Transactional Behavior of a Workflow Instance

Tatiana A. S. C. Vieira and Marco A. Casanova

Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225
Rio de Janeiro, RJ - Brazil - CEP 22.453-900
Phone: +55-21-3114-1500 ext. 4347 / FAX +55-21-3114-1530

tascvieira@yahoo.com.br
casanova@inf.puc-rio.br

Abstract. Workflow management systems usually interpret a workflow definition rigidly, allowing no deviations during execution. However, there are real life situations where users should be allowed to deviate from the prescribed static workflow definition for various reasons, including lack of information about parameter values and unavailability of the required resources. To flexibilize workflow execution, this paper proposes an exception handling mechanism that allows the execution to proceed when otherwise it would have been stopped. The proposal is introduced as a set of extensions to OWL-S, the language adopted to define workflows, and is based on process and resource ontologies that capture the semantic information needed for the flexibilization mechanism and on the concept of abstract machine. In particular, this paper focus on the transactional behavior of a workflow instance, in the sense that it guarantees that either all actions executed by the instance terminate correctly or they are all abandoned.

1 Introduction

Workflow management systems, or workflow systems, for short, have a extensive list of requirements, from cooperation and coordination to synchronization. In this work, we focus on a requisite that we call *flexible execution*.

Usually, workflow systems execute a workflow instance rigidly, not allowing any deviation from the workflow definition at runtime. However, this rigid interpretation may sometimes lead to a long waiting time, among other problems.

We therefore propose a model for workflow instance execution based on a notion of workflow flexibilization that uses ontologies. For example, the flexibilization mechanism proposed allows a workflow instance to proceed execution without waiting for the availability of a resource or the value of a parameter.

We specify the mechanism using the notion of exception treatment. More importantly, we define all flexibilization concepts with the help of an abstract machine for process instance execution.

This paper is organized as follows. Section 2 summarizes related work. Section 3 presents some preliminary concepts for the understanding of the paper,

including a brief introduction about the flexibilization mechanism proposed. Section 4 presents the ontologies constructed to support that mechanism. Section 5 discusses the transactional behavior of a workflow instance. Finally, Section 6 presents the conclusions of this work.

2 Related Work

The evolution of workflow systems towards less restrictive coordination approaches is discussed in [11]. Most of the earlier proposals for flexibilization mechanisms suggest to dynamically change the workflow structure at runtime [3].

Rule-driven frameworks that support structural changes in a workflow, by dictating how tasks can be inserted or removed from the workflow at runtime, are presented in [8]. The flexibilization mechanisms described in [13] [1] also offer the user the possibility to include, remove or even stop activities during workflow execution. The mechanisms proposed in our paper follow a different strategy by allowing execution to proceed in the presence of incomplete or negative information, with minimal user intervention.

The OPENflow system [7] offers two distinct flexibilization approaches: flexibility by selection and flexibility by adaptation. Flexibility by selection allows several paths to be included in the workflow description, but only one path is chosen at runtime. Flexibility by adaptation leads to workflow adaptation, at the instance level, when changes occur in the workflow structure at runtime (inclusion of one or more execution paths). Our approach does not cover flexibility by selection, but it achieves flexibility by adaptation by using semantic information about workflow description to partly guide instance execution.

The flexibilization mechanisms in [6] account for the cooperation between users and the anticipation of task execution. The mechanism offers a special transaction model for cooperative systems that deviates from the conventional start-end synchronization model [2].

The construction of workflow schemas from a standard set of modeling constructs is proposed in [9]. This is partially done at design time, and completed at runtime, according to selection, termination and workflow definition constraints, which dictate how each fragment of work can be included in the workflow, under what conditions the workflow instance can be terminated and what conditions must hold during workflow definition. Flexibilization is achieved, in this case, by leaving workflow definition to be completed at runtime, according to the specified constraints. To some extent, our mechanism for choosing alternatives achieves a similar effect, as discussed in Section 3.

Lastly, the flexibilization mechanisms we propose bear some similarity to the query relaxation strategy adopted in the CoBase system [4] and in the CoSent system [5]. CoBase is a database system that uses the idea of cooperative queries. When queries are submitted to the system, CoBase analyzes a hierarchy of additional information to enhance the query with relevant information. The information added to the query is bounded by a maximum semantic distance from the

information present in the original query. This query modification mechanism is similar to the strategy we propose to deal with negative or incomplete information. In our proposal, the workflow description is enhanced with additional information available in the workflow ontology, as discussed in Section 4.

3 Preliminary Concepts

We briefly introduce in this section informal definitions for the terms and concepts that we will use throughout the paper.

3.1 Process and Process Instance

Following OWL-S [10], from now on, we use the terms *process* and *process instance* instead of the terms workflow and workflow instance.

A *process* is a representation of a set of activities, manual or automatic, that are composed to reach a common goal. Likewise, a *process instance* is the representation of a process definition execution, involving not only static information, but also data produced at runtime.

A process is *atomic* if it is indivisible, and *composite* if it is a composition of two or more processes, called *subprocesses* or *component* processes. A composite process is created with the help of *control constructs*, such as *join*, *split* and *sequence*.

A process may need *resources* to be executed. A resource may be *physical* or *logical* and must be allocated to the process instance before the instance starts executing.

A process also has *pre-* and *post-conditions*. A pre-condition defines in which condition the associated process can be executed. We assume that the availability of the necessary resources is an implicit precondition of a process.

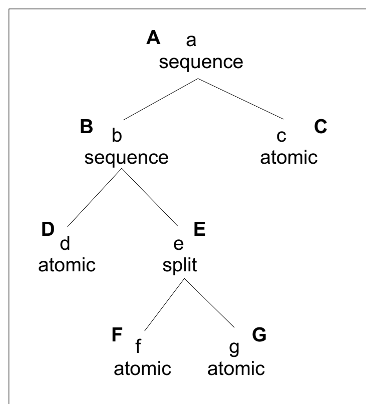

Fig. 1. Example of a composite process a.

In Figure 1, process a is a sequential composition of processes b and c; b is a sequential composition of processes d and e, and e is a split composition of processes f and g. Processes c, d, f and g are atomic processes. In this way, b, c, d, e, f, and g are subprocesses of process a, but just b and c are direct subprocesses of a.

We call A the *instance* generated by the execution of process a. So, A is *superinstance* of the instances B and C, generated, respectively, by processes B and C.

The *control flow* of an instance A is defined by the control constructs used to define process a and by the pre-conditions of its subprocesses.

An *abstract process* is a process that cannot be directly executed because it does not have any associated implementation. Hence, an abstract process, by itself, cannot generate any process instances. By contrast, a *concrete process* has an associated implementation and, consequently, may generate process instances.

3.2 Introduction to Flexibilization

The flexibilization mechanism we propose treats exceptions, as explained latter, allowing that:

- the definition of a process to be completed at runtime, depending on execution data;
- the execution of a process instance to proceed even when some resource is unavailable or when the value of a parameter is unknown.

The ontologies explained in the next section guide the flexibilization mechanism to:

- select concretizations for abstract processes and resources;
- compute default values for the parameters;
- find alternative processes and resources, used when the original processes or resources cannot be used;
- find processes to handle certain (types of) exceptions.

An *exception* is a situation that is not considered in the static process definition and that, once registered with the mechanism, can be properly treated.

Exceptions form a hierarchy (Figure 2), where an exception at level n can only be reached if an exception at level n-1 was reached, but it could not be treated.

At the first level of the hierarchy, we have three (types of) exceptions, namely:

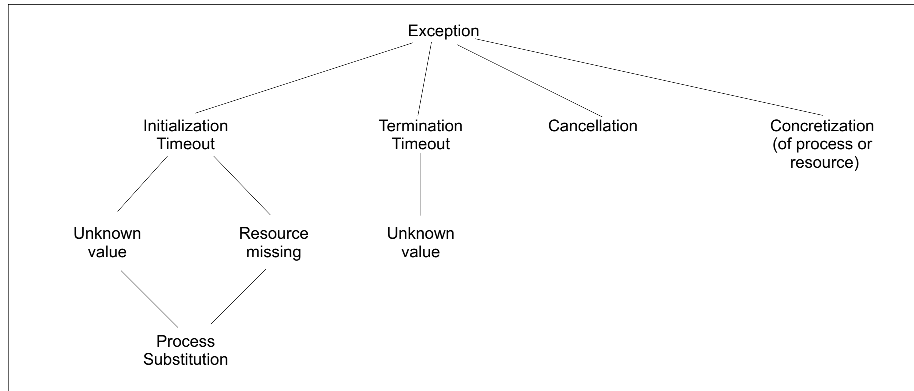

Fig. 2. Hierarchy of exceptions.

initialization timeout exception : the system throws this exception when it cannot initialize an instance execution because the value of a parameter is unknown or because a required resource is unavailable. In this case, the exception is treated by selecting for execution another process which was previously defined;

termination timeout exception : the system throws this exception when the instance cannot produce all its results and, thereby, it cannot terminate;

concretization exception : the system throws this exception when it finds, during instance execution, a reference to an abstract process or to an abstract resource. The exception treatment corresponds to finding a concrete process or a concrete resource which is associated with the abstract one;

cancellation exception : the system throws this exception when an atomic process instance is aborted. The treatment corresponds to undo the instance effects before aborting the instance.

Under the *initialization timeout exception*, we have two other exceptions that the system only throws if the flexibilization mechanism does not find any process to treat the *initialization timeout exception*:

unknown value exception : the system throws this exception when the *initialization timeout exception* cannot be treated because some parameter values are unknown at initialization time. The flexibilization mechanism treats this exception by finding default values for the parameters;

resource missing exception : the system throws this exception when *initialization timeout exception* cannot be treated because some required resources are unavailable at initialization time. The flexibilization mechanism treats

this exception by trying to find alternative resources that are semantically equivalent to the resources that could not be allocated to the instance.

The *termination timeout exception* indicates that the instance exceeded the allowed runtime, included in the process definition, without generating all output parameter values. If the flexibilization mechanism cannot treat this exception, it throws a second level exception (*unknown value exception*) and tries to use default values, this time for one or more output parameter.

A third level exception can be generated if an initialization or a *termination timeout exception* cannot be properly treated. In this case, the flexibilization mechanism tries to find alternative processes (*process substitution exception*).

Besides the concept of exception, our flexibilization mechanism depends on the concept of weighted semantic proximity, defined as a relationship between pairs of objects, processes or resources, with an attribute representing the weight.

For instance, let p and p' be two processes. We indicate that p' is an alternative process for p by defining a *semantic proximity relationship* between p and p'. The weight of the relationship indicates the similarity of p' and p. The flexibilization mechanism will use such weights to find the best alternative for p. It is important to pay attention to the fact that, if p' substitutes p, there must be a valid mapping between the parameters of p and p'.

Concretizations of abstract processes and resources are similarly modeled.

Default values can be treated statically by including them in the process definition, or dynamically through *presupposition rules*, taking into account the status of the process instance execution.

We say that an instance that is executing in response to an exception treatment is a *flexibilization instance*. However, it must be noted that any instance can the flexibilized, except for instances that undo the effects of aborted instances.

Furthermore, the subinstances of a given instance P are not directly defined by the static structure of p. They are rather defined as those instances that are under the control of P, after flexibilization. Instances that undo other instances do not have superinstances and cannot suffer flexibilization. In this sense, the flexibilization of an instance dynamically generates an instance tree.

4 Processes, Resources and Application Ontologies

The flexibilization mechanism we propose depends on additional information represented by mean of ontologies:

- *pr*, the central ontology of processes and resources, which is domain independent;
- *lib*, a library of process definitions and resource descriptions, which applications can reuse.

To define a process according to the vocabulary of the *pr* ontology, the designer can use the *lib* library. The process is defined in another namespace and refers itself to an application ontology.

The *pr* ontology must allow the definition of: abstract and concrete processes; atomic and composite processes; abstract and concrete resources; concrete processes that implement abstract processes and the weight of this relationship; which process(es) treat(s) the *initialization/termination timeout exception* of a process p; the execution cost of a process; the utilization cost of a resource; if a resource is or is not available; the default value of process parameters; and the parameter mappings between related processes.

Since OWL-S covers many of these aspects, we opted to define the *pr* ontology by extending OWL-S. The namespaces used to define the *pr* ontology and the application ontology are:

process : namespace of the OWL-S process ontology, http://www.daml.org/services/owl-s/1.1/Process.owl#

p-ext : namespace of the extended OWL-S process ontology, abbreviation of http://www.inf.puc-rio.br/ tati/tese/Process-extended.owl#

r : namespace of the OWL-S resource ontology, http://www.daml.org/services/owl-s/1.1/Resource.owl#

r-ext : namespace of the extended OWL-S resource ontology, http://www.inf.puc-rio.br/ tati/tese/Resource-extended.owl#

pr : namespace of the *pr* ontology, http://www.inf.puc-rio.br/ tati/tese/prOntology.owl#

lib : namespace of the *lib* library of processes and resources, http://www.inf.puc-rio.br/ tati/tese/prLibrary.owl#

app : abbreviation of the application ontology namespace.

Figure 3 shows the composition of the *pr* ontology and how it relates to the library of processes and resources and to the application ontology. It also presents the extensions (classes and resources) added at each ontology.

As the focus of this paper is not the *pr* ontology, we do not explain in details this composition and the extensions defined (see [12]). We just mention that, to define relationship weights, we used ternary relations and created ontology rules to complement the definition with information that could not be readily represented using OWL.

We also observe that, to clarify the semantics of OWL-S ad the extensions we propose, we defined in [12] an operational semantic for OWL-S, based on the concept of an abstract machine, guided by triggers. The next section presents a transactional model for process instances.

5 Transactional Model for Flexible Workflow Execution

This section presents the transactional model that dictates the way process instances are executed.

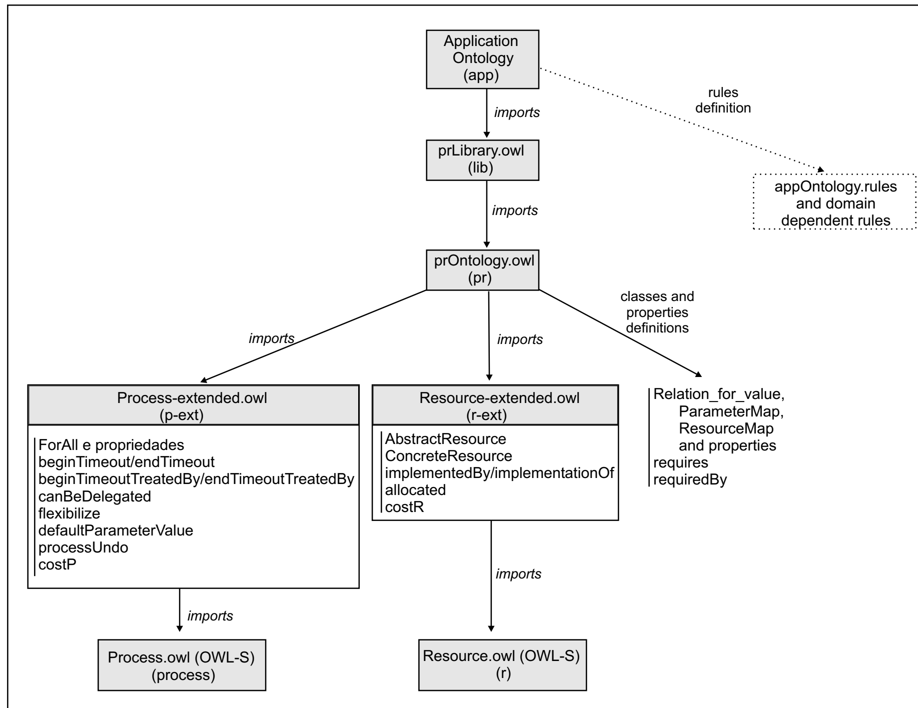

Fig. 3. *pr* ontology composition.

5.1 Instance Execution in the Abstract Machine

The specification of an operational semantic of OWL-S and its extensions through the definition of an abstract machine (AM) is justified for two reasons:

- the abstract machine uses very simple concepts, begin easy to understand;
- the definition of the abstract machine can be made incrementally, through successive extensions that incorporate new language constructs.

As shown in Figure 4, it is important to note that:

- each process instance P contains the representation of the corresponding process definition p, named *process*; a unique id, named *instanceId*; the identification of its superprocess instance, named *superInstance*; a tuple with the current state and the associated timestamp, and a set of attributed values for the defined parameters, called *blackboard*;

– each process instance is maintained in the *workspace area* of the abstract machine;

– the log associated with each instance is maintained in the *log area* of the abstract machine and register the execution steps;

– the abstract machine, in addition to the *workspace area* and the *log area*, consists of a *Controller*, responsible for instance management, and of an *Active Triggers Manager*. The triggers capture the semantics of atomic and composite processes, based on their control constructs, and guide the state transitions during instance execution. During the activation of a trigger, messages are sent by the *Active Triggers Manager* to the *Controller*, determining the instance processing.

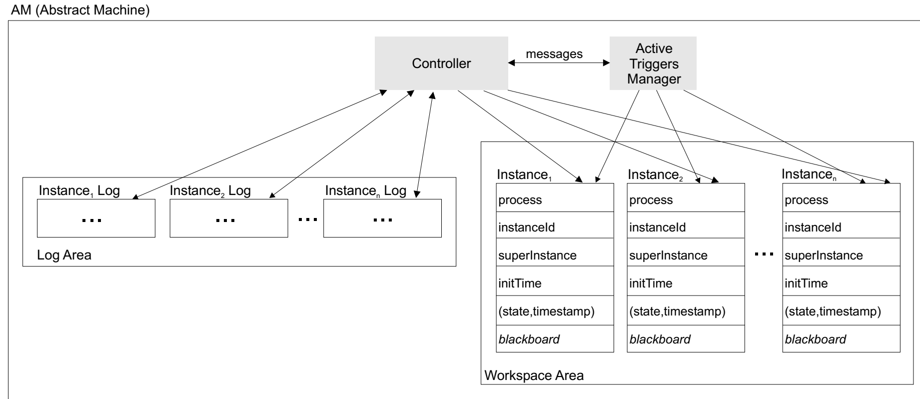

Fig. 4. Abstract machine (without extensions to flexibilization).

The abstract machine *AM* outlined above does not include the flexibilization mechanism. Under this proviso, the states traversed by a process instance are shown in Figure 5.

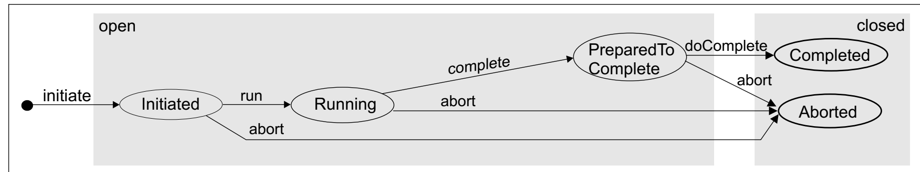

Fig. 5. State transition diagram for an instance P of a process p (OWL-S without extensions).

According to the above figure, a process instance is *closed* if and only if its state is *Completed* or *Aborted*, and it is *open* if and only if its state is *Initiated*, *Running* or *PreparedToComplete*.

The transactional behavior of an instance in *AM* is guaranteed by the following two properties, where P is a composite process instance:

Property 1 : if P terminates in the *Completed* state, then all direct subinstances of P also terminate in the *Completed* state;

Property 2 : if P terminates in the *Aborted* state, then all direct subinstances of P also terminate in the *Aborted* state.

By transactional behavior we understand that either all actions of an instance terminate correctly or all of them are abandoned and properly identified.

We note that Property 1 immediately implies that, if P terminates in the *Completed* state, then all direct and indirect subinstances of P also terminate in that state (and analogously for Property 2).

Since the abstract machine does not have strict control over the atomic actions of an instance, it does not automatically undo the effects of the abandoned actions (as in a database management system).

Since trigger definitions reference process preconditions and the OWL-S documents do not explicitly define the semantic of a process with multiple preconditions, we assume that, if there is more than one precondition, all of them must evaluate to true for the process to be executed. Also, we assume that a process cannot start executing unless all resources it requires are available and the values of all input parameters are known. We treat these conditions as implicit pre-conditions.

Moreover, in case of multiple instances of the same process, to capture the most recently terminated instance, each instance keeps the timestamp in which the current state was reached.

The dataflow in OWL-S is also unclear. This forced us to assume that, if a process p receives input values from the parameters of another process p', then p and p' must have a superprocess in common, i.e., they belong to the same process hierarchy.

5.2 Instance Execution in the Extended Abstract Machine

Figure 6 shows the extended abstract machine (*EAM*), incremented to accommodate the semantic of the flexibilization mechanism.

Note that the *EAM* contains a new component, the *Ontology Manager*, responsible for implementing the proposed flexibilization mechanism.

This mechanism is accessible through messages exchange from the *Controller* to the *Ontology Manager*. Likewise, the service response (i.e., the response for the accessed mechanism) is sent from the *Ontology Manager* to the *EAM Controller*.

Each instance P in *EAM* keeps additional information: the set of process subinstances of P; the instance type (if it is an instance executed in the predefined

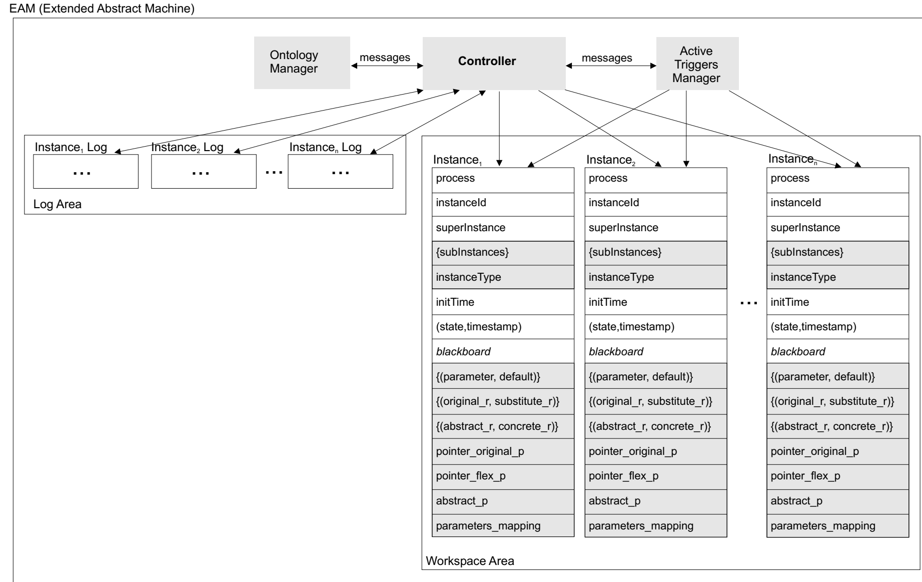

Fig. 6. Extended abstract machine (*EAM*).

order in the superprocess definition, or if it was generated by the mechanism of flexibilization, as explained below); the default values for process parameters; the resource substitutions performed; the resource concretizations performed; the instance that causes the flexibilization of the current instance (if it is the case); the process concretizations performed; and the parameter mappings between the processes of related instances. The information about resource mappings is not kept in the related instance because this information is not necessary in any future moment of the instance execution.

Each instance P' in the *EAM* has a type defined by an string, as follows:

"temporal" : when P' is related to a process which was selected to treat a timeout exception raised by another instance;

"concretization" : when P' is related to a concrete process which was selected to treat a *concretization exception* raised by an abstract process found in the superprocess definition;

"substitution" : when P' is related to a process which was selected to treat an exception that called for the substitution of a process;

"undo" : when P' is related to a process which was selected to undo the effects of a canceled atomic instance (*PreparedToAbort* state), as a result of a *cancellation exception*; and

"null" : when the instance P' is related to a process p' statically defined at the flow control of the superprocess definition of p'.

In the above three cases, we say that P' is a *flexibilization instance*. If the exception called for a process substitution, we say that P' *substituted* P.

The *pointer_original_p* entry of P' identifies the instance P for which the instance P' is executing; the entry *pointer_flex_p* of instance P refers to P'. If the *instanceType* value of P' is "null", the *pointer_original_p* for P' is also null.

We suppose that any flexibilization process instance P' also can be flexibilized, except for instances which are undoing the effects of an atomic process instance which was aborted.

It is important to remember that, in the presence of flexibilization, the subinstance definition is a bit different from the case with no flexibilization. With flexibilization, the subinstances of an instance P are not defined directly by the control structure of the process p of P. We consider as subinstances of P those instances defined in the entry *subInstances* of *P*. This entry is updated each time the instance suffers flexibilization. This means, as already mentioned, that flexibilization implies the dynamic generation of an instances tree, which contrasts with a tree statically defined by the control structure of the associated process.

Figure 7 shows the modified state transition diagram, adapted to support flexibilization. It has seven new states: *BeginTimed-out*, *EndTimed-out*, *Skipped*, *PreparedToByPass*, *Forced*, *PreparedToAbort* and *Undone*.

In the previous figure, the dotted arrows indicate that the corresponding messages are processed by the *Controller* during the execution of the superinstance of the instance P that had its state changed.

Just as we defined two properties associated with the behavior of instances in the *AM*, we also defined properties for instances in the *EAM* to accommodate transactional processing under flexibilization.

Hence, to support the undo of atomic process instances, the triggers managed by the *EAM* must guarantee the following properties, where P is a composite process instance (Property 1' is a modification of Property 1 previously presented).

Property 1' : if P terminates in the *Completed* state, then all direct subinstances of P terminate in the *Completed* or *Forced* states;

Property 2 : if P terminates in the *Aborted* state, then all direct subinstances of P terminate in the *Aborted*, *Undone* or *Forced* states and at least one subinstance, direct or not, of P terminates in the *Aborted* state;

Property 3 : if P terminates in the *Undone* state, then all direct subinstances of P terminate in the *Undone* or *Forced* states;

Property 4 : if P terminates in the *Forced* state, then all direct subinstances of P terminate in the *Aborted* or *Undone* states.

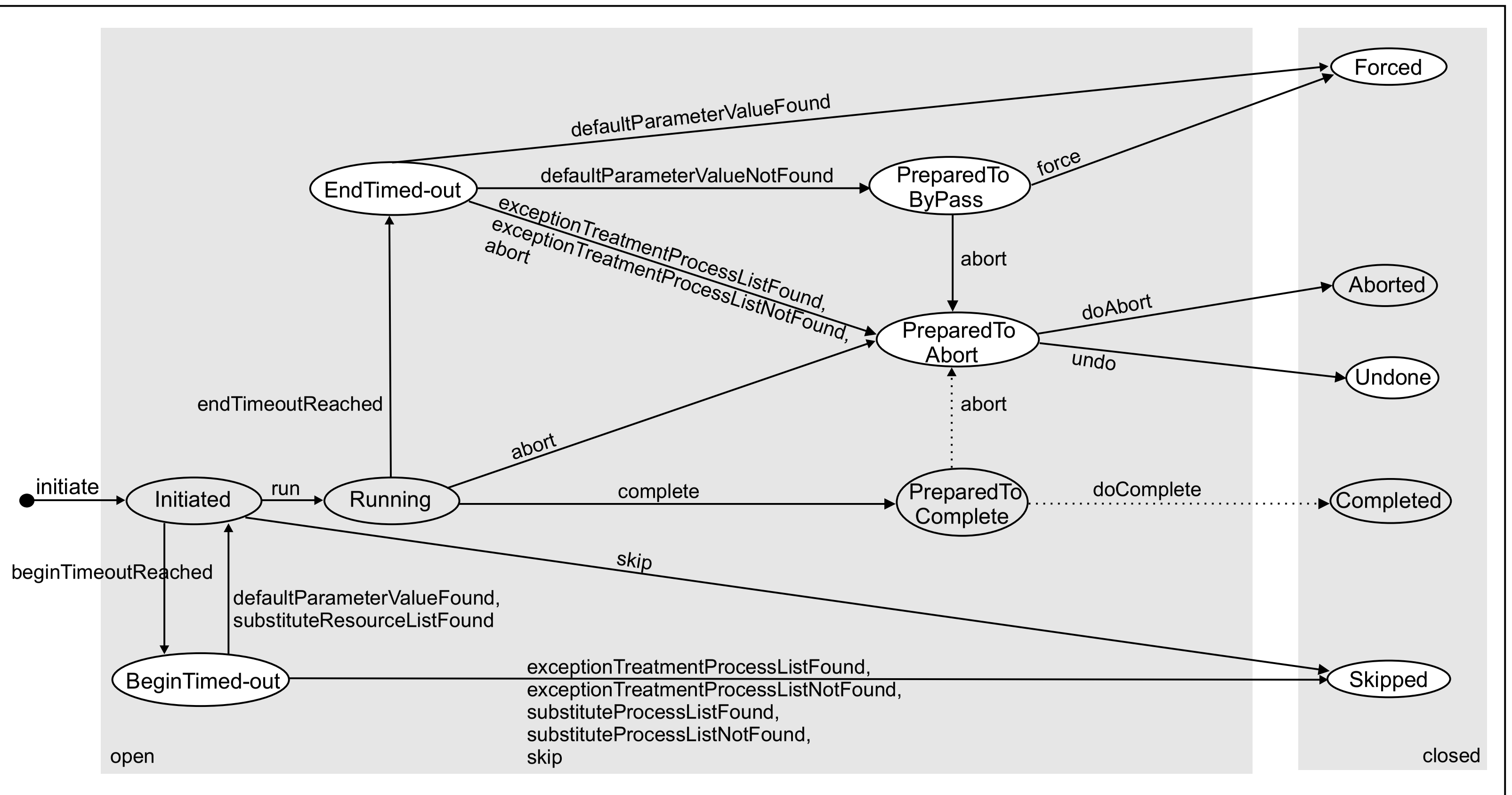

Fig. 7. Modified state transition diagram.

It is important to note that the subinstances of an instance P in *EAM* are the instances that in fact executed until they terminate. Consequently, the triggers which help instance processing, when associated with an instance P of a process p, verify the current state of the subinstances not directly created by P, but the state of the subinstances registered in the {*subinstances*} entry of P.

To accommodate the flexibilization mechanism, especially the new *undo* state, the triggers which in the *AM* verified if the current state of the subinstances of an instance P was *Aborted*, in the *EAM*, they verify if it is *PreparedToAbort*, since it is possible to undo atomic instances and, consequently, undo composite instances, through the properties already defined.

In addition, the triggers in the *EAM* responsible for completing the execution of an instance P do not just verify if their subinstances are in the *PreparedToComplete* state, but they also check if they are in the *Forced* state. The *Forced* state represents the fact that an instance terminated using default parameter values for their output parameters.

To find parameter values in the same data flow, when instance P' is the receiver of the values of instance P, the *Controller* of the *EAM* must traverse all the instances list formed by the *pointer_flex_p* of instance P, until reach the last flexibilization instance and compare its timestamp with the timestamp of all the (possible) another flexibilization instances of P. The most recently finished instance must be chosen.

Note that, when an instance P is flexibilized by P', the *Controller* must obey the information about parameter mappings saved at instance P'.

We explain the state transition diagram as follows. Basically, when an instance P of a process p is initialized, through the *initiate* message processed by the *Controller*, P reaches the *Initiated* state. If all its preconditions are satisfied, P reaches the *Running* state by the processing of the message *run* by the *Controller*.

According to the flexibilization mechanism, if in the *Initiated* state P reaches its static timeout defined for the initialization, the *Controller*, through message processing, puts instance P in the *BeginTimed-out* state.

From this state, instance P can reach the *Skipped* state, when a process p' of *initialization timeout exception* of p or if there is no way to execute P or a flexibilization instance of P. From the *BeginTimed-out* state, P can also go back to the *Initiated* state, if default parameter values were found for the missing value parameters, if it is the case, or if alternative resources were found and allocated to instance P execution, in the case of unavailability of the defined resources for p.

From the *Running* state, P can reach three distinct states:

- *PreparedToComplete*, if the execution of P finishes without any problem;
- *PreparedToAbort*, if instance P is aborted;
- *EndTimed-out*, if instance P reaches its statically defined time for termination.

From the *PreparedToComplete* state, P can reach two states, as follows:

- *PreparedToAbort*, when at least one of the subinstances of P reached the *PreparedToAbort* state;
- *Completed*, when all subinstances of P reached the *Completed* state or the *Forced* state.

For an instance to reach the *PreparedToComplete* state, the *EAM Controller* must verify, during the processing of message *complete*, if a conflict did not occur because of a possible use of default parameter value when the instance was in the *BeginTimed-out* state. If a conflict occurred, instance P is directly sent to the *PreparedToAbort* state.

From the *EndTimed-out* state, instance P can reached three states:

- *PreparedToAbort*, in case of a process p' of *termination timeout exception* of p was found, when P is aborted or when a process of exception treatment cannot be found and instance P cannot use default values for its output parameters;
- *PreparedToByPass*, when neither a process of exception treatment instance P nor default values for its output parameters cannot be found. So, P reaches *PreparedToByPass* state to the flexibilization mechanism try to find default parameters values for the superinstance of P, if it and P participate in the same data flow (if P receives as value for its output parameters values coming from its superinstance);
- *Forced*, if it was found default values for the output parameters of P that did not have their values known during execution.

The *PreparedToByPass* state is very special, because it is another way for the *Controller* to allow flexibilization. From that state, P can reach the *Forced* or *PreparedToAbort* states. P reaches the *Forced* state if default values were found for their output parameters, from the output parameters of its superinstance in the same data flow. In another hand, P reaches the *PreparedToAbort* state if those values could not be found.

From the *PreparedToAbort* state, an instance P can reach two different states:

- *Undone*, if P is an atomic instance, the flexibilization mechanism finds at least one process p' that treats the *cancellation exception* of P and instance P' created from p' executed with success and without flexibilization;
- *Aborted*, otherwise.

Following the transactional behavior of instances in the *EAM*, if P is an instance of a composed process, so its termination in the *Abort* or *Undone* state depends on the termination of its subinstances, as defined in Property 2 and in Property 4, respectively.

If all subinstances of P finished in the *Undone* or *Forced* state, then P also finishes at the *Undone* state.

Just to mention, we defined a log for each instance in the EAM, for registering information about instance execution and, specially, instance flexibilization. Besides saving the states reached during instance execution, the log saves information about the input and output parameter values of the instance, the concrete resources and processes used as implementations of the abstract resources and processes found at process definition, the default values used, the flexibilization instances and the resources and parameters mapping between the correspondent instances.

6 Conclusions

To flexibilize workflow execution, we proposed in this paper an exception handling mechanism that allows the execution to proceed when otherwise it would have been stopped. The proposal was cast as a set of extensions to OWL-S, and is based on process and resource ontologies that capture the semantic information needed for the flexibilization mechanism and on the concept of abstract machine. We focused on the transactional behavior of a workflow instance, in the sense that it guarantees that either all actions executed by the instance terminate correctly or they are all abandoned.

A distributed architecture using the EAM for the instance processing is described in [12]. This architecture considers several EAM, called *process managers*, to execute process instances.

Acknowledgment

This work was partially supported by CNPq, under grants 140600/01-9 and 550250/05-0.

References

1. Ilia Bider and Maxim Khomyakov. Is it Possible to Make Workflow Management Systems Flexible? Dynamical Systems Approach to Business Processes. In *Proceedings of the 6th International Workshop on Groupware (CRIWG' 2000)*, pages 138-141, Madiera, Portugal, October 2000.
2. G. Canals, C. Godart, F. Charoy, P. Molli, and H. Skaf. COO Approach to Support Cooperation in Software Developments. In *IEEE Proceedings in Software Engineering*, volume 145, pages 79-84, April/June 1998.
3. Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Workflow Evolution. In Bernhard Thalheim, editor, *International Conference on Conceptual Modeling / the Entity Relationship Approach (15th ER' 96)*, pages 438-455, Cottbus, Germany, October 1996. Lecture Notes in Computer Science.

4. Wesley W. Chu, Q. Chen, and M. Merzbacher. *Studies in Logic and Computation 3: Nonstandard Queries and Nonstandard Answers*, volume 3, chapter CoBase: a Cooperative Database System, pages 41-72. Oxford University Press, New York, 1994. Edited by R. Demolombe and T. Imielinski.
5. Wesley W. Chu and Wenlei Mao. CoSent: a Cooperative Sentinel for Intelligent Information Systems, March 2000. Computer Science Department - University of California, LA.
6. Daniela Grigori, François Charoy, and Claude Gobart. Flexible Data Management and Execution to Support Cooperative Workflow: the COO Approach. In *Proceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications (CODAS 2001)*, pages 124-131, April 2001.
7. J. J. Halliday, S. K. Shrivastava, and S. M. Wheater. Flexible Workflow Management in the OPENflow System. In *Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference (EDOC '01)*, pages 82–92. IEEE, September 2001.
8. G. Joeris. Defining Flexible Workflow Execution Behaviors. In *Enterprise-wide and Cross-enterprise Workflow Management - Concepts, Systems, Applications, GI Workshop Proceedings - Informatik '99*, pages 49-55, 1999. Ulmer Informatik Berichte Nr. 99-07.
9. Peter Mangan and Shazia Sadiq. On Building Workflow Models for Flexible Processes. In *ACM International Conference Proceeding Series - Proceedings of the Thirteenth Australasian Conference on Database Technologies (ADC'2002)*, volume 5, pages 103-109, Melbourne, Australia, January/February 2002. Australian Computer Society, Inc. Darlinghurst.
10. David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. W3C Member Submission, November 2004. http://www.w3.org/Submission/2004/SUBM-OWL-S20041122/Overview.html.
11. Gary J. Nutt. The Evolution Toward Flexible Workflow Systems. In *Distributed Systems Engineering*, volume 3, pages 276-294, December 1996.
12. Tatiana A. S. C. Vieira. *Execução Flexível de Workflows*. PhD thesis, Department of Informatics - Pontifical Catholic University of Rio de Janeiro, Brazil, Rio de Janeiro, RJ - Brazil, August 2005. In Portuguese.
13. Mathias Weske. Flexible Modeling and Execution of Workflow Activities. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 713-722, January 1998.