Business Process Intelligence : Discovering and Improving Transactional behavior of Composite Services From Logs

Walid Gaaloul

LORIA - INRIA - CNRS - UMR 7503
BP 239, F-54506 Vandœuvre-ls-Nancy Cedex, France
gaaloul@loria.fr

Abstract. One of the main features that makes Web services a promising technology for automating B2B interactions is the ability to dynamically combine a set of services into a new value added composite Web services (CS). However, a main problem that remains is how to ensure a correct and reliable execution. This paper presents a set of techniques to improve CS transactional behavior for a better failures handling. Basically, we propose a set of mining techniques to discover CS model and its transactional behavior from logs. Then, based on this mining step, we use a set of rules to improve its design. We refer to this set of techniques as the Business Process Intelligence (BPI) techniques suite.

1 Introduction

Nowadays, enterprises are able to outsource their internal business processes as services and make them accessible via the Web. Then, they can dynamically combine individual services to provide new value-added composite services (CSs). However, due to the inherent autonomy and heterogeneity of services, a fundamental problem that remains concerns the guarantee of correct and reliable execution.

In this paper, we present a different approach that starts from a CS executions log and uses a set of mining techniques to discover the CS model and the CS transactional behavior. Then, based on these mined information, we use a set of rules to improve the CS failures handling and recovery. Discovering the transactional behavior allows to detect gaps mentioned above to improve the application reliability and interoperability. Our approach starts from effective executions, while previous works use only specification properties (which remain ”assumptions”).

The remainder of this paper is organized as follows. Section 2 presents the notion of composite Web service from a transactional point view. In section 3, we introduce some distinctive concepts and needed prerequisites. After that, we detail our approach that mainly proceeds in two steps. The first one consists in discovering the CS patterns from an event-based log (see section 4). Then, based on this mining step, we use a set of rules to improve the CS transactional design (see section 5). Finally, we conclude our paper by summarizing the main contributions and presenting some future works.

2 Transactional composite Sevices

Transactional composite services (TCS) is a CS that emphasizes transactional behavior for failures handling and recovery. Within TCS, we distinguish between *the control flow* and *the transactional behavior*.

2.1 Control flow

A process definition is composed of services. Services are related together to form a control flow via transitions which can be guarded by a control flow operator. The control flow dimension is concerned with the partial ordering of services. The services that need to be executed are identified and the routing of cases along these services is determined. Conditional, sequential, parallel and iterative routing elements are typical structures specified in the control flow dimension. We use workflow patterns [1] to express and implement the control flow dimension requirements and functionalities.

2.2 Transactional Behavior

CS transactional behavior specifies mechanisms for failures handling. It defines *service transactional properties* and *transactional flow (interactions)*.

services transactional properties : Within (TCS), services emphasizes transactional properties for its characterization and correct usage. The main transactional properties that we are considering are *retriable*, *compensatable* and *pivot*[2]. An service a is said to be retriable (a^r) *iff* it is sure to complete. a is said to be compensatable (a^{cp}) *iff* its work can be semantically undone. Then, a is said to be pivot (a^p) *iff* its effect can not be compensated.

Transactional flow : A transactional flow defines a set of interactions to ensure failures handling. Transactional CSs take advantage of service transactional properties to specialize their transactional interactions.

3 Prerequisites

3.1 TCS Event log

Following a common requirement in the areas of business processes and services management, we also expect the composite services to be traceable, meaning that the system should in one way or another keep track of ongoing and past executions. As shown in the UML class diagram in figure 1, `TCSLog` is composed of a set of events streams (see Definition 1). Each events stream traces the execution of one case (instance). It consists of a set of events (`event`) which capture the services life cycle performed in a particular TCS instance. An `Event` is described by the service identifier that it concerns, the current service state (`aborted`, `failed`, `canceled`, `completed` and `compensated`) and the time when it occurs (`TimeStamp`).

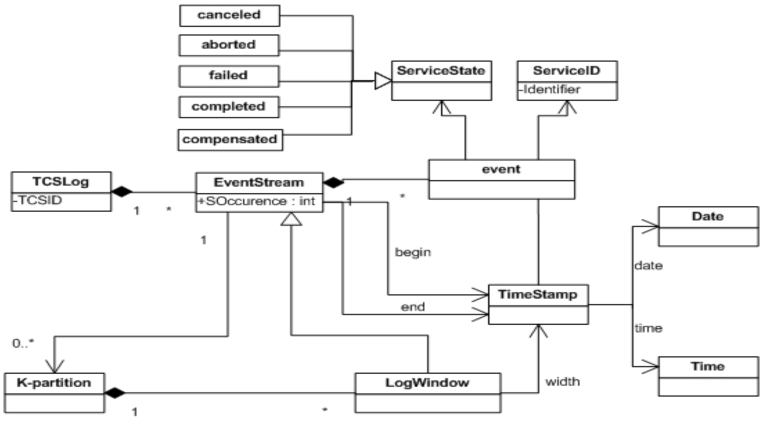


Fig. 1. Structure of a TCS Log

Definition 1 ***TCSLog***

*A TCS log is considered as a set of events streams. Each events stream represents the execution of one case. More formally, an events stream is defined as a quadruplet **stream**: (beginTime, endTime, sequenceLog, SOccurence) where:*

✓(beginTime: time) and (endTime: time) are the moment of log beginning log end,

*✓(sequenceLog: $\{$**event**$\}$): is an **ordered** events log tracing one TCS instance,*

✓(SOccurence : int) is the service execution instance number.

*So, **TCSLog**: (TCSID, $\{ServiceStream_i$: **stream**; $0 \leq i \leq$ number of TCS instantiations$\}$) is a TCS log where $ServiceStream_i$ is the events stream of the i^{th} TCS execution case.*

A `LogWindow` (see Definition 2) defines a set of `events` over an `EventStream`. Formally, the window beginning and ending represents an events stream interval characterizing the *width* of the `LogWindow`. A `K-partition` (see Definition 3) builds a set of partially overlapping `LogWindow`s partition over an `EventStream` where the width of `LogWindow`s is K. Each window is built by adding the next event log not included in the previous window.

Definition 2 ***log window***

*A log window defines a set of events over an events stream S : **stream** (bStream, eStream, sLog, TCSocc). Formally, we define a log window as a triplet **window**(wLog, bWin, eWin), where:*

✓(bWin : time) and (eWin : time) are the moment of the window beginning and end (with bStream $\leq$ bWin and eWin $\leq$ eStream)

*✓wLog $\subset$ sLog and $\forall$ e: **event** $\in$ S.sLog where bWin $\leq$ e.TimeStamp $\leq$ eWin $\Rightarrow$ e $\in$ wLog.*

Definition 3 ***K-partition***

*A **K-partition** builds a set of partially overlapping windows partition over an events stream.*

*K-partition : **TCSLog** $\rightarrow$ ($\{$ **window**$\}$)**

S : ***stream****(sLog, TCSocc, bStream, eStream)* $\rightarrow \{w_i$ ***: window****; 1≤i≤n*$\}$*; where:*

✓w_1*.bWin = bStream and* w_n*.eWin = eStream,*

✓$\forall w$: ***window*** $\in$ ***K-partition****,* width*(w)= K,*

✓$\forall$ *0≤i<n;* w_{i+1}*.wLog -* $\{$*the last e:****event*** *in* w_{i+1}*.wLog*$\}$ $\subset$ w_i*.wLog and* w_{i+1}*.wLog* $\neq$ w_i*.wLog.*

3.2 CS Set of Termination States

The state of a TCS instance composed of n services, at a specific time, can be presented by the tuple (x_1, x_2, ..., x_n), where x_i is the state of the service instance s_i at this time. A termination state of a TCS instance is the state in which this instance terminates. Let cs a TCS, we define $STS(cs)$ the set of all possible termination states of all cs instances.

4 Control Flow Mining

TCSs are case-based, many cases can be handled by following the same TCS process definition. Routing elements are used to describe sequential, conditional, parallel, and iterative routing thus specifying the appropriate route of a case. In this present work, we are interested in discovering ”elementary” routing TCS patterns: Sequence, AND-split, OR-split, XOR-split, AND-join, OR-join, and M-out-of-N Join patterns inspired from workflow patterns[1].

The main challenge which we cope with is the discovery of the sequential, conditional and concurrent behavior of these patterns. We meet these conditional and concurrent behavior after : (i) a ”fork” point where a single thread of control splits into multiple threads of control or before (ii) a ”join” point where multiple threads of control merge in a single thread of control. Thus, we have three types of *dependence relations* between services :

1. *Exclusive relation* is a activation dependency relation that captures the sequencing of services (*e.g.* Sequence pattern) where the enactment of a service depends only on the completion of one other.
2. *Conditional relation* is a activation dependency relation that captures selection of one service or more from a set of services potentially following after a ”fork” point (*e.g.* OR-split), or preceding before a ”join” point (*e.g.* OR-join), the execution of a given service.
3. *Concurrent relation* is not a activation dependency relation. In fact, it captures only concurrency or parallelism between services after a ”fork” point (e.g. AND-split pattern) and before a ”join” point (e.g. AND-join pattern).

Our control flow mining approach proceeds in two steps [3–5]: Step (i) the construction of statistical dependency table SDT, Step (ii) the mining of TCS patterns through a set of rules using the statistical specifications of these properties.

Construction of the statistical dependency table SDT : We use statistical calculus that extract control flow dependencies between services that are executed without ”exceptions” (*i.e.* they reached successfully their **completed** state). There is no need to use others events streams relating to failure executions containing `failed` or `aborted` or `compensated` or `canceled` states. In fact, these cases concern only TCS transactional behavior (see section 2) which tailors the mechanisms for failures handling and recovery. For these reasons, we need to filter TCS log and take only events streams of instances executed ”correctly”. We denote by $TCSLog_{completed}$ this TCS log projection.

Thus, the minimal condition to discover TCS patterns is to have TCS logs containing at least the `completed` event states. This feature allows us to mine control flow from ”poor” logs which contain only `completed` event state. Any information system using transactional systems such as ERP, CRM, or workflow management systems offer this information in some form [6].

For each service A, we extract From $TCSLog_{completed}$ the following information in the statistical dependency table (SDT): (i) The overall frequency of this service (denoted $\#A$) and (ii) The activation dependencies to previous services B_i (denoted $P(A/B_i)$). The size of SDT is N*N, where N is the number of TCS services. The (*m*,*n*) table entry (notation P(*m*/*n*)) is the frequency of the m^{th} service **immediately preceding** the n^{th} service. This SDT is a variation of the one from [7]. As it is computed, it presents some problems to express correctly services dependencies especially relating to concurrent or parallel behavior. In the following, we detail these issues and propose solutions to correct them.

Undetectable dependencies : If we assume that the events stream is exactly correct (i.e., contains no noise) and derives from a sequential (i.e no concurrent behavior) TCS, a zero entry in SDT represents an activation independence and a non-zero entry means an activation dependency relation (*i.e.* sequential or conditional relation). But, in case of concurrent behavior, as we can see in TCS patterns (like AND-split, AND-join, OR-join, etc) the events streams may contain interleaved events sequences from concurrent threads. As consequence, a service might not, for some concurrency reasons, depend on its immediate predecessor, but it might depend on another ”indirectly” preceding service.

To unmask and correct this erroneous frequencies we calculate the frequency using a *concurrent window*, i.e. we consider not only the events occurring immediately backwards but also the preceding events covered by the *concurrent window*. Formally, a concurrent window defines a log slide over an events stream (see Definition 2). *The width* of the *concurrent window* is the maximal duration that a concurrent execution can take. It depends on the studied TCS and is estimated by the user. Based on that, we construct an events stream partition (see Definition 3). This partition is formed by a set of overlapping windows.

Based on this definition, we compute the new statistical service dependencies. we scan the set **K-partition window**s over **TCSlog**, **window** by **window**, and for each **window** we compute for the last service the frequencies of its preceded services and the corresponding table is updated in consequence. The statistic servive activation dependency will be found by dividing each row entry in the previous table by the frequency of its corresponding service.

Erroneous dependencies Some entries in SDT can indicate non-zero entries that do not correspond to dependencies. These entries are erroneous because there is no activation dependencies between these services as suggested. Formally, two services A and B are in concurrence *iff the P(A/B)* and *P(B/A)* entries in SDT are different to zero. Based on this definition we propose an algorithm to detect services parallelism and then mark the erroneous entries in SDT. Through this marking, we can eliminate the confusion caused by the concurrence behavior which produces these erroneous non-zero entries. This algorithm scans the initial **SDT** and marks the dependencies of concurrent services by changing theirs values to (-1).

4.1 TCS patterns mining

After the compute of the statistical dependency table, the last step is the identification of TCS patterns through a set of rules. In fact, each pattern has its own statistical features which abstract statistically its activation dependencies, and represent its unique identifier. These rules allow, if TCS log is completed, the discovery of the whole TCS patterns included in the mined TCS. To be complete, TCS log should cover all the possible cases (i.e. if a specific routing element can appear in the mined TCS model, the log should contain an example of this behavior in at least one case). But, our control flow mining rules are characterized by a ”local” TCS patterns discovery. Indeed, these rules are context-free, they proceed through a local log analyzing that allows us to recover partial results of mining TcS patterns. In fact, to discover a particular TCS pattern we need only events relating to pattern’s elements. Thus, even using only complete fractions of TCS log, we can discover correctly corresponding TCS patterns (which their events belong to these fractions). To be complete, these fraction should cover all the possible cases of these patterns.

We divided the TCSs patterns in three categories : sequence, fork and join patterns. In the following we present rules to discover the most interesting TCS patterns belonging to these three categories.

Sequence pattern : In this category we find only the sequence pattern, in witch the enactment of the service B depends only on the completion of service A (*c.f.* table 1).

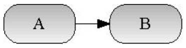
Rules	**workflow patterns**
$(\#B = \#A)$	Sequence pattern
$(P(B/A) = 1)$	

Table 1. Rules of sequence workflow pattern

Fork patterns : This category (*c.f.* table 2) has a ”fork” point where a single thread of control splits into multiple threads of control which can be, according to the used pattern, executed or not.

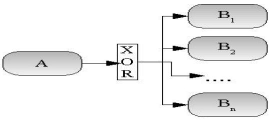
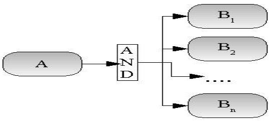
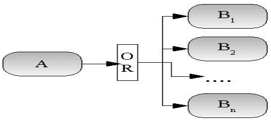
Rules	**workflow patterns**
$(\varSigma_{i=0}^{n}\ (\#B_i)$=#A)	XOR-split pattern 
$(\forall 0 \leq i \leq n;P(B_i/A)=1) \wedge$ $(\forall 0 \leq i,j \leq n;\ P(B_i/B_j)=0)$	
$(\forall 0 \leq i \leq n; \#B_i$=#A)	AND-split pattern 
$(\forall 0 \leq i \leq n;\ P(B_i/A)=1) \wedge$ $(\forall 0 \leq i,j \leq n\ P(B_i/B_j)=-1)$	
$(\#A \leq \varSigma_{i=0}^{n}\ (\#B_i)) \wedge$ $(\forall 0 \leq i \leq n;\ \#B_i \leq \#A)$	OR-split pattern 
$(\forall 0 \leq i \leq n;\ P(B_i/A)=1) \wedge$ $(\exists 0 \leq i,j \leq n;\ P(B_i/B_j)=-1)$	

Table 2. Rules of fork workflow patterns

Join patterns : This category (*c.f.* table 3) has a ”join” point where multiple threads of control merge in a single thread of control. The number of necessary branches for the activation of the service B after the ”join” point depends on the used pattern.

5 Mining and Improving Transactional behavior

We define at this level a set of rules allowing to mine transactional behavior. These rules allow to tailor the services transactional properties according to the discovered control flow and set of termination states.
$\forall\ service\ s$:

1. $s.failed \notin STS(s) \Longrightarrow$ *s is retriable*
2. $s.failed \in STS(s) \Longrightarrow$ *s is not retriable*
3. $s.compensated \notin STS(s) \Longrightarrow$ *s is not compensatable*
4. $s.compensated \in STS(s) \Longrightarrow$ *s is compensatable* $\wedge$ ***have to be compensated when one of its compensation conditions occurs***; We extract from the discovered control flow and the set of terminated state the compensation condition of a denoted $cpsCond(a)$.

The first (respectively the second) rule says that if a never fails (respectively can fail) then a is (respectively is not) retriable. The third and forth rules allow to deduce when a service a is compensatable and when it will be compensated.

Mining transactional dependencies returns to mine corresponding preconditions of services external transitions. We can extract the potential compensation conditions for

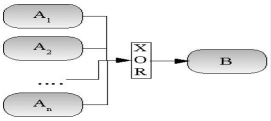
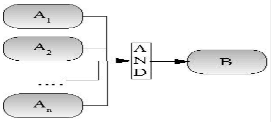
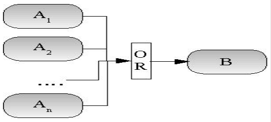
Rules	**workflow patterns**
($\Sigma_{i=0}^{n}$ ($\#A_i$)=#B)	XOR-join pattern
($\Sigma_{i=0}^{n}$ P(B/A_i)=1) $\wedge$ ($\forall 0 \leq i, j \leq n$; $P(A_i/A_j) = 0$)	
($\forall 0 \leq i \leq n$; $\#A_i$=#B)	AND-join pattern
($\forall 0 \leq i \leq n$; $P(B/A_i) = 1$)$\wedge$ ($\forall 0 \leq i, j \leq n$ $P(A_i/A_j) = -1$)	
($m * \#B \leq \Sigma_{i=0}^{n}$ ($\#A_i$)) $\wedge$ ($\forall 0 \leq i \leq n$; $\#A_i \leq \#B$)	M-out-of-N Join pattern
($m \leq \Sigma_{i=0}^{n}$ $P(B/A_i) \leq n$) $\wedge$ ($\exists 0 \leq i, j \leq n$; $P(A_i/A_j) = -1$)	

Table 3. Rules of join workflow patterns

a given service s ($ptCpsCond_i(s)$) from the mined composite service skeleton and the the set of termination states (STS). The idea is : a potential compensation condition of s becomes a compensation condition if it is satisfied in a $ts \in STS$ such that the state of s in ts is compensated. We proceed similarly to deduce alternative ($ptAltCond_i(s)$) and cancelation conditions ($ptCnlCond_i(s)$) of each service.

Now, we can improve a TCS, regarding its initial design, through the following two phases : (i) omitting and correcting the wrong transactional mechanisms, (ii) and/or adding relevant transactional mechanisms for a better failure handling and recovery. By wrong transactional mechanisms we mean component transactional properties and transactional dependencies initially specified and which do not coincide with the reality. These wrong transactional mechanisms can be simply costly but also source of error. The following rules allow to generate the appropriate suggestions according to the discovered TCS transactional behavior.

$\forall$ component service, s, of TCS (we suppose that $\Diamond$F means F is eventually true):

1. $\forall\, ptCpsCond_i(s) \in ptCpsCond(s)$,
   $\Diamond(ptCpsCond_i(s)) \bigwedge ptCpsCond_i(s) \notin CpsCond(s) \Rightarrow$
   (a) s must be compensatable and
   (b) $CpsCond(s) = CpsCond(s) \bigoplus ptCpsCond_i(s)$.
2. $\forall\, ptCnlCond_i(s) \in ptCnlCond(s)$,
   $\Diamond(ptCnlCond_i(s)) \bigwedge ptCnlCond_i(s) \notin CnlCond(s) \Rightarrow CnlCond(s) = CnlCond(s)$ $\bigoplus ptCnlCond_i(s)$.
3. $\forall\, ptAltCond_i(s) \in AltCond(s)$,
   $\Diamond(ptAltCond_i(s)) \bigwedge ptAltCond_i(s) \notin AltCond(s) \Rightarrow AltCond(s) = AltCond(s)$ $\bigoplus ptAltCond_i(s)$.

$AltCond(s)$ defines the precondition to be enforced before the service s can be activated as an alternative of other service(s) (similarly for cancelation and abortion). The first rule postulates that for each potential compensation condition of s, $ptCpsCond_i(s)$; if this condition is eventually true and does not belong to the discovered compensation condition of s, then s must be compensatable and $ptCpsCond_i(s)$ becomes a compensation condition of s. That means $\forall$ s' $\in ptCpsCond_i(s)$ add a compensation dependency from s' to s according to $ptCpsCond_i(s)$. The second rule postulates that for each potential cancelation condition of s, $ptCnlCond_i(s)$; if this condition is eventually true and does not belong to the discovered cancelation condition of s, then $ptCnlCond_i(s)$ becomes a cancelation condition of s. That means $\forall s' \in ptCnlCond_i(s)$ add a cancelation dependency from s' to s according to $ptCnlCond_i(s)$. The third rule postulates that for each potential alternative condition of s, $ptAltCond_i(s)$; if this condition is eventually true and does not belong to the discovered alternative condition of s, then $ptAltCond_i(s)$ becomes an alternative condition of s. That means $\forall s' \in ptAltCond_i(s)$ add an alternative dependency from s' to s according to $ptAltCond_i(s)$.

6 Conclusion and Future work

In this paper we presented an original approach for ensuring reliable Web services compositions. Different from previous works, our approach starts from a TCS log. Generally, previous approaches develop, using their CS modelling formalisms, a set of techniques to analyze the composition model and check some properties [9–13]. Although powerful, the above formal approaches may fail, in some cases, to ensure CS reliable executions even if they formally validate the CS composition models. This is because properties specified in the studied composition models may not coincide with the reality (*i.e.* effective CSs executions). Note also that a number of research efforts in the area of workflow management have been directed for mining workflows models (a detailed survey of this research area is provided in [14]). This issue is closely to that we propose in terms of control flow discovery. But there are practically no approaches to transactional behavior workflow mining except in [8, 4].

Our approach uses a set of mining techniques to discover the TCS control flow and the TCS transactional behavior. Then, based on this mining step, we use a set of rules to improve the TCS design. The mining phase is itself divided into two steps. The first one consists in mining TCS patterns. Then from the discovered information, we use a set of rules to mine the TCS transactional behavior. Our approach starts from effective executions, while previous works use only specification properties (which remain assumptions). Our control flow mining approach is original regarding other proposed techniques. It is characterized by a ”local” discovery techniques that allows to recover partial results. In besides, it discovers more behavioral complex features with a better specification of ”fork” point and ”join” point.

However, the work described in this paper represents an initial investigation. In our future works, we hope to discover more complex patterns by using more metrics (*e.g.* entropy, periodicity, etc.) and by enriching the TCS log. We are also interested in the modeling and the discovery of more complex transactional characteristics of cooperative TCS.

References

1. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
2. A. Elmagarmid, Y. Leu, W. Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the sixteenth international conference on Very large databases*, pages 507–518. Morgan Kaufmann Publishers Inc., 1990.
3. W. Gaaloul, S. Alaoui, K. Bana, and C. Godart. Mining Workflow Patterns through Event-data Analysis. In *The IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05). Workshop 6 Teamware: supporting scalable virtual teams in multi-organizational settings*. IEEE Computer Society Press, 2005.
4. W. Gaaloul, S. Bhiri, and C. Godart. Discovering workflow transactional behaviour event-based log. In *12th International Conference on Cooperative Information Systems (CoopIS'04)*, LNCS, Larnaca, Cyprus, October 25-29, 2004. Springer-Verlag.
5. W. Gaaloul, S. Alaoui, H. Bakkali, K. Bana, and C. Godart. WorkflowMiner : An infrastructure for Mining Workflow Patterns. In *3mes Journes Nationales sur les Systmes Intelligents: Thories et Applications (SITA'04)*, pages 10–13, Rabat, Morocco, December 6-7, 2004.
6. W.M.P. van der Aalst and L. Maruster. Workflow mining: Discovering process models from event logs. In *QUT Technical report, FIT-TR-2003-03, Queensland University of Technology*, Brisbane, 2003.
7. Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th international conference on Software engineering*, pages 73–82. ACM Press, 1995.
8. W. Gaaloul, S. Bhiri, and C. Godart. Discovering workflow patterns from timed logs. In *EMISA 2004, Informationssysteme im E-Business und E-Government, Beiträge des Workshops der GI-Fachgruppe EMISA*, LNI, pages 84–94, Luxemburg, October 6-8, 2004. GI.
9. Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the twelfth international conference on World Wide Web*, pages 403–410. ACM Press, 2003.
10. Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, pages 191–200. Australian Computer Society, Inc., 2003.
11. Benchaphon Limthanmaphon and Yanchun Zhang. Web service composition transaction management. In *Proceedings of the fifteenth conference on Australasian database*, pages 171–179. Australian Computer Society, Inc., 2004.
12. Paulo F. Pires, Mario R. F. Benevides, and Marta Mattoso. Building reliable web services compositions. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 59–72. Springer-Verlag, 2003.
13. Daniela Grigori, Fabio Casati, Malu Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Comput. Ind.*, 53(3):321–343, 2004.
14. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.