Using Patterns to parameterize the execution of Collaborative Tasks

Mamadou Lakhassane Cisse, Hanh Nhi Tran, Samba Diaw, Bernard Coulette, Alassane Bah

Using Patterns to parameterize the execution of Collaborative Tasks

Mamadou Lakhassane Cisse
University of Toulouse Jean Jaures
IRIT Laboratory
Toulouse, France
mamadou.cisse@irit.fr

Hanh Nhi Tran
Paul Sabatier University
IRIT Laboratory
Toulouse, France
hanh-nhi.tran@irit.fr

Samba Diaw
Cheikh Anta Diop University
UMMISCO
Dakar, Senegal
samba.diaw@ucad.edu.sn

Bernard Coulette
University of Toulouse Jean Jaures
IRIT Laboratory
Toulouse, France
coulette@univ-tlse2.fr

Alassane Bah
Cheikh Anta Diop University
UMMISCO
Dakar, Senegal
alassane.bah@ucad.edu.sn

***Abstract*—During the execution of a process, managing the collaboration inside a task performed by various actors is not straightforward due to possible changes of the process's context and the collaboration strategy. Process management solutions which describe the collaboration at modeling time offer a rigid control for conducting such collaborative tasks and thus cannot adapt to changes. To enable a flexible execution of collaborative tasks, we propose using the late-binding mechanism to allow process actors, at execution time, choosing or adapting strategies to perform their collaboration. To do so, first we model collaboration strategies as process patterns providing different ways to implement a collaborative tasks at execution time. These collaboration patterns describe how to establish necessary relations for coordinating different instances of the task, for sharing and exchanging working artifacts among actors performing those instances. Then we define actions to execute collaborative tasks. These actions take collaboration patterns as parameters. Thus, by letting process actor selecting a suitable collaboration pattern, they allow binding dynamically a collaborative task to its implementation flexibly.**

***Index Terms*—Collaboration Modeling, Process Execution, Multi-Instance Task, Collaboration Process Pattern**

I. Introduction

Conventionally, a task is the smallest unit of work in a process subject to management accountability. Existing process management systems focus on coordinating different process's tasks but pay less attention to managing the collaboration of different actors inside a given task to achieve a common goal. We are interested particularly in a special form of collaborative task, so-called *multi-instances task* (MIT) which is a task performed by a group of actors having the same role. Thus, at execution time, it can have multiple instances, each instance being performed by one actor and all instances participating to the completion of the task. As an example, in the RUP based process [1] applied to analyze a complex system, the task *DescribeUseCases* would typically be a collaborative task performed by a set of engineers playing the role *Analyst* to detail different use cases.

A collaboration strategy defines how instances of the same task are executed, i.e. the order to execute the instances, and the way the task's inputs and outputs are shared among the instances [2]. In principle, a collaborative task can be performed with different strategies based on different contexts (dependencies among artifact's components, availability of actors, etc). In practice, if the collaboration strategy is described in the process model, the relations between the task instances are already given at modeling time and cannot change at execution time without modifying the process model or deviating from the original process. The consequence of such rigid relations is that the actors cannot adapt the collaboration strategy to fit to the evolution of the execution context (for example, adding or removing an actor).

The objective of our work is to enable executing collaborative tasks in a controlled but flexible way. To enable a fine-grained control of collaboration during process execution, both the structural and behavioral aspects of collaborative tasks must be known. To enable a flexible execution, these aspects of a collaborative task are given in two times: at modeling time, only the task's structural elements (e.g. performing role, used artifacts) are described, then at execution time, when the task is instantiated into several instances performed by various actors, the relations among the task's instances (e.g. work-sequences, exchanged-data) will be specified.

The challenge for the chosen approach is how to generate at execution time the behavioral model of a collaborative task without requiring process actors to go back to the modeling phase. Dealing with this question, first we define a *collaboration pattern* as a pattern capturing a collaboration strategy which determines typical relations among the instances of a multi-instance task at execution time. Then, we use the late-binding mechanism to let actors selecting dynamically a suitable collaboration pattern and use it as a template to generate the inter-instances relations for the collaborative task. To enable adapting the execution of a collaborative task to

DOI 10.1109/WETICE.2019.00031

the evolution of project's context, during the execution, the collaborative task can change its collaboration strategy by selecting another bound collaboration pattern.

The paper is structured as follow. Section II recalls the concept of collaboration pattern and how it is used to describe the implementation of a collaborative task at execution time [2]. Section III describes the main contribution of this paper: a process engine allowing the late-binding of collaboration patterns to make the execution of collaborative tasks flexible. A prototype implementing our work is presented in Section IV. We discuss in Section V some related works and Section VI concludes the paper and presents some perspectives.

II. Collaboration Patterns

A strategy providing a recurrent solution to perform a collaborative task in a specific context can be captured as a *collaboration pattern*. A collaboration pattern describes the typical relationships among the instances of a collaborative task at execution time from two main perspectives: the control-flow and the data-flow. The control-flow perspective provides the execution order of different task instances, represented by the work-sequence relations among the instances. The data-flow perspective specifies, via the task parameter relations, the data manipulated, exchanged or shared by the task's instances.

In contrast to the modeling patterns proposed in [3], [4] and [5] that are applied at modeling time for describing collaboration scenarios, collaboration patterns presented in this paper are applied dynamically at execution time to generate the detailed model of running collaborative tasks.

In our approach, at modeling time a process is partially defined: the process model contains only structural elements. The missing part, the elements defining the process's behavior, will be completed at execution by applying a collaboration pattern. Considering a collaborative task T in a process, its model can be seen as a parameterized function *TM(c:CollaborationPattern)*. c will be bound later to a concrete pattern cp selected by the process manager. Executing the task T means *run(TM(cp))* to create the task instance TI. TI is defined with two sources of information: structural elements of TI are instantiated from the task model TM, behavioral elements of TI are created by applying the pattern cp.

We defined the language named ECPML [2] to model collaborative processes. Beside the standard process concepts, this language distinguishes in particular two types of task: a *SingleTask* which has only one instance at execution time and a *CollaborativeTask* which can have several instances at execution time. The task model TM mentioned above is represented as a *CollaborativeTask* in ECPML.

Aiming at developing a built-in operational semantics for ECPML, in Fig. 1, we present the concepts used to represent the dynamic instances created at execution time from the elements defined in a process model. Concretely, a *Task* in a process model will be instantiated into one or many *TaskInstance* at execution time. A *TaskInstance* is enacted by an *Actor* who plays a *Role* and produces or uses some *WorkProductInstances* (WPI).

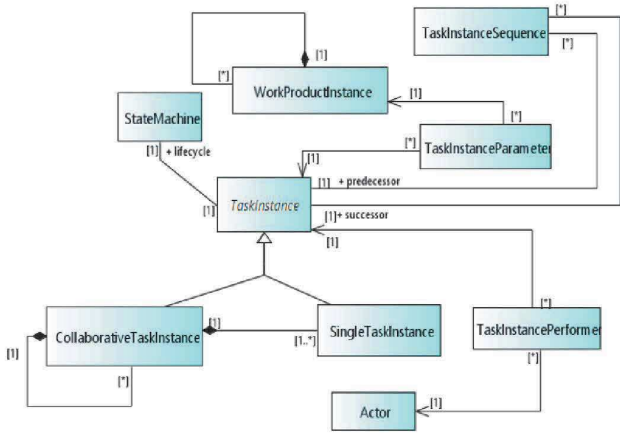

Fig. 1. Metamodel defining instances of the concepts in ECPML metamodel

Focusing on controlling the execution of a multi-instance task performed by several actors, we distinguish *SingleTask-Instance (STI)*, which is an instance of *SingleTask*, and *CollaborativeTaskInstance (CTI)*, which is an instance of *CollaborativeTask*. An *STI* is the main executable element representing a unit of work assignable to a single actor. A *CTI* is composed of several *STIs* performed by separate actors. The dynamic semantics of a $TaskInstance$ is given via a $StateMachine$ describing the states and the transitions during the life cycle of a task instance.

TaskInstanceSequence relations can be established among *TaskInstances* to synchronize their execution. For a *CollaborativeTask*, such relations among its instances are not given at modeling time but can be generated by using a template provided in a *collaboration pattern*. Thus, we use the concepts at instance level as presented in Fig.1 to model the solution of a collaboration pattern. For example, the pattern cp mentioned previously captures a model of task instances which contains elements of the metamodel in Fig. 1.

We have identified several patterns based on the way the manipulated artifacts are shared. In this paper, we consider only how the output artifacts that are changed by the collaborative task are shared among its instances. The input artifacts are implicitly considered as *"read-only"* items shared by the instances inside the collaborative task and are not shown in the patterns. In the following, we present 2 representative patterns corresponding to the two main types of execution: in parallel and in sequence.

1) PAR-INSTANCES-COP *(Pattern Parallel Instances with Composite Out Parameter):* Given a collaborative task *T* having one output parameter *P* composed of *n* independent components P_i, this pattern is used to execute a set of n STIs t_i inside the CTI of T simultaneously. Each task instance t_i will manipulate separately an instance p_i of the component P_i. Fig. 2 shows the solution of *PAR-INSTANCES-COP* as a model of the collaborative task T with 2 task instances.

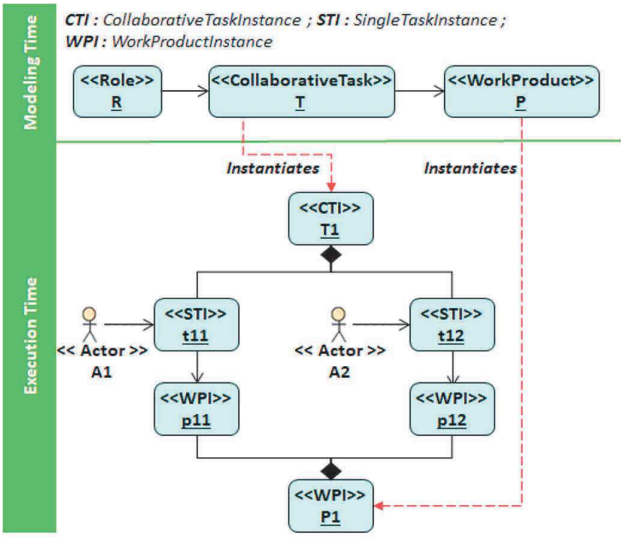

Fig. 2. Collaboration pattern PAR-INSTANCES-COP for a collaborative task T with two instances at execution.

2) SEQ-INSTANCES-COP *(Pattern Sequential Instances with Composite Out Parameter):* Given a collaborative task T having one output P which is composed of n dependent components P_i, $i \in [1, n]$, this pattern is used to execute a series of n consecutive STIs t_i, $i \in [1, n]$ inside the CTI of T.

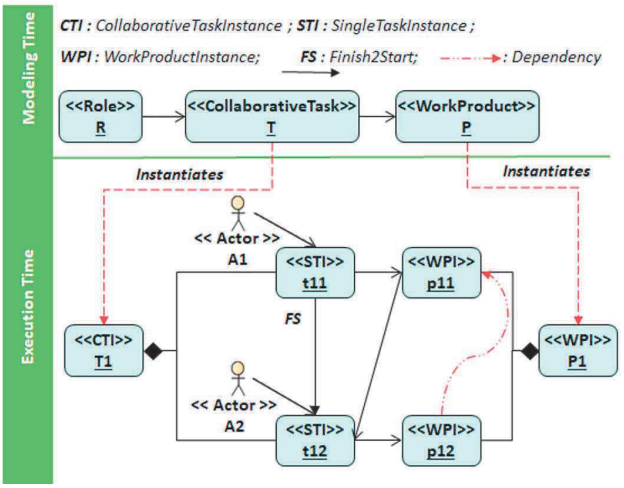

Fig. 3. Collaboration pattern SEQ-INSTANCES-COP for a collaborative task T with two instances at execution.

Each STI t_i manipulates an instance p_i of the component P_i and is performed by actors playing the same role. The execution order FS among the STIs is imposed by the dependencies among the components of P: the creation of $P_i + 1$ needs the completion of P_i thus $t_i + 1$ (which works on $P_i + 1$) has to follow t_i (which produces P_i).

III. EXECUTING COLLABORATIVE TASKS

As a dynamic entity, a *TaskInstance* has a lifecycle composed of different states through which it goes when executed. To allow deploying and executing an MIT, we need to define the task instance's lifecycle, its operational semantics. This section defines the operational semantics, presented by the state machine associated to *TaskInstance* in our meta-model. It allows to instantiate a task and makes its dynamic instances evolve flexibly during the execution of the task.

While the lifecycle of an STI can be defined with the conventional operational semantics of tasks having one instance at execution, our new concept CTI requires a specific operational semantics that enables its flexible execution. Fig. 4 presents the simplified CTI's state machine defining the different states during the lifecycle of a collaborative task. There are six events that trigger the CTI's state transitions:

- **TaskCreation (Task T)**: when this event occurs, the action *createCollaborativeTaskInstance(T)* will be executed to create a CTI node presenting an instance of the collaborative task T in the state *Instantiated*. As discussed in the previous section, we use the late-binding mechanism to apply a collaboration pattern to a CTI at execution time in order to obtain dynamically the sequencing of the different $STIs$ inside the CTI. The novelty of our proposition consists in not defining rigidly the actions on state transitions of the CTI's state machine. Rather we define them as parameterized functions taking collaboration patterns as effective parameters to complete the CTI's operational semantics.
- **TaskAssignment(CTI cti, Actor a[])**: this event occurs when actors are assigned to perform the CTI. The associated action *assignTask(cti, cp)* makes the relations between an actor and the STI that he performs inside the CTI and puts the task instance into the state *Assigned* During this transition, the relations among the $STIs$ inside a CTI are created according to the pattern cp received.
- **TaskStarting(CTI cti)**: when the assigned actors start to perform the collaborative task, this event occurs. If the condition for starting the task is verified, the associated action *startCollaborativeTaskInstance(cti)* puts the task instance into the state *InProgress*.
- **PatternChange(CTI cti, Pattern cp)**: during the execution of a CTI, process actors may want to change the task's collaboration strategy, i.e. select another collaboration pattern to carry out the CTI. In that case, this event happens and the action *applyPattern(cti, cp)* will be executed to apply a new collaboration pattern cp to the running collaborative task instance cti.
- **TaskEvolve(CTI cti)**: when the need to transform an STI to a CTI occured, this event happens. The action associated, *applyEvolution(cti, cp)*, allows to make the transformation and to apply a newly chosen pattern to the obtained CTI.

- **TaskFinishing(CTI cti)**: this event occurs when process actors terminate the task instance's actions.

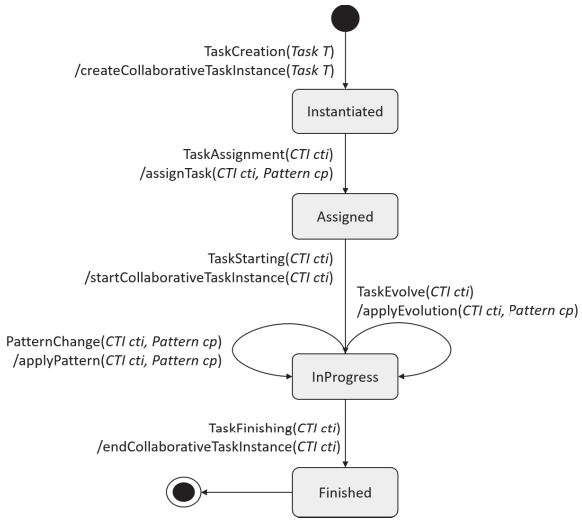

Fig. 4. Lifecycle of a CollaborativeTaskInstance.

After the deployment of a process, the instances of process's elements, including $STIs$, $CTIs$, actors, product instances, are stored in the process management system (PMS)'s runtime database (*Instances Store*, c.f. Section IV). The operational semantics defined via the state machines of the process's elements defines the behavior of the PMS and allows it updating correctly the running instances of the process.

Following, we describe the algorithms of two important actions in the CTI's lifecycle: the application of a collaboration pattern to carry out the CTI and the evolution of an STI into a CTI.

A. Applying a Collaboration Pattern to a CTI

We present in fig. 5 a sample of RUP modeled in ECPML. It contains one single task BuildUseCase and a collaborative task DescribeUseCase each one being performed by actors playing the role Analyst. The first task produces the use cases diagram, used as input in the second task. At the end of this process, the use cases description is obtained as a set of scenarios.

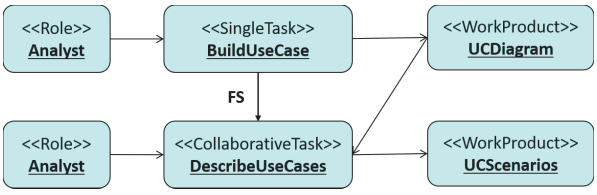

Fig. 5. Extract of the RUP in ECPML at modeling time.

Algorithm 1 shows the main steps of the action $applyPattern$ to apply a collaboration pattern cp to a cti. It uses, as a template, the relations among the $stis$ inside a cti to reproduce them on the $stis$ of the given cti.

Algorithm 1: applyPattern(cti, cp)

Input: CollaborativeTaskInstance cti, CollaborationPattern cp;
Output: SingleTaskInstance[] sti;
begin
  $wsType$ = identifyPatternWorkSequenceType(cp);
  $dfType$ = identifyPatternDataFlowType(cp);
  sti = cti.linkedSTI;
  for $i = 1$ **to** $sti.length - 1$ **do**
    applySequencing($sti[i]$, $sti[i+1]$, $wsType$);
    applyDataFlow($sti[i]$, $sti[i+1]$, $dfType$);
  end
end

Here, *identifyPatternWorkSequence(cp)* and *identifyPatternDataFlow(cp)* allow to recognize respectively the worksequence type and the data-flow type of inter-instances relations defined inside the pattern cp. *applySequencing(sti1, sti2, wsType)* and *applyDataFlow(sti1, sti2, dfType)* establish the identified relation types between two $STIs$ of the CTI. After repeating this procedure, every STI of cti is linked to the next one according to the sequencing defined in the pattern cp.

Applying a collaboration pattern to a CTI can be done at task instance's creation, during its execution to change the collaboration strategy used for carrying out the task or during the evolution of an STI.

Fig. 6 gives the final result after creating a CTI for the task *DescribeUseCases* and applying the collaboration pattern *SEQ-INSTANCES-COP* to this CTI. This pattern represents a sequential strategy to execute the collaborative task, thus the worksequence between the three STIs, *DUC1, DUC2, DUC3* is *FinishToStart*.

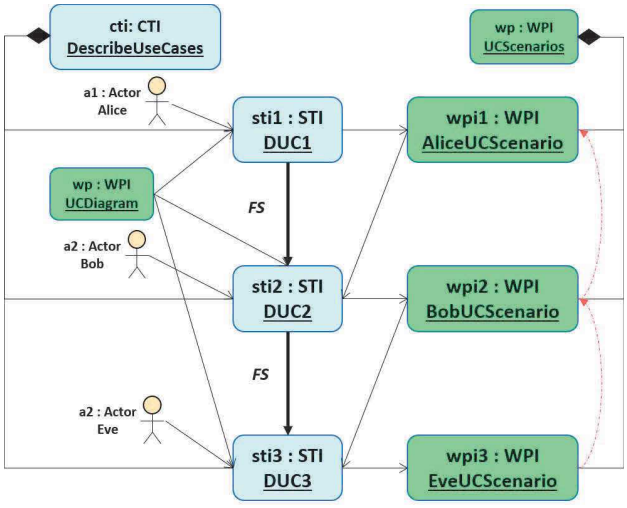

Fig. 6. DescribeUseCases after applying the pattern SEQ-INSTANCES-COP.

B. Evolving an STI into a CTI

At any time during execution, it would be useful to consider a single task as a collaborative one. This functionality

reinforces the dynamic flexibility of our approach. Algorithm 2 shows the different steps for the evolution of an STI into a CTI. During the application, a collaboration pattern is selected to be applied to the newly created CTI.

Algorithm 2: applyEvolution(cti, cp)

Input: SingleTaskInstance sti, CollaborationPattern cp;
Output: CollaborativeTaskInstance cti;
begin
  cti = changeNodeType(sti)
  sti.linkedCTI.add(cti)
  sti.linkedCTI = null
  applyCTIDataFlow(cti)
  applyCTISequencing(cti, sti)
  applyPattern(cti, cp)
end

During the execution of this algorithm, a new CTI is created through *changeNodeType(sti)*. Afterwards, it is linked to the former CTI linked to the STI to be changed. *applyCTIDataFlow(cti)* and *applyCTISequencing(cti, sti)* allow to establish respectively the relations between the new CTI and the input/output if existing and the relation between the new CTI and the successor and predecessor of the STI to be changed. The pattern cp is used to apply a collaboration pattern to cti. For that purpose the algorithm *applyPattern(cti, cp)* defined on algorithm 1 is used. For example, considering the process in fig. 6, actors can be dealing with constraints forcing them to split the third STI into two or more. In this situation, *DUC3* will evolve into a CTI requiring a pattern for its execution. Fig. 7 gives the final result after evolution of the third STI and application of the pattern *PAR-INSTANCES-COP* for the new CTI, thus no worksequence between the STIs inside that CTI.

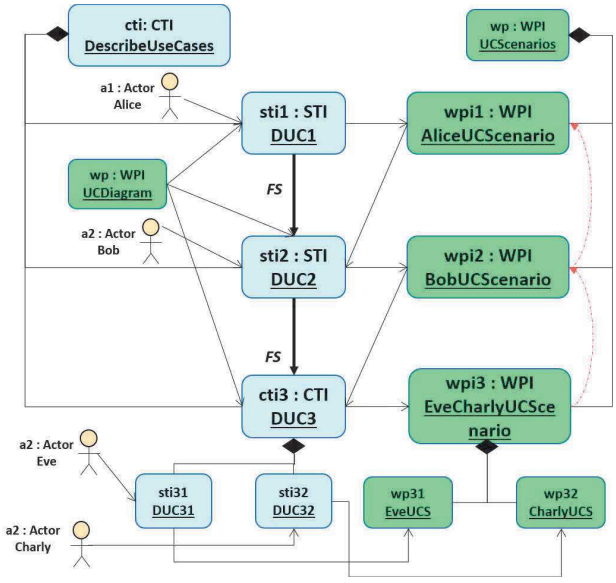

Fig. 7. DescribeUseCases CTI and its linked instances after evolution of the third STI into a CTI.

The state machines defining the operational semantics of ECPML presented in this section have been implemented and tested in a prototype.

IV. IMPLEMENTATION OF A PROTOTYPE

As introduced in [6], we have developed a *CPE (Collaborative Process Engine)* supporting flexible execution of MITs. It allows users to chose the collaboration patterns corresponding to an actual execution context. Since, CPE has been enriched with the implementation of the algorithms defined in section III. Fig. 8 below describes the general architecture of CPE.

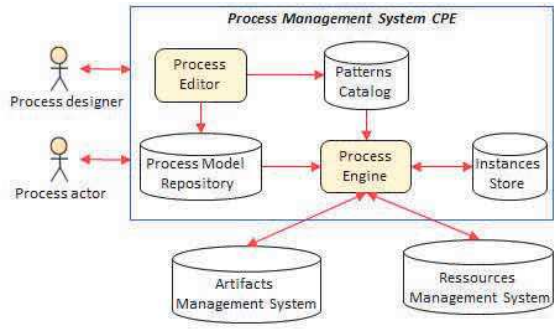

Fig. 8. General architecture of CPE.

The two main components of CPE are the **Process Editor** allowing process designers to model processes that are stored in the *Process Model Repository* using ECPML and the **Process Engine** helping process actors to perform their process. This latter updates the process's instances during execution by mean of their operational semantics.

The physical artifacts and human resources manipulated during the process execution are managed by external *Databases*: *Artifacts Management System* for artifacts and *Resources Management System* for actors. These databases are connected to CPE which manages just the references of artifacts and actors inside its $InstancesStore$.

Thanks to CPE, the project manager can monitor the execution of collaborative tasks and adapt the collaboration strategy for conducting collaborative tasks at any moment according to the alteration of project's constraints and needs. CPE provides process actors with not only the necessary functionalities to perform their task (by verifying the condition to create, start, end or assign resources to a task instance) but also a global and real-time view on the progress of development tasks (by showing the information about the collaborative task that he participates in: what is the current state, who are other actors performing the task, what are the exchanged data, etc.).

Although CPE is helpful for all kinds of processes which have multi-instance tasks, it can benefit particularly system and software processes which are often performed by several teams to produce different parts of the final product. Moreover, generally system and software processes' projects have changing contexts because of the evolution of product's specification as well as the evolution of production's constraints. The above

characteristics make system and software processes require more assistance during their execution - as offered by CPE.

V. RELATED WORK

The need for process flexibility has often been addressed in literature. In the workflow and process technology communities, a process is considered flexible if it is possible to change it without replacing it completely [7]. This definition is not different from our approach since we do not intend to change the process itself but allow a flexible execution of it. The work in [8] introduces some premises of how process-based applications could be. They expressed the ability to deal with unpredictable situations by allowing the process model to be partially unknown at design-time and refined at runtime.

Works, such as [9] and [10] are focused on flexible execution through deviations management. They rely on PSEE (Process-Centered Software Engineering Environment) to detect and tolerate agent deviations. In contrast, we do not allow deviating from the original process model, rather we use the late-binding to precise the way of executing during enactment.

Reference [11] introduced a Workflow management system aiming at supporting cooperative work, and among these requirements were high flexibility and dynamicity. They proposed an approach allowing users to modify the instance of a process, such as adding an activity and starting an activity even when the activation conditions are not met. While it allows a certain flexibility, it cannot be applied to processes that require a fine-grained control of the dataflow. In order to achieve flexibility by looseness, [12] proposes DECLARE, a constraint-based system for supporting loosely-structured process models. As for [13], it proposed a process-aware CSCW system supporting process schemas that are created on-the-fly. AristaFlow [7] allows flexibility through process composition and ad hoc changes of single process instances.

Compared to the cited works, we also adopt the late-binding, but propose to use dynamically patterns to parameterize the behavior of the process engine and thus make it flexible.

VI. CONCLUSION

Our current research focuses on the flexible management of collaborative processes. Our work targets the modeling and execution of collaborative tasks. The work presented in this paper considers in particular multi-instance tasks (MIT) which are instantiated several times at execution and performed by different actors but all collaborating to produce a common result. The novelty of our approach is providing a solution to model partially MITs and then using the late-binding to complete the tasks behavior flexibly at execution time.

We have proposed a set of collaboration patterns describing the typical behavior models of MITs. Then we have defined the operational semantics of the different executable elements inside a process, especially collaborative tasks. This operational semantics allows binding a pattern to a collaborative task instance during its execution to deploy a collaboration strategy. By taking collaboration patterns as parameters of a task's execution, we enable a more flexible way to enact a collaborative task. The collaboration strategy can be changed at any time during the task's execution. Moreover, we allow the evolution of a single task into a collaborative task so that process actors can delegate their works when necessary.

To improve the validation of our approach, we need to apply it to other case studies and especially to real projects. Adding new collaboration patterns is also desirable but the limited set of collaboration patterns implemented, so far, does not question the validity of our approach.

We aim also supporting more complex collaborative task behaviors. Currently, we only deal with patterns describing one kind of work-sequence relations among the single task instances of a collaborative task (for example $Finish2Start$). However, sometimes in practice there are many kinds of inter-instances relations inside a task. To support more complex collaborations, we intend to investigate the proposition of new patterns covering those situations. We explore also the possibility of automatically recommending collaboration patterns to the project manager based on project's context analysis.

REFERENCES

[1] P. Kruchten, *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[2] M. L. Cisse, H. N. Tran, S. Diaw, B. Coulette, and A. Bah, "A pattern-based process management system to flexibly execute collaborative tasks," in *14th International Conference on Evaluation of Novel Approaches to Software Engineering*. Heraklion, Crète, Grèce: SCITEPRESS, 2018, in press.

[3] J. Lonchamp, "Process model patterns for collaborative work," in *Proceedings of the 15th IFIP World Computer Congress, Telecooperation Conference, Telecoop'98*, Vienna, Austria, 1998.

[4] W. M. V. der Aalst, A. H. T. Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," 2003.

[5] T. T. Vo, B. Coulette, H. N. Tran, and R. Lbath, "An approach to define and apply collaboration process patterns for software development," in *Third International Conference, MODELSWARD 2015, Angers, France, Revised Selected Papers. Model-Driven Engineering and Software Development, CCIS, Vol. 580, Springer*, December 2015.

[6] M. Cisse, H. Tran, S. Diaw, B. Coulette, and A. Bah, "Collaborative processes management: from modeling to enacting," in *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2018, pp. 461–466.

[7] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012.

[8] H. Ariouat, E. Andonoff, and C. Hanachi, "Do process-based systems support emergent, collaborative and flexible processes? comparative analysis of current systems," *Procedia Computer Science*, vol. 96, pp. 511–520, 2016.

[9] M. da Silva, X. Blanc, and R. Bendraou, "Deviation management during process execution," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 528–531.

[10] M. Smatti and M. Nacer, "Dealing with deviations on software process enactment: Comparison framework." in *ICAASE*, 2014, pp. 108–115.

[11] F. Charoy, A. Guabtni, and M. Faura, "A dynamic workflow management system for coordination of cooperative activities," in *International Conference on Business Process Management*. Springer, 2006, pp. 205–216.

[12] M. Pesic, H. Schonenberg, and W. M. V. der Aalst, "Declare: Full support for loosely-structured processes," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. IEEE, 2007, p. 287.

[13] S. Dustdar, "Carambaa process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams," *Distributed and parallel databases*, vol. 15, no. 1, pp. 45–66, 2004.