i and A_j be activities of a process. If there is a selective split, so that either A_i or A_j is executed, we write $A_i \otimes A_j$.*

Definition 3 (Parallelism). *Let A_i and A_j be activities of a process. We write $A_i \parallel A_j$ if not $A_i = A_j \;\vee\; A_i < A_j \;\vee\; A_j < A_i \;\vee\; A_i \otimes A_j$. The activities A_i and A_j are called parallel to each other.*

Definition 4 (Read and Write Sets). *We define $\mathcal{R}(A_i)$ as the set of variables which are read by activity A_i (input container) and we define $\mathcal{W}(A_i)$ as the set of variables which are written by A_i (output container). $\mathcal{V}(A_i) := \mathcal{R}(A_i) \cup \mathcal{W}(A_i)$*

Note that we call two activities to be parallel to each other although they are potentially executed alternatively. This is required if it is not possible to determine whether the activities will be executed parallel or alternatively, e.g. if a transition condition which restricts access to parallel paths is evaluated at runtime. Such control flow conditions are assigned to the according activity A_i. Therefore the variables which are used in a condition are included in $\mathcal{R}(A_i)$.

Since mobile environments suffer from unreliable or expensive communication, it is suggestive to use data replication for allowing mobile devices to work on a process as autonomously as possible. Figure 4 shows a process where each parallel path P_i has its own replica $\mathfrak{R}_i$ containing the data needed for the execution of the assigned activities. Splitting the control flow to parallel paths means replicating the current process state and transferring the replicas to the selected devices. While executing a parallel path, all read and write operations are thus initially executed on the local replica and the replicas are only merged when the control flow is finally synchronized. Consequently, during the parallel execution there are multiple replicas containing different values of each variable.

3.2 Detecting Dependency Conflicts

Concurrent access to shared resources has to be controlled to avoid unexpected behavior. In order to minimize coordination efforts for parallel process paths, it is essential to determine under which circumstances processes need concurrency control mechanisms for ensuring *correct* results. To evaluate the correctness of process execution we assume an activity to be an atomic unit of work [11] and thus a *transaction*. This allows to use classical transaction correctness criteria as summarized in Section 2. Regarding the correctness criterion of *one-copy serializability*, the concurrent execution of transactions on replicas is correct if it is equivalent to a serial execution on a non-replicated data set [4]. Thus the limiting factor is the need for an equivalent serial execution without replicas.

[2] Loops can be realized with a block construct

However, while database management systems have to cope with transactions in an unpredictable order, data access within processes mostly follows a fixed process model. Therefore it is possible to analyze this model and to identify potential conflicts a priori:

A trivial case is given if the output containers of all activities do not contain variables used by other parallel paths, i.e. $\mathcal{W}(A_i) \cap \mathcal{V}(A_j) = \emptyset$ for $A_i \parallel A_j$. The activities' output data is changed on the local replica respectively and at the end of the parallel execution the modified variables are simply merged. The execution is clearly equivalent to every possible serialization of the parallel activities.

A more difficult case appears if activities read variables which are modified by an activity in another parallel path. As reads and writes affect only the local replica, not every possible serialization is allowed. Figure 4 shows A_3 reading y which is also modified by the parallel activity A_2. During the actual execution (using the replicas), A_3 reads the value of y which was written by A_1. Accordingly, the serial execution $A_1A_2A_3A_4$ is not equivalent since here A_3 reads the value of y which was written by A_2. In real executions on replicas, this would be only the case if A_3 waits for a message from A_2 containing the actual value of y. Since one aim is to avoid messaging, such serial executions are preferred, in which A_3 appears before A_2. This fact is expressed in the following definition:

Definition 5 (Read-Write Dependencies). *Let A_R and A_W be activities of a process. We write $A_R \prec A_W$ if $A_R \parallel A_W$ and $\mathcal{R}(A_R) \cap \mathcal{W}(A_W) \neq \emptyset$.*

Beyond the precedence relation $<$, the relation $\prec$ likewise affects the order of activities in allowed serializations. To derive equivalent serial executions, we use a *dependency graph*, which integrates both relations:

Definition 6 (Dependency Graph). *A dependency graph is a directed graph $G = (V, E)$, whose vertices V represent the set of a process's activities which are actually executed and whose edges represent the dependencies between these activities: $E = \prec \cup \lessdot$.*

Figure 5 shows the acyclic dependency graph of the process from Figure 4. If the dependency graph does not contain any cycles, the topological order of the activities indicates equivalent serializations (i.e. $A_1A_3A_2A_4$). Consequently, there is no need for concurrency control if the dependency graph is acyclic.

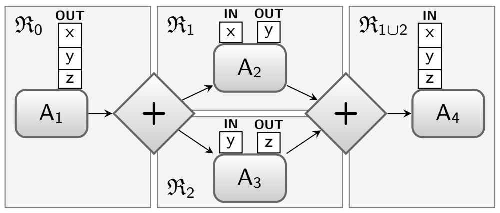


Figure 4. Usage of replication in a simple process with concurrent access to y

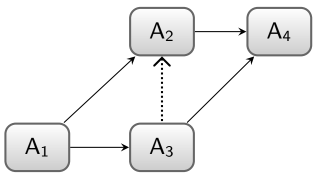


Figure 5. Dependency graph of the process in Figure 4

Respectively considering the example in Figure 2, the following read-write dependencies can be identified: $A_{13} \prec A_{21}$, $A_{14} \prec A_{23}$, $A_{22} \prec A_{11}$, $A_{23} \prec A_{14}$, $R_3 \prec A_{12}$. We call the resulting cycles *dependency conflicts*:

Definition 7 (Dependency Conflict). *A cycle in a dependency graph is called dependency conflict. The set $\mathfrak{C}$ contains all activities belonging to the cycle.*

If a dependency conflict exists, read-write dependencies or precedences in combination with read-write dependencies do not allow to build an equivalent serial execution of the process. Therefore, with respect to serializability, the distributed execution of parallel paths is not possible without communication between the respective replicas.

Until now, we have omitted the case where parallel writes change the same variable. Until the variable is not read on one of the paths later, the order of write operations in the serialization is irrelevant. While other approaches assume that $\mathcal{W}(A_i) \subseteq \mathcal{R}(A_i)$ (compare [5] and traditional locks), our approach uses the fact that processes clearly differentiate between input and output containers, i.e. $\mathcal{W}(A_i) \backslash \mathcal{R}(A_i) \neq \emptyset$ is allowed. In addition, writing the output container is atomic since there is no concurrent access to the respective replica. Consequently, there is no need for an artificial order of parallel writes unless there is a subsequent read on one of these paths. In the latter case, there would be a read-write dependency deciding the order of these operations.

Using these observations, process designers should analyze unintended dependency conflicts already at design time, and, if possible, refactor the process model as a first step to reduce coordination overhead at runtime. However, as dependency conflicts can only be derived from the process model, also conflicts could be detected which actually do not occur during runtime. This is primarily the case if alternative paths contain conflicts. However, if at runtime the states of all omitted activities are set to *skipped* and the new states are propagated to relevant devices, the updated dependency graph ignores this kind of activities and this dependency conflict does not appear anymore.

3.3 Data Classes

Due to its generality, serializability is a useful and reasonable correctness criterion. However, in some cases it is too restrictive, which results in unnecessary or even unwanted synchronizations. Since loosening serializability is application-specific and the process model lacks semantical information, we introduce *data classes* as a generic approach to express the process designer's intentions. Inspired from [21], data classes specify application-specific guarantees concerning the consistence of the used data and differ in the method to deal with dependency conflicts. In consequence, the process designer can select the most suitable data class for potentially conflicting process variables in order to further reduce the need for runtime coordination. In order to illustrate this idea, we briefly introduce three possible data classes. In general, also many other data classes are possible.

Serialized. This first and mandatory data class is the strongest class – leading to serializability by default for ensuring correct process execution if data class selection is omitted. Accordingly, every dependency conflict that results from variables of this data class has to be resolved by concurrency control. Considering the example in Figure 2, the process variable *balance* is a possible candidate for this data class.

To resolve a dependency conflict, a read-write dependency $A_R \prec A_W \in \mathfrak{C}$ is chosen. The needed coordination is introduced by adding the new precedence $A_W \lessdot A_R$ to the process model. This new precedence can not cause a new cycle, since otherwise $A_R < A_W$ had to be hold. In fact, it is $A_R \prec A_W$ which is a contradiction. After modifying the process model, read-write dependencies and the dependency graph have to be recomputed, because it is now $A_W < A_R$ instead of $A_W \parallel A_R$. This procedure causes a synchronization between the considered paths, leading to the fact that A_R and A_W cannot be executed in parallel any more. The dependency conflict is suspended and the process can be executed according to the adapted model. However, since adding such a precedence restricts parallelism, it is recommended to perform conflict resolution as late as possible in order to avoid unnecessary synchronizations which could e.g. appear in conjunction with *OR splits*. Additional synchronizations are adapted to the actual execution state of the process, so there is no need for other activities to finish before processing a synchronization. We propose to use an optimistic approach to resolve dependency conflicts in a distributed way (see Section 3.5).

Unsynchronized. In contrast to *serialized*, the data class *unsynchronized* does not take care of any dependency conflict derived by variables of this class. As a tribute to such a loss of serializability, lost updates can appear. Therefore, the use of this data class is only acceptable under certain circumstances. The process variable *information* would be an example for this data class. As the variable is changed in both process paths independently but should not be used after synchronization, dependency conflicts resulting from that variable can be ignored without any need for additional concurrency control.

MaxAge. To conclude the discussion of data classes, we give an example for a data class in between the two presented extrema. In some cases, serializability can be omitted but the accessed data must not exceed a certain age. If, for example, a weather service continuously updates the forecast for tomorrow, the particular updates do not essentially differ. The process execution system can realize this by checking the respective variables for changes on other replicas if necessary, i.e. if the process contains a parallel write on this variable and a given period of time has passed until the last check. Concerning the presented example process, the variable *finished* is a possible candidate for this data class, especially if it would be used more often within process path P_2.

3.4 Control Flow Distribution and Synchronization

Considering runtime support for parallel process paths, the two main questions to deal with are which devices execute the parallel paths and how do they execute them. Since several approaches on chosing suitable devices with respect to specified functional and/or non-functional requirements has been developed [7,10], this section concentrates on the second question. If a participant has been selected, it receives a newly created replica of the current state of the process. A replica consists of four parts: the static process model, the execution state of the activities, the values of the variables, and the precedences which were added due to preceding conflict resolutions. To make sure that every device executes only the intended path, the first activity on the according path is set as the *start activity*. The process engine therefore proceeds executing the process at this activity.

To check for to be resolved dependency conflicts, every device participating in parallel execution generates and updates its own view on the dependency graph from its local replica. Before executing an activity belonging to a dependency conflict, an optimistic method to resolve the conflict is started (cp. Section 3.5). This method also deals with meanwhile changes in the process model, that are relevant for the current path but not have been propagated to the device yet.

Synchronizations between parallel paths have to be performed according to the process model with respect to new precedences possibly inserted by concurrency control. To perform final synchronization, the respective parallel paths' replicas have to be re-united by one of the participants. Since this can be realized as an incremental procedure, the following strategy is proposed: As soon as a device finishes executing a parallel path the replica is sent to a reachable device processing another parallel path or to a predefined device determined in advance. If a device holds more than one replica of the same process waiting at the same synchronization point, the device merges the data.

So, synchronization involves merging every part of the replicas into a new joint replica which is henceforth used. Since the static parts of the process model are never changed, it is irrelevant which replica becomes the source for this data. However, each replica changes the execution state of the activities processed by the respective participant. Because every parallel path is processed on exactly one replica, merging the execution states can be done without conflicts. In contrast, merging the values of the variables has to be done with respect to serializability criteria (or the data class defining the respective correctness criterion). By default, serialization of parallel activities is derived from the dependency graph. For each variable, the activity executing the last write access in the serialization has to be identified and the new joint replica adopts the value from the replica which has executed this activity. Finally, the precedences added by concurrency control are merged. All replicas are thus successively synchronized in pairs until the replicas of all relevant paths are processed and process execution can be continued. Prevention of inconsistencies is ensured by the optimistic conflict resolution as presented below.

3.5 Optimistic Conflict Resolution at Runtime

To resolve a remaining dependency conflict, the replicas of each parallel path affected by that conflict have to correspond with the precedence $A_W \lessdot A_R$ which breaks the cycle. This coordination cannot be realized if a device moves into an

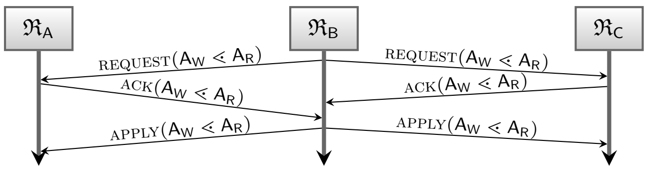

$\mathfrak{R}_A$
$\mathfrak{R}_B$
$\mathfrak{R}_C$
REQUEST($A_W \lessdot A_R$)
REQUEST($A_W \lessdot A_R$)
ACK($A_W \lessdot A_R$)
ACK($A_W \lessdot A_R$)
APPLY($A_W \lessdot A_R$)
APPLY($A_W \lessdot A_R$)

Figure 6. Message exchange for resolving a dependency conflict between three replicas. $\mathfrak{R}_B$ starts the resolution which is confirmed by the other devices.

area without network coverage. An optimistic approach can deal with such a situation by initially performing a local conflict resolution which can optionally be rolled back later. Therefore it is assumed that other parallel paths have not reached the conflict yet. It is furthermore assumed, that the notification about the chosen conflict resolution will reach the affected parallel paths before they start a conflict resolution on their own. Under the assumption that these optimistic presumptions are fulfilled, the conflict resolution can be realized by locally adding a new precedence which breaks the cycle. Thus, the following activities are executed – presumed that their effects can be compensated if necessary.

Since such optimistic assumptions cannot be guaranteed in general, two parallel paths may start a conflict resolution independently by adding a new precedence to their local replica. As this may lead to an inconsistent execution of the process, the following two properties have to be checked: First, two optimistic conflict resolutions must not produce a cycle of precedences. Second, the new precedences must not affect activities which have been started or which have already been executed. If at least one of these conditions does not hold for any replica, optimistic conflict resolutions have to be canceled successively until both properties apply. To ensure this, we propose a protocol which is partly based on the two-phase commit protocol (cp. Figure 6). Because the properties can only fail on replicas involved in the current dependency conflict, communication can be restricted to these devices.

The protocol uses five types of message: If a parallel path starts the optimistic conflict resolution, the message REQUEST($A_W \lessdot A_R$) is sent to all devices affected by the respective dependency conflict. This request represents a query to add the precedence $A_W \lessdot A_R$. If the two consistency properties are fulfilled, the receiver will add this tentative precedence to a local set $\mathcal{P}_O$ and will answer with ACK($A_W \lessdot A_R$), otherwise with NACK$_\omega$($A_W \lessdot A_R, B_W \lessdot B_R$) as disagreement. The last case includes a weight ω standing for the complexity of the work which has already been executed (e.g. processing time) to identify which of the incompatible optimistic resolutions should be aborted. This strategy minimizes the effort for aborting and repeating activities which have already been executed. In case of disagreement, the message CANCEL($A_W \lessdot A_R$) is sent to all involved devices in order to cancel the current optimistic conflict resolution, i.e. the receiver will remove the precedence from $\mathcal{P}_O$. Only if all affected parallel paths confirm the new precedence, the message APPLY($A_W \lessdot A_R$) is sent to finalize the conflict resolution by moving the precedence from $\mathcal{P}_O$ to the permanent process model.

Whenever the control in the non-optimistic phase reaches an activity A_W participating in a dependency conflict $\mathfrak{C}$, execution is stopped until A_W is conflict free at least in the local view. In such a case, there is an activity A_R satisfying the read-write dependency $A_R \prec A_W \in \mathfrak{C}$ because either $A_R \prec A_W$ or $A_R \lessdot A_W$ must exist for A_W to be part of a conflict. If there was $A_R \lessdot A_W$, the optimistic conflict resolution would have been executed already at A_R (or earlier) and there is an activity A_R with $A_R \prec A_W$. Selecting this particular read-write dependency for conflict resolution by adding the new precedence $A_W \lessdot A_R$ is advantageous, since the current device can prepare for the newly required synchronization at A_R without waiting for other devices and execute the following activities optimistically.

While waiting for acknowledgments of the request to accept the new precedence, the following activities can be executed if they are compensable: First, a current replica backup is created in order to allow recovery if optimistic conflict resolution fails. In order to avoid nested aborts in case of multiple subsequent dependency conflicts, a process has to pause execution until the optimistic phase is finished. For the same reason synchronizations are not to be performed in the optimistic phase. Replicas which have been created at *splits* can be held back until the optimistic phase is finished. In addition, execution of a parallel path should be paused when reaching an activity which is part of a tentative precedence because, otherwise, the probability of cancellation increases considerably.

4 Evaluation

This section demonstrates the correctness as well as applicability of the concepts presented so far. The central criterion for *correct* process execution using one-copy serializability is shown by formal verification in Section 4.1. Subsequently, relevant practical experiences with a prototype component supporting parallelism for mobile processes are reported in Section 4.2.

4.1 Formal Verification of Correctness Criterion

In order to verify the soundness of the introduced concept we have to show that every execution of an arbitrary process is correct with respect to one-copy serializability. It should be noticed, that this type of correctness only affects variables of the data class *serialized*, while variables of other classes follow other correctness criteria which are not considered in the following proof.

Since one-copy serializability was introduced in the context of databases, its formal background uses *database logs* (short *log*). A *log* is a partial ordered set of operations on a database [4]. Every operation belongs to a transaction T_i and reads resp. writes a variable x on a replica $\mathfrak{R}_a$. In the following, $r_i[x_a]$ resp. $w_i[x_a]$ are used for such read resp. write operations. The partial order reflects the order which exists in the involved transactions. Prior to any read operation on a variable there must have been a write operation on this variable on the same replica and conflicting operations must be ordered. Two operations are conflicting if they access the same variable on the same replica and one of these operations is a write. A *one copy log* is a special log, where only one replica exists for every variable. A log with more than one replica is also called *replicated database log* (*rd log*) for better distinction.

An execution of a process according to the introduced concept has therefore to be translated into an *rd log*. The execution of the process has hence completed, so due to optimistic conflict resolution there are no more dependency conflicts, i.e. the dependency graph has no cycles. Activities from non-executed paths are in the state *skipped* and will therefore be ignored. However, activities which execute read operations followed by write operations (and hence hold an order of these operations) correspond to transactions of an *rd log*. Creation and synchronization of replicas at *splits* resp. *joins* have to be treated as special transactions. Splitting a replica is formally a transaction of reading the values from the old replica and writing the values to two or more new replicas. Synchronizing replicas is a corresponding transaction which reads the values of the old replicas (i.e. for each variable x selects the replica holding the activity which has written x most recent in the topological order of the dependency graph) and writes them to a new replica. The order of these different types of transactions is extended by the order given by precedences from the process model and finally constitutes the *rd log*. Without loss of generality, we assume that the first transaction in a process initializes every variable with a default value. Hence, on every replica each variable is written before it is read. Because of the serialized access to variables on a replica, all conflicting operations are ordered and thus the result of such a transformation is a valid log.

The following proof makes use of the fact that an execution is correct with respect to one-copy serializability if it is equivalent to a serial execution on one copy. According to Bernstein and Goodman this is true if two logs have the same *reads-from* relation [4]. It thus has to be shown that for every possible execution there is a serial one copy log, which has the same *reads-from* relation:

Definition 8 (reads-from). *Two transactions T_i and T_j of a log L are in relation T_j* reads-x-from *T_i if $w_i[x_a]$ and $r_j[x_a]$ are operations in L which hold the order $w_i[x_a] < r_j[x_a]$ and no $w_k[x_a]$ falls between these operations. [4]*

The *reads-from* relation is therefore an unambiguous mapping $f\colon D \to T$, with $D \subseteq T \times V$, T being the set of transactions and V the set of variables. The domain of this mapping is equal for all logs which contain the same transactions. To proof equality of the *reads-from* relations f_1 and f_2 of two logs with the same transactions it has to be shown that for every tuple $(T, V) \in \mathrm{Dom}(f_1)$ the equation $f_1(T, V) = f_2(T, V)$ applies. Hence it has to be shown that every T_j reads-x-from T_i existing in the first log also exists in the second log.

Theorem 1. *Let L be an rd log according to the presented concept and let L_S be the one-copy log of the serial execution used at the synchronizations in the actual execution. Then every T_j* reads-x-from *T_i in L also exist in L_S.*

Proof: Let T_j reads-x-from T_i in L. According to Definition 8, three properties have to apply for T_j reads-x-from T_i in L_S to be true:

1. There are operations $w_i[x]$ and $r_j[x]$ in L_S. *Proof:* Because T_j reads-x-from T_i in L, there are $w_i[x_a]$ and $r_j[x_a]$ in L and hence the according operations $w_i[x]$ and $r_j[x]$ are in L_S.
2. It is $w_i[x] < r_j[x]$ in L_S. *Proof:* Because T_j reads-x-from T_i in L, $w_i[x_a] < r_j[x_a]$ is in L. Hence, the process model must contain a precedence $T_i < T_j$ which generates this order. Since L_S is derived from the dependency graph of L, this order also applies to the corresponding operations $w_i[x]$ und $r_j[x]$ in L_S.
3. There is no $w_k[x]$ between $w_i[x]$ and $r_j[x]$ in L_S. *Proof:* Let $w_k[x]$ be an arbitrary write operation on x in L_S. If $k = i$ or $k = j$, the proposition is obviously true. Therefore inequality is assumed. Three cases can be distinguished:
   (a) T_k is executed parallel to T_j in L. Hence there is a read-write dependency $T_j \prec T_k$. Because L_S is derived from the dependency graph, $r_j[x]$ is before $w_k[x]$ in L_S. Because $w_i[x] < r_j[x]$ and $r_j[x] < w_k[x]$, $w_k[x]$ is not between $w_i[x]$ and $r_j[x]$ in L_S.
   (b) T_k is executed parallel to T_i and not to T_j in L. The writes $w_k[x_a]$ and $w_i[x_b]$ are parallel but both before $r_j[x_c]$ (since $T_j < T_k$ with $T_i < T_j$ from 2. leads to a contradiction). Because T_j reads-x-from T_i in L, the selected serialization implies $T_k < T_i < T_j$ to be true for L_S.
   (c) T_k is not executed parallel to T_j and T_i in L. Hence T_k, T_j and T_i are executed in a serial sequence in L. Because T_j reads-x-from T_i in L, it is $T_k < T_i$ or $T_j < T_k$ in L. This order can be found in the dependency graph. Because L_S is derived from the dependency graph, $w_k[x]$ is before $w_i[x]$ or after $r_j[x]$ in L_S and therefore not between these operations. □

Theorem 2. *Every execution with respect to the concept is correct in terms of one-copy serializability.*

Proof: Let L be the *rd log* of an arbitrary execution with respect to the concept. According to Theorem 1 there is a serial one-copy log L_S, which has the same *reads-from* relation. According to [4], L and L_S are therefore equivalent. Hence L is correct in terms of one-copy serializability. □

4.2 Practical Experiences

Abstract prototype components of the presented models and mechanisms have been realized and integrated into the existing mobile process management system *DEMAC (Distributed Environment for Mobility-Aware Computing)* [9]. The resulting test environment consists of an XPDL[17]-based process description language, a corresponding execution engine, a supporting context management system, and an underlying communication infrastructure for asynchronous messaging. Figure 7 gives an overview of the integrated prototype architecture for supporting distributed parallel paths by using system-specific adapters.

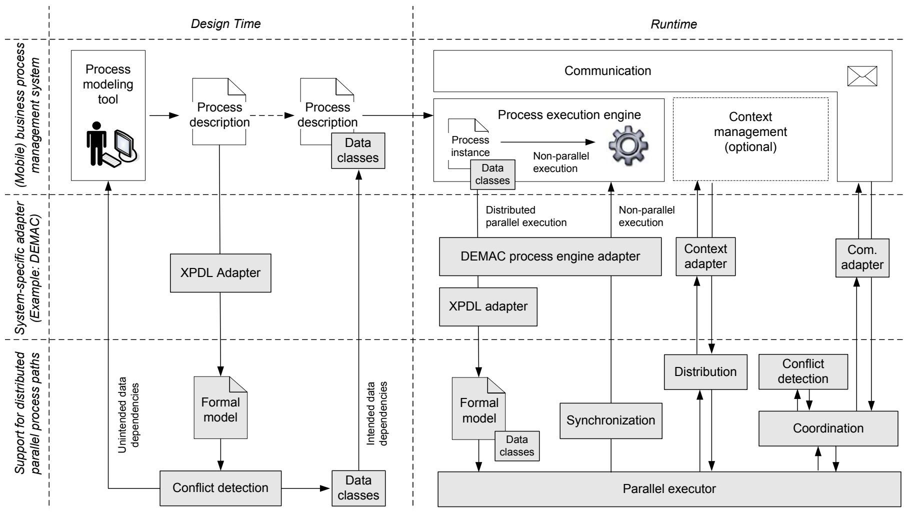


Figure 7. Schematic overview of prototype components integrated into the overall DEMAC process management system

In this prototype environment, the process designer starts with modeling the mobile process and thus produces a process description in XPDL format. Using the XPDL-adapter, the relevant parts of the process description are then translated into the formal model as presented in Section 3. Consequently, the process designer can test the developed process model for dependency conflicts and solve unindented data dependencies already at design time (e.g. the conflict resulting from process variable *information*). This design time conflict detection also forms the basis for the assignment of data classes. As current process modeling languages do not support this, data classes can be stored in an additional data container. Hence, execution engines which do not support the data class concept can simply ignore the container. As they have to ensure a serializable execution, relaxing serializability can be realized as a fully backward compatible procedure.

During runtime, the process description and its data classes are distributed to available devices. To realize this, the DEMAC context management system is responsible for selecting appropriate devices in order to execute the specified tasks. Finally, the communication system of the existing mobile process management system can be utilized for the communication with other (mobile) process engines – provided that messages sent by a device are received in the same order (*single-source FIFO*) and other relevant process participants can be addressed and searched by a unique identifier (e.g. URL or UUID). It is further assumed that devices become re-available in finite time.

The runtime behavior of the system was practically tested by realizing the example process as presented in Section 1. Figure 8 summarizes six tested vari-

	Execution variants	Resulting actual execution order during the tests	Correctness		Coord. mess.
			Serial.	User	
1. Unrestricted execution	**a) Immediate user interaction**	$A_{01}, \{A_{11}, A_{12}, A_{13}, A_{14}\} \parallel \{A_{21}, A_{22}, A_{23}, A_{24}\}, A_{02}$			
	New Precedences:	{}			0
	Variables:	**Resulting values and effects:**			
	balance	8.5	no	no	
	information	A_{13} and A_{22} received intended input data	no	yes	
	finished	A_{24} was executed (semantically incorrect)	yes	no	
	b) Delayed user interaction	$A_{01}, A_{21}, A_{22}, A_{23}, A_{24}$, <wait>, $A_{11}, A_{12}, A_{13}, A_{14}, A_{02}$			
	New Precedences:	{}			0
	Variables:	**Resulting values and effects:**			
	balance	8.5	no	no	
	information	A_{13} and A_{22} received intended input data	no	yes	
	finished	A_{24} was executed (semantically correct)	yes	yes	
2. Serialized execution	**a) Immediate user interaction**	$A_{01}, \{A_{11}, A_{12}\} \parallel A_{21}, A_{13} \parallel A_{22}, A_{23}, A_{14}, A_{02}$			
	New Precedences:	$\{A_{11}\rightarrow A_{22}, A_{21}\rightarrow A_{13}, A_{12}\rightarrow A_{24}, A_{23}\rightarrow A_{14}\}$			12
	Variables:	**Resulting values and effects:**			
	balance	16,9	yes	yes	
	information	A_{13} received wrong input data (i.e. data collected by sensor)	yes	no	
	finished	A_{24} was omitted (semantically correct)	yes	yes	
	b) Delayed user interaction	A_{01}, A_{21}, <wait>, $A_{11}, \{A_{12}, A_{13}\} \parallel A_{22}, A_{23}, A_{14}, A_{02}$			
	New Precedences:	$\{A_{11}\rightarrow A_{22}, A_{21}\rightarrow A_{13}, A_{12}\rightarrow A_{24}, A_{23}\rightarrow A_{14}\}$			12
	Variables:	**Resulting values and effects:**			
	balance	16,9	yes	yes	
	information	A_{22} received wrong input data (i.e. data entered by user)	yes	no	
	finished	A_{24} was omitted	yes	yes	
3. Data class execution	**a) Immediate user interaction**	$A_{01}, \{A_{11}, A_{12}, A_{13}, A_{14}\} \parallel \{A_{21}, A_{22}\}, A_{23}, A_{02}$			
	New Precedences:	$\{A_{14}\rightarrow A_{23}\}$			3
	Variables / data class:	**Resulting values and effects:**			
	balance: serialized	16,9	yes	yes	
	information: unsynchronized	A_{13} and A_{22} received intended input data	relaxed	yes	
	finished: maxAge(30)	A_{24} was omitted (correct)	relaxed	yes	
	b) Delayed user interaction	$A_{01}, A_{21}, A_{22}, A_{23}, A_{24}$, <wait>, $A_{11}, A_{12}, A_{13}, A_{14}$			
	New Precedences:	$\{A_{23}\rightarrow A_{14}\}$			3+2
	Variables / data class:	**Resulting values and effects:**			
	balance: serialized	16,9	yes	yes	
	information: unsynchronized	A_{13} and A_{22} received intended input data	relaxed	yes	
	finished: maxAge(30)	A_{24} was executed (semantically correct)	relaxed	yes	

Figure 8. Example scenario execution variants and test results

ants including a comparison of unsupported execution, serialized execution, and execution with data classes. Each variant was tested under two conditions, i.e. under a) both parallel paths could be executed immediately and under b) the user interaction A_{11} was delayed.

As expected, the evaluation shows that variant 1 supports unrestricted parallelism, but produces wrong results (e.g. the value of *balance* does not consider modifications on both paths). Variant 2 can be compared with the behavior of distributed (non-mobile) workflow systems. Besides the variable *information* (which results from quite arguable modeling), it produces correct results. Nevertheless, parallelism is highly restricted: In case both paths are reachable and parallel tasks are processed (variant 2a), this might still be acceptable. However, variant 2b shows that P_2 is totally halted because it has to wait for results processed by A_{11} – although (semantically) this is not necessary. Accordingly, variant 2 also produces a relatively high number of coordination messages.

In contrast, the actual execution of variant 3, using the data classes as proposed in Section 3.3, shows results and effects which are correct with respect to serializability or at least to the intentions of the user. In case of variable *information*, serialization is unwanted and thus relaxed. The correctness criteria of variable *finished* is determined by its age. In case the data is older than 30 minutes, a more recent data value has to be fetched, but it does not matter if (and by which activity) this variable has been modified, so R_3 does not have to wait for other activities to be finished. Although activities A_{11} and A_{21} are rather longrunning (> 30 minutes), variant 3a does not require updating the variable, because the value was included in the previous synchronization resulting from precedence $A_{14} \lessdot A_{23}$. In contrast, in variant 3b an additional request-response pair of messages has been sent in order to get the latest value of this variable. As to see, parallelism is restricted only in case the variable *balance* is read. However, due to optimistic conflict resolution both parallel paths can proceed even in case of (temporary) communication problems. Considering variant 3b, even all of P_2 can be executed without waiting for "lazy" activities on P_1. The applicability of the presented concept can therefore also be confirmed by practical experiments.

5 Conclusion

This paper proposed a multilevel approach to realize distributed parallel process execution with multiple mobile process participants. Optimistic conflict resolution and application-specific data classes have been applied in order to reduce communication efforts and thus increase parallelism. Main benefits are a formally proven model for correct execution of parallel process paths and a respective prototype component which was integrated and tested in an existing mobile process management system. In such a scenario, process management systems for distributed parallel process paths can use the generic meta model by attaching a system-specific adapter. Furthermore, also (stationary) cross-organizational business processes involving execution engines of different collaborating parties can benefit from a loosely coupled distribution and synchronization strategy without coordination overhead by sharing the intentions of the process designer respectively.

References

1. Adelstein, F., Gupta, S.K.S., Richard III, G., Schwiebert, L.: Fundamentals of Mobile and Pervasive Computing. McGraw-Hil (2005)
2. Alonso, G., et al.: Exotica/FMDC: Handling disconnected clients in a workflow management system. In: Conf. on Cooperative Information Systems. pp. 99–110 (1995)

3. Baresi, L., Maurino, A., Modafferi, S.: Workflow partitioning in mobile information systems. In: Proc. of IFIP TC8 Working Conference on Mobile Information Systems. pp. 93–106. Springer, Boston (2004)
4. Bernstein, P.A., Goodman, N.: The failure and recovery problem for replicated databases. In: Proc. of 2nd Annual ACM Symp. on Principles of Distributed Computing. pp. 114–122. ACM (1983)
5. Davidson, S.B.: Optimism and consistency in partitioned distributed database systems. ACM Transactions on Database Systems 9(3), 456–481 (1984)
6. Hackmann, G., Haitjema, M., Gill, C., Roman, G.C.: Sliver: A BPEL workflow process execution engine for mobile devices. In: Proc. of Int. Conf. on Service-Oriented Computing (ICSOC 2006). pp. 503–508. Springer (2006)
7. Hackmann, G., Sen, R., Haitjema, M., Roman, G.C., Gill, C.: Mobiwork: Mobile workflow for MANETs. Tech. rep., Washington University (2006)
8. Kopp, O., Khalaf, R., Leymann, F.: Deriving explicit data links in WS-BPEL processes. In: Proc. of the 2008 IEEE Int. Conf. on Services Computing (Vol 2). pp. 367–376. IEEE Computer Society, Washington, DC, USA (2008)
9. Kunze, C.P., Zaplata, S., Lamersdorf, W.: Mobile processes: Enhancing cooperation in distributed mobile environments. J. of Computers 2(1), 1–11 (2 2007)
10. Kunze, C.P., Zaplata, S., Turjalei, M., Lamersdorf, W.: Enabling context-based cooperation: A generic context model and management system. In: 11th Int. Conf. on Business Information Systems (BIS 2008). pp. 459–470. Springer (2008)
11. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. PTR Prentice Hall (2000)
12. Martin, D., Wutke, D., Leymann, F.: A novel approach to decentralized workflow enactment. In: Proc. of the 12th Int. IEEE Enterprise Distributed Object Computing Conf. pp. 127–136. IEEE Computer Society, Washington, DC, USA (2008)
13. Mecella, M., et al.: WORKPAD: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In: Proc. of Int. Symp. on Collaborative Technologies and Systems (CTS'06). pp. 173–180. IEEE Computer Society (2006)
14. Montagut, F., Molva, R.: Enabling pervasive execution of workflows. In: Proc. of the 1st Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing. IEEE Computer Society, Washington, DC, USA (2005)
15. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A.K., Weikum, G.: From centralized workflow specification to distributed workflow execution. J. Intell. Inf. Syst. 10(2), 159–184 (1998)
16. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. In: Proc. of the 19th Annual Conf. on Object-Oriented Programming, Systems, Languages, and Applications. pp. 170–187. ACM, New York (2004)
17. Norin, R., Marin, M.: XML process definition language. Specification WFMC-TC-1025, Workflow Management Coalition (2002)
18. Reichert, M., Rinderle, S., Dadam, P.: ADEPT workflow management system: Flexible support for enterprise-wide business processes. In: Proc. 1st Int. Conf. on Business Process Management (BPM '03). pp. 371–379. Springer (2003)
19. Satyanarayanan, M.: Fundamental challenges in mobile computing. In: Proc. of the 15th ACM Symposium on Principles of Distributed Computing (1996)
20. Sen, R., Roman, G.C., Gill, C.D.: CiAN: A workflow engine for MANETs. In: Proc. of COORDINATION 2008. pp. 280–295. Springer (2008)
21. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. Comput. Syst. 20(3), 239–282 (2002)

A.6 From Requirements to Executable Processes – A Literature Study

Authors:

UniDue: Andreas Gehlert

USTUTT: Olha Danylevych

USTUTT: Dimka Karastoyanova

Published in:

- Proceedings of the 5th International Workshop on Business Process Design. BPD 2009. September 7, 2009 - Ulm, Germany.

From Requirements to Executable Processes – A Literature Study

Andreas Gehlert[1], Olha Danylevych[2], Dimka Karastoyanova[2]

[1]University of Duisburg-Essen, Schützenbahn 70; 45117 Essen
andreas.gehlert@sse.uni-due.de
[2]University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart
{olha.danylevych | dimka.karastoyanova}@iaas.uni-stuttgart.de

Abstract. Service compositions are a major component to realize service-based applications (SBAs). The design of these service compositions follows mainly a process-modelling approach—an initial business process is refined until it can be executed on a workflow engine. Although this process-modelling approach proved to be useful, it largely disregards the knowledge gained in the requirements engineering discipline, e. g. in eliciting, documenting, managing and tracing requirements. Disregarding the requirements engineering phase may lead to undesired effects of the later service compositions such as lack of acceptance by the later users. To defuse this potentially critical issue we are interested in the interplay between requirements engineering and process modelling techniques. As a first step in this direction, we analyse the current literature in requirements engineering and process modelling in order to find overlaps where the techniques from both domains can be combined in useful ways. Our main finding is that scenario-based approaches from the requirements engineering discipline are a good basis for deriving executable processes. Depending whether the focus is on requirements engineering or on process design the integration of the techniques are slightly different.

Keywords: Requirements Engineering, Process Modelling, Use Cases, Process Fragments

1 Introduction

Service compositions are the central element in service-based applications (SBAs) — the new paradigm in software and service development. A service composition combines a set of services according to a meaningful business goal. In the case of using a process-based approach for service compositions the services are connected by control and data flow definitions. The underlying idea of such a process-modelling approach is to step-wise refine a process model until it contains all necessary information to be executed, for instance, an executable BPEL process can be executed on a BPEL engine (cf. bottom of Fig. 1).

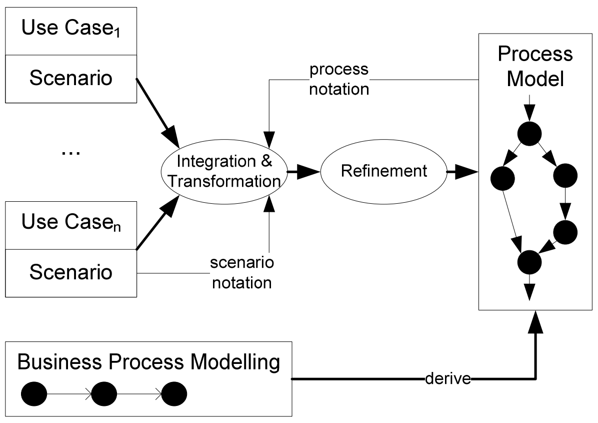


Fig. 1. Framework for Intertwining Requirements Engineering and Process Modelling Approaches.

One of the advantages of this approach is straightforward and of great practical importance for the creation of SBAs: the same modelling paradigm (process modelling) can be used during the entire development process of the service composition. This enables an automated support for the verification of the process model and its translation from the initial process model to the executable one. This modelling process, however, is a highly creative activity [e. g. 1], which can be difficult to control and plan.

Another discipline, which heavily depends on modelling, is requirements engineering (RE). Requirements engineering techniques such as prioritisation allow explicitly planning, managing and controlling modelling projects. In addition, agreement techniques allow to reason about the agreements achieved by different stakeholders. Lastly, tracing techniques allow documenting the origin and the destination of a requirements artefact [cf. 2 for a recent review on techniques of the different RE activities]. So, there is a potential to use RE techniques to structure and manage the process modelling activities.

Most interestingly, RE techniques, such as use cases based techniques share many commonalities with processes since use cases contain scenarios. A scenario is a sequence of activities, which describe a typical system interaction. Therefore, scenarios document parts of a process [3]. In particular, use case based techniques allow to model use cases in isolation and to verify and integrate those individual use cases later [4, p. 314]. Another argument for using RE techniques is based on empirical findings. Nawrocki et al. found in [5] that use cases are significantly easier to understand with respect to *error detection* than corresponding process models written in the Business Process Modelling Notation (BPMN).

Use case based approaches do not make any assumptions about the technology of the system to be built. Consequently, we are interested in answering the questions how RE techniques can help to derive process models. The starting point for this interaction is the idea to apply use case based RE techniques to derive isolated process fragments. These fragments need to be integrated and subsequently translated into an executable process modelling language. We are not only interested in the current state of the art from the two disciplines—requirements engineering and process modelling, but also to know what information can be captured by this approach and what information need to be added manually (cf. top of Fig. 1).

The research question we are addressing here is how requirements engineering and process modelling techniques can be combined to support the definition of executable workflows. To answer this question, we analyse the interplay between requirements engineering and process modelling techniques from a requirements and from a process modelling perspective. This analysis is based on an extensive literature review of both disciplines.

The paper is structured as follows: Section 2 reviews the related work in requirements engineering and process modelling to analyse existing approaches in both disciplines as well as approaches, which are bridging the gap between both disciplines. Because of the focus on requirements engineering and process modelling respectively in section 2 the interplay of the techniques of those disciplines differ slightly. In section 3 we discuss the consequences of these differences and give pointers to possible future work.

2 From Requirements to Executable Processes – Combining Techniques

The state of the art in business process modelling encompasses a variety of notations for modelling business process, ranging from declarative approaches like DecSerFlow [6] to imperative/workflow-like approaches such as BPMN and the Business Process Execution Language (BPEL). The process modelling notations have different levels of abstraction—from technology-independent graphical modelling notations like Event-Driven Process Chains (EPC) [7] or BPMN to executable process modelling notations like BPEL.

However, the design of a process model is currently based on the step-wise refinement of an initial business process. This refinement is a complex modelling activity, which is difficult to manage and control. Although very similar approaches for modelling processes, managing and controlling those modelling activities are well understood in the RE discipline, work has just started to use RE techniques for designing business processes (cf. subsection 2.1). Those approaches apply use cases to develop isolated scenarios, which are integrated later on to derive a process. This process must than be further refined in order to be executable.

Approaches originated from the process modelling discipline translate use cases into process modelling notations (e. g. EPCs or BPMN), integrate the resulting models and translate them further to executable process models (e. g. written in BPEL, c.f. subsection 2.2). In addition, those approaches introduce relevant information, which is currently not covered by traditional RE approaches such as constraints.

2.1 Requirements Engineering Perspective

One accepted requirements engineering approach is the use case approach. A use case is a structured description of the interaction between the system and its users. According to Cockburn [3], a use case description contains a primary actor initiating the use case, stakeholders influenced by the use case, the goal of the use case, guarantees (e. g. post-conditions), which hold when the use case is executed, pre-conditions and triggers determining when the use case is started, the main scenario and extensions to this main scenario describing the different use case steps. Use cases are usually documented in textual forms with the help of use case templates.

A system specification based on use cases is then a set of such use cases. Using a set of use cases for the system specification allows eliciting and documenting the system's requirements in decentralised teams. This comes at the cost of having a large number of use cases, which must be carefully managed and integrated [8]. Although UML use case diagrams [9] allow modelling the dependencies between different use cases by means of use case diagrams, their degree of formality alone is not enough to foster the integration of the embedded scenarios and, therefore, to derive a process [4].

The use case elements of interest for this paper are scenarios. Scenarios describe a sequence of steps, which lead to the fulfilment of the use case's goal. Each scenario can be extended by other scenarios in order to introduce alternatives and loops. Scenarios have particularly proven to be useful in requirements engineering projects especially when abstract modelling fails or when interdisciplinary teams work in the project [10, p. 38].

The scenarios embedded in the use case are usually expressed in natural language. Although this fosters the communication with non-technical stakeholders, it is associated with the difficulty of using the provided information in an automated manner, e. g. to automatically integrate scenarios or to automatically check scenarios for validity and consistency. One way to deal with this problem is to use a more formal notation to represent scenarios such as message sequence charts (MSCs). Automated support in deriving MSCs from textual use cases are for instance provided in [11].

Since their introduction in 1996 by the International Telecommunication Union (ITU), MSCs have a long and successful history in the RE discipline [12]. Their formal syntax and semantics allow verifying, transforming and integrating MSCs automatically. In their recent paper Uchitel et al. [8, p. 101] describe three approaches to integrate MSCs: the first approach is built on modelling the relations between the individual MSCs in a high level message sequence chart (hMSC). The introduction of hMSCs further allows re-using individual MSCs in different paths of the system's behaviour. Another approach is built on the component's states embedded in the MSC. Identical states in different MSCs are used for the integration. Lastly, a constraint-based approach [e. g. 4] can be used to integrate individual MSCs.

Each of those integration approaches comes with their distinct advantages and disadvantages. hMSCs for instance provide a good overview of the system and allow at the same time to re-use scenarios in different parts of the hMSC. This approach fosters the creation of many small scenarios, which are themselves difficult to understand. The integration with the help of scenario states allows modelling larger chunks of the system in one scenario but hinders scenario re-use and complicates the integration of the individual scenarios. Lastly, the constraint-based approach is most expressive and allows the description of arbitrary combination of individual scenarios. Since the constraints, however, are formulated in a formal language, they are difficult to understand for non-technical stakeholders.

So far, we have demonstrated that the requirements engineering discipline provides a tool-chain, which allows to elicit use cases in an informal manner, to derive more formal scenarios based on this specification and to integrate individual scenarios forming a coherent system specification (cf. Fig. 2). The missing element is a transformation algorithm, which translates the integrated scenario into an executable process model, e. g. into BPEL.

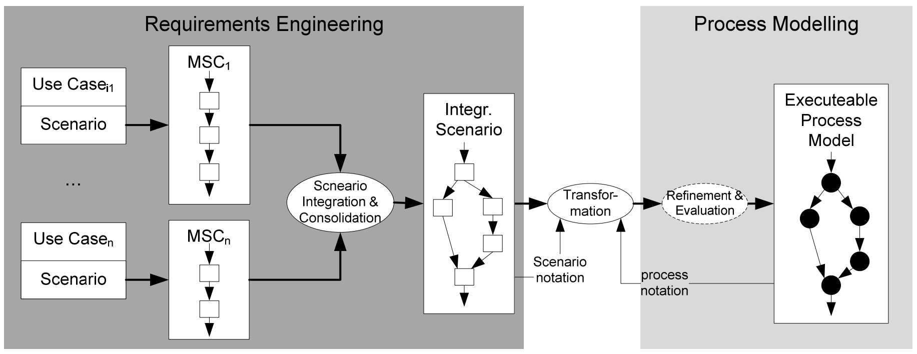


Fig. 2. Requirements Engineering Perspective of Deriving Process Models Based on Use Cases

Current transformation approaches such as [13-15] are based on an intermediate format, e. g. EPC or BPMN, which are in turn transformed into BPEL code [e. g. 16]. These approaches are discussed in more detail below. Having the focus on requirements engineering it is important to note that the individual scenarios as part of the requirements are integrated prior to their translation to a suitable workflow notation. This issue will be elaborated in more detail below.

2.2 Process Modelling Perspective

An alternative way of deriving an executable process is to translate the *individual* use-case scenarios to a process oriented language and to integrate the resulting process fragments afterwards using process-merge technologies. Lübke [15] for instance provide algorithms, which transforms a textual scenario as part of a use case specification into an EPC and integrates the resulting EPCs to a coherent model (cf. Fig. 3).

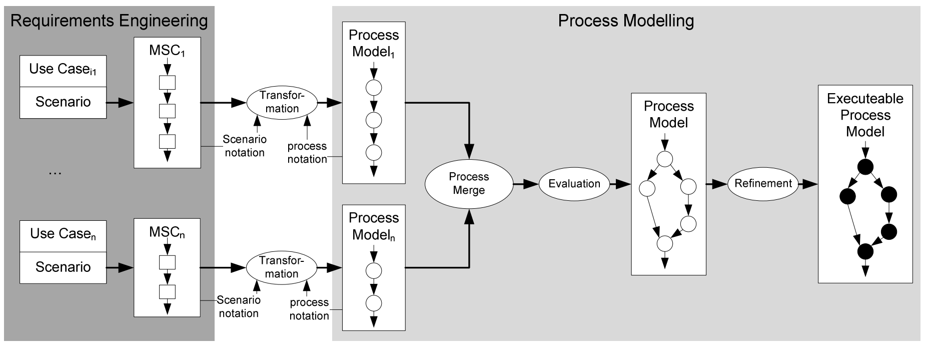


Fig. 3. Process Modelling Perspective of Deriving Process Models Based on Use Cases

Having the focus on processes, the scenarios as part of the requirements are transformed to processes and these processes are merged later on. In contrast to the requirements engineering perspective discussed above, the merging activity is performed to processes and not to scenarios. This difference is further elaborated in section 3.

The approaches presented in [13, 15] are based on merging the resulting EPCs according to their pre- and post-conditions, e. g. on the events initiating and finalising an EPC process fragment. The difference between this work and the RE approach is the point in time the use case is translated into a process model. While in RE approaches the scenarios are integrated first and translated afterwards into a process model, the process oriented approaches translate use cases to process models first and integrate them afterwards.

Although the two approaches seem to be similar in nature, merging process models allow the elicitation and the usage of additional constraints on those process models. This additional constrains guide the integration of the resulting process fragments, hence, allowing a better control over those fragments.

Although not exhaustive, the following constraints, which are typically not considered during a use case analysis, are used in the process modelling literature:

- Quality of Service related requirements for the whole system such as availability of the service composition.
- Operational properties like maximal costs for one process instance.
- Non-functional requirements of the single use cases, e. g. the use case A must be performed in at most 30 minutes.
- Extended relations between use cases such as use case A must be executed two times before use case B.

These constraints may also influence the process model itself. For instance, if use case A must be executed 10 times before use case B (extended relation between use cases) and the execution time for use case A should be small (non-functional constraint), the process designer may decide to re-organise the process according to the multiple instance pattern so that different instances of use case A can be executed in parallel to save time and to fulfil the non-functional constraint.

Specification and modelling of constraints from requirements has been extensively covered in the literature. Lu et al. [17] for instance classify constrains in strategic, operational, regulatory and contractual constraints. Gagne and Trudel [18] treat the specification of temporal logic constraints for process modelling using Allen's interval algebra [19]. Förster et al. [20] address the verification of process constraints expressed as Business Rules."

After the application of the constraints to the process fragments, there may still be more than one process, which fulfils all constraints since the integration of the different process fragments can be seen as combinatory problem. Process metrics help to chose among the possible process models fulfilling all constraints.

The metrics to be adopted depend on the important properties of the process models. On the one hand these are metrics used for prediction of the Quality of Service for a process-based service composition. This issue has been extensively treated, among others, by Marzolla et al. [21] and Rud et al. [22]. On the other hand the metrics evaluating "quality" of the model are also of interest. Such metrics may include the cohesion and coupling metrics proposed by Vanderfeesten et al. in [23]. The authors transform the well understood metrics from the software engineering discipline ([24]) to processes. Vanderfeesten et al. argue that the ratio of cohesion and coupling is an important characteristic of execution quality and maintenance and, therefore, can be used while choosing the most appropriate process model. Other metrics applicable for the evaluation of process model alternatives are the modularity metric by Reijers and Mendling [25], which is proven to be important for the understandability, and cyclic complexity of processes [cf. 26].

The conceptual difference between many process modelling techniques (e. g. EPC and BPMN) and executable process languages such as BPEL, which are mainly due to a different expressiveness and a different paradigm (graph based vs. block based) of the languages [27] lead to the need to refine processes. Process refinement has been extensively covered by both industry and academia. The goal is to iteratively and incrementally refine an abstract process model (e. g. the one obtained from use cases) into an executable one with the help of one or more semi-automated model transformations.

Born et al. [28] for instance treat the application of technologies to business process refinement, with the emphasis on goal-modelling and reuse or pre-existing business process patterns and fragments.

Markovic and Kowalkiewicz [29] also address the alignment of business goals and business processes using semantic technologies. They introduce a business goal ontology that is correlated to the already proposed business process ontology. High-level business goals (e.g. "increase company competitiveness") are hierarchical, can broken down into sub-goals (e.g. "uncover technology trends"), and are then refined into operational goals (e.g. "activate service") that are claimed to be easily mappable to concrete business process models.

3 Summary and Conclusions

We have shown in the paper that current RE techniques allow to elicit, document, verify and manage requirements, which are relevant for the design of processes. Use case based approaches contain scenarios, which can be used as process fragments when deriving processes. The key difference between RE approaches and process modelling approaches is that RE approach always aim to elicit and document a set of scenarios, which need to subsequently be integrated based on high level message sequence charts, scenario states or constraints. The resulting integrated scenarios can then be translated into process models (cf. subsection 2.1).

Approaches, which originate from the process modelling discipline, translate initial use cases into EPC or BPMN models, integrate these process models and translate them into executable process models. In contrast to RE approaches, the focus here is on the constraints, which guide the integration of different process fragments and on evaluation criteria, which help choosing between different process variants in case more than one process model fulfils all given constraints (cf. subsection 2.2).

Having analysed the requirements engineering and process modelling perspective we found a major difference: from the requirements engineering perspective the individual *scenarios are merged* before they are translated to a process modelling notation. In the process modelling perspective, the individual scenarios are translated to a process modelling notation and the resulting *process models are merged*. Although this difference seems to be a minor issue, the resulting consequences are important:

- *Merging Scenarios*: Merging scenarios is a well understood and much elaborated in the requirements engineering discipline. The overall process of designing executable processes could benefit from this maturity.
- *Merging Process Models*: comes with a couple of advantages, which are mainly due to the fact that process models are produced later in the process design lifecycle. Process models (e. g. a BPEL process) contain, among others, activities that perform message exchanges with (i. e. invocations of) other services (for instance, the invoke, receive and reply activities in BPEL).
  Moreover, the process models can be further annotated with Process Performance Metrics (PPMs) that explain how to calculate/evaluate particular performance attributes (e.g. completion time and average activity execution time). PPMs are not necessarily defined on single process executions (e. g. completion time of one instance), but their evaluation can span across multiple executions (e. g. average completion time).
  During the monitoring (either at run-time or post-mortem) of the execution of business processes and fragments annotated with PPMs, data about the processes are produced and aggregated by evaluating the PPMs.
  Notice that this information is 'linked' to executable artefacts (e. g. the particular process fragment in a Fragment Library owned by an enterprise), and not scenarios. That is, the data collected by PPMs are not known at design time, but only after the use of fragments in “production” (e. g. using them to compose business processes that are then run and monitored).
  The availability of this 'run-time' information is very relevant for the creation of merges of process fragments that are “optimal”, e. g. in terms of QoS. Moreover,

the additional information provides more flexibility during the process merge, allowing the consideration of criteria that span beyond the design time knowledge of the behaviour of the fragments, but also how they behaved at run-time (which is often not foreseeable at design time: for instance, you can generally never say if a process always terminates or not). Of course, this additional flexibility is not available while merging scenarios, because those data might not be available.
Finally, the data collected and aggregated from the PPMs during monitoring can support the identification of which fragments in the business process should be explicitly modelled (for instance using activities and control flows), and which one should instead be "masked" behind an external service. This is of course related to the out- and in-sourcing of business process fragments. There is an entire branch of BPM research, called (Business Process) Gap Analysis that deals with it, and (among others) with the problem of identifying the right "granularity" for the services.

Consequently, when choosing between the two alternatives the maturity of the scenario integration activity (focus on requirements engineering) must be balanced with the possibility of re-using more information (focus on process modelling). Which approach is more effective cannot be answered on the basis of this literature review and requires a future empirical evaluation.

There are a couple of deficiencies in both approaches, which are apparent from our literature study:

- *Missing information about services*: Neither scenario based techniques nor traditional process modelling techniques capture information about the services used in an executable process model or service composition. Weidlich at al. [30], for instance, argue that the process designer and the requirements engineer should work together when specifying the service interfaces and when discovering and selecting the service. Although they do not provide a clear methodology, they speculate that typical requirements engineering techniques such as use case visualisation, glossaries and requirements tracing can foster this collaboration.
  Vara and Sánchez [31] provide an alternative approach. They argue that process modelling should be executed by a process expert as first step in the development of SBAs. Once the process model is defined, the requirements engineer derives use cases for each activity in the process model and, thereby, specifies the behaviour of the service. However, the link between the process model and the use case is described informally only and a clear methodology is missing.
- *Different expressiveness of process modelling languages*: As argued in [27], process modelling languages and executable process languages have a different expressiveness, which leads to information loss or information deficits when transforming conceptual process models into executable ones. While this problem can be solved by annotating process models with the respective execution information [e. g. 32], it remains unclear how this additional information affects the readability of those models.
- *Difficult translation from scenarios to process models:* Because of the conceptual differences between scenarios and process models, the translation between the two worlds is difficult and not yet well understood.

Consequently, the bridge between requirements engineering techniques and executable workflows is not yet complete, e. g. it is not yet possible to develop and design service-based applications based on traditional requirements engineering techniques. This incompleteness results in manual and, consequently, error-prone and cost-intensive model transformations.

Future research directions is fourfold: First, empirical research is needed to decide in which situation a requirements centred and in which situations a process centred perspective is beneficial. Second, requirements engineering techniques must be extended to cover important aspects for the service world such as quality of service, service selection and compliance. Third, in the process world the translation between process models and their executable counterparts need to be researched. Finally, the translation between requirements engineering and process modelling notations should be investigated in more detail.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

1. Dresbach, S.: Modeling by Construction: A New Methodology for Constructing Models for Decision Support. Lehrstuhl für Wirtschaftsinformatik und Operations Research, University of Cologne, Germany (1995)
2. Cheng, B.H.C., Atlee, J.M.: Research Directions in Requirements Engineering. Conference on the Future of Software Engieering (FOSE 2007), Washington, USA (2007) 285-303
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Professional (2000)
4. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland (2000) 314-323
5. Nawrocki, J.R., Nedza, T., Ochodek, M., Olek, L.: Describing Business Processes with Use Cases. In: Abramowicz, W., Mayr, H.C. (eds.): 9th International Conference on Business Information Systems (BIS 2006) Vol. 85, Klagenfurt, Austria (2006) 13-27
6. Aalst, W.M.P.v.d., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Leymann, F., Reisig, W., Thatte, S.R., Aalst, W.M.P.v.d. (eds.): The Role of Business Processes in Service Oriented Architectures, Vol. 06291, Dagstuhl, Germany (2006)
7. Keller, G., Nüttgens, M., Scheer, A.-W.: Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi). Universität des Saarlandes (1992)
8. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Transactions on Software Engineering **29** (2003) 99-115
9. OMG: UML 2.0 Superstructure Specification. (2003)
10. Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenarios in System Development: Current Practice. IEEE Software **15** (1998) 34-45

11. Miga, A., Amyot, D., Bordeleau, F., Cameron, D., Woodside, C.M.: Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In: Reed, R., Reed, J. (eds.): 0th International SDL Forum (SDL 2001), Vol. 2078. Springer, Copenhagen, Denmark (2001) 268-287
12. ITU: Message Sequence Charts. International Telecomunication Union, Telecommunication Standardization Sector (1996)
13. Lübke, D., Schneider, K., Weidlich, M.: Visualizing Use Case Sets as BPMN Processes. 3rd International Workshop on Requirements Engineering Visualization (REV 2008), Barcelona, Spain (2008)
14. Ouyang, C., Dumas, M., Breutel, S., Hofstede, A.H.M.t.: Translating Standard Process Models to BPEL. In: Dubois, E., Pohl, K. (eds.): 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006), Vol. 4001. Springer, Luxembourg, Luxembourg (2006) 417-432
15. Lübke, D.: Transformation of Use Cases to EPC Models}. 5. Workshop der Gesellschaft für Informatik e.V. (GI) und Treffen ihres Arbeitskreises "Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (WI-EPK), Vol. 224. CEUR Workshop Proceedings, Vienna Austria (2006) 137-156
16. Ziemann, J., Mendling, J.: EPC-Based Modelling of BPEL Processes: A Pragmatic Transformation Approach. International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2005) Genoa, Italy (2005)
17. Lu, R., Sadiq, S., Governatori, G.: On Managing Business Processes Variants Data & Knowledge Engineering **68** (2009) 642-664
18. Gagné, D., Trudel, A.: A Formal Temporal Semantics for Microsoft Project based on Allen's Interval Algebra. In: Abramowicz, W., Maciaszek, L.A., Kowalczyk, R., Speck, A. (eds.): Business Process, Services Computing and Intelligent Service Management, Vol. 137, Leipzig, Germany (2009) 32-45
19. Allen, J.F.: Maintaining Knowledge about Temporal Intervals. Communications of the ACM **26** (1983) 832-843
20. Förster, A., Engels, G., Schattkowsky, T., Straeten, R.v.D.: Verification of Business Process Quality Constraints Based on Visual Process Patterns. First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, 2007. TASE '07, Shanghai, China (2007) 197-208
21. Marzolla, M., Mirandola, R.: Performance Prediction of Web Service Workflows. Third International Conference on Quality of Software ArchitecturesSoftware Architectures, Components, and Applications. Springer, Medford, MA, USA (2007) 127-144
22. Rud, D., Kunz, M., Schmietendorf, A., Dumke, R.: Performance Analysis in WS-BPEL-Based Infrastructures. 23rd Annual UK Performance Engineering Workshop (UKPEW 2007), Edge Hill University, Ormskirk, Lancashire, UK (2007) 130-141
23. Vanderfeesten, I., Reijers, H.A., Aalst, W.M.P.v.d.: Evaluating Workflow Process Designs using Cohesion and Coupling Metrics. Computers in Industry **59** (2008) 429-437
24. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering **20** (1994) 476-493
25. Reijers, H., Mendling, J.: Modularity in Process Models: Review and Effects. 6th International Conference on Business Process Management. Springer, Milan, Italy (2008) 20-35
26. Cardoso, J., Mendling, J., Neumann, G., Reijers, H.A.: A Discourse on Complexity of Process Models (Survey Paper). In: Eder, J., Dustdar, S. (eds.): Business Process Management Workshops, Vol. 4103. Springer, Vienna, Austria (2006) 117-128
27. Recker, J., Mendling, J.: On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modelling Languages. In: Krogstie, J., Halpin, T.A., Proper, H.A. (eds.): 11th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006). Namur University Press, Namur, Belgium, Luxembourg, Luxembourg (2006) 521-532

28. Born, M., Brelage, C., Markovic, I., Weber, I.: Semantic Business Process Modeling: from Business Goals to Execution-Level Business Processes. Forschungszentrum Informatik, Karlsruhe (2008)
29. Markovic, I., Kowalkiewicz, M.: Linking Business Goals to Process Models in Semantic Business Process Modeling. 12th International IEEE Enterprise Distributed Object Computing Conference, 2008 (EDOC 2008), Munich, Germany (2008) 332-338
30. Weidlich, M., Grosskopf, A., Lübke, D., Schneider, K., Knauss, E., Singer, L.: Verzahnung von Requirements Engineering und Geschäftsprozessdesign. 1. Workshops für Requirements Engineering und Business Prcess Management (REBPM 2009), Kaiserslautern, Deutschland (2009)
31. Vara, J.L.d.l., Sánchez, J.: Improving Requirements Analysis through Business Process Modelling: a Participative Approach. In: Fensel, D., Abramowicz, W. (eds.): 11th International Conference on Business Information Systems (BIS 2008). Springer, Innsbruck, Austria (2008) 165-176
32. White, S.A.: Using BPMN to Model a BPEL Process. BPTrends (2005) 1-18