Using LOTOS for Rigorous Specifications of Workflow Patterns

(Invited Paper)

Pedro L. Takecian and João E. Ferreira
IME, University of São Paulo
Rua do Matão 1010, 05508-090
São Paulo, Brazil
Email: {plt, jef}@ime.usp.br

Simon Malkowski and Calton Pu
CERCS, Georgia Institute of Technology
266 Ferst Drive, 30332-0765
Atlanta, USA
Email: {zmon, calton}@cc.gatech.edu

***Abstract*—Collaborative applications require understanding of the theoretical foundations. In case of workflow systems, one possibility to achieve this is an accurate description of workflow functionalities. Despite its growing popularity and success, it has not yet been evaluated whether Language of Temporal Ordering Specification (LOTOS) is actually suitable for representing comprehensive workflow functionality in real-world workflow systems describing the behavioral perspective of control-flow. Our primary contribution is the complete mapping of a collection of workflow patterns to LOTOS notation. We further discuss suitability and expressiveness of LOTOS in the context of workflow patterns. This study can be used for pattern-based workflow execution, reasoning, and simulation as well as for future research on theoretical aspects of workflows for collaborative applications.**

I. Introduction

It is well understood that embedding workflow systems in formal frameworks is highly beneficial during their entire life cycle [1]. Sound models as a foundation for workflow systems enable the use of a battery of sophisticated methods for verification, validation, diagnosis and execution control. Despite the popularity of standards such as BPMN [2] and BPEL [3], there is little consensus on the "right" choice of formalism [4], [5]. Advocators of modern process algebra as foundation for corporate workflow emphasize features such as powerful communication [6] or direct translation into actual implementation [7], [8]. The Language of Temporal Ordering Specification (LOTOS) [9] is one of the most expressive process algebraic notations with rich repository for process data encapsulation and process communication functionality. Consequently, LOTOS has been previously used as rigorous foundation for workflow systems [10], [11]. However, no systematic and complete investigation of LOTOS in the context of real-world workflow has been conducted yet.

The first contribution of this paper is the complete translation of comprehensive workflow functionality to LOTOS. We use a control-flow pattern repository [12] to systematically formalize comprehensive real-world workflow requirements and equip them with unambiguous execution semantics. The second contribution of this paper is a mature suitability and expressiveness analysis of LOTOS based on workflow patterns.

The remainder of this paper is organized as follows. Sect. II provides a brief introduction to LOTOS. In Sect. III we introduce the complete mapping of the control-flow pattern set. Sect. IV evaluates our expressiveness and suitability results. Sect. V discusses related work followed by the conclusion in Sect. VI.

II. LOTOS

Process algebras extend classical automate theory and are well suited for description of reactive systems; e.g., operating, automation, or communication systems. With a special focus on concurrency and their compositional property, these formal notations allow building modular and hierarchical system descriptions in an intuitive fashion. LOTOS is a highly expressive formal description standard (ISO 8807) for design of distributed systems tailored towards OSI services and protocols. The key idea is to define temporal relations between system events instead of using system state representation. LOTOS is formed by two orthogonal components, which are mainly based on three algebraic theories: the process algebras CCS [13] and CSP [14] for system dynamics and the abstract data type language ACT ONE [15] for data structures and value expressions. The LOTOS formalism has been widely applied in large communication system specification. Soundness, formal semantics, different abstraction levels, and rich description repository for concurrency and communication are some of its characteristic properties.

A LOTOS process performs unobservable *internal actions* and also has the ability of interacting with other processes. These interactions are made in a synchronous way through atomic units of synchronization and are called *observable actions*. Interactions may involve the exchange of data and use *gates* as points of interaction. As convention, a synchronization and its corresponding observable action are named identically. If a process is ready for interaction through a particular synchronization a_i, it is said to *offer* the observable action a_i to the environment. Apart from observable actions for synchronization and communication between processes, unobservable internal actions represent atomic internal system decisions, which cannot be influenced by the environment.

TABLE I
BASIC LOTOS OPERATOR SUMMARY.

Operator	Name
;	action prefixing
[ ]	choice between behaviors
|[. . .]|	parallel composition in general
||	full synchronization parallel composition
|||	pure interleaving parallel composition
>>	sequential composition
[>	disabling behavior
hide	hiding actions in behaviors
stop	inaction indicator
exit	successful termination indicator

A. Basic LOTOS

The high expressiveness of LOTOS comes at the cost of increased complexity, which may result in understanding difficulties. In an effort to address this issue and to emphasize the importance of the algebraic operators, the simplified notation *Basic LOTOS* was created. This subset of *Full LOTOS* facilitates understanding and is beneficial in the development of process equivalence theory. Basic LOTOS focuses on describing the temporal ordering of actions and there is no data exchange between processes during synchronization. Actions are identified by the names of the gates that offer them to the environment. These gate names are sufficient for establishing process synchronizations and the algebraic expressions that reflect process behavior become the central part of the process definition. The latter, referred to as *behavior expressions*, are built in a modular fashion from actions, LOTOS operators, and other behavior expressions. For this paper it is important to understand the Basic LOTOS operators in Tab. I. Please refer to [9] for a comprehensive introduction to Basic LOTOS and a detailed description of operator syntax and semantics.

B. Full LOTOS

Full LOTOS (or LOTOS for short) contains additional structures for data representation and exchange. In Full LOTOS, the idea of identifying synchronizations through identically named actions remains the same, but it is now necessary to specify data to be exchanged during synchronization. Therefore, actions need to be represented by the name of a gate with an additional list of values and variables associated with this gate. For example, $sinc\langle 5\rangle$ and $p\langle$*'teste'*$, 17\rangle$ are valid actions offering their values at the specified gates. Synchronization is possible when actions have the same name.

The structure used to prefix behavior expressions is called *action denotation*. Bolognesi et al. [9] describe it as a structure composed by a gate name *p*, followed by a finite list of n attributes (i.e., value or variable declarations), in the form $p\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n$.

A *value declaration* uses the notation $!E$ where E is an expression that describes a value. Therefore, $!5$ and $!(x+y+4)$ constitute valid declarations. By placing an attribute value declaration α_i in the list belonging to gate p, the value α_i will be offered through gate p. If the expression contains variables such as $!(x+4)$, they will be replaced by values according to their context. Therefore, in the structure *sinc* $!(6-2)$, the gate *sinc* offers expression $(6-2)$, which describes $sinc\langle 4\rangle$.

A *variable declaration* uses the notation *?x:t* where x is the name of a variable, and t is the name of a data set that can be attributed to x. Therefore, *?y:char* and *?x:nat* are valid declarations. In a synchronization every value contained in the specified data set can be accepted into the variable. Setting the variable x with value v, any occurrence of variable x in the behavior expressions following the synchronization will automatically be replaced by the value v. Following the notation in [9] we can generalize: $p\ ?x{:}t; B(x) \xrightarrow{p\langle v\rangle} B(v)$.

Furthermore, it is possible to associate data facilities as conditions to behavior expressions imposing some rules on their execution. Variable declarations can also be used to define parametric processes, such that these variables can be used inside behavior expressions. Due to the rigid space constraints of this paper it is out of scope to provide a complete introduction to LOTOS. However, the knowledge of LOTOS is crucial in the understanding of Sect. III. Please refer to [9] for a comprehensive discussion.

III. MAPPING CONTROL-FLOW WITH LOTOS

In this section, we map the complete control-flow patterns repository [12] to LOTOS. This methodology has been previously applied for systematic analysis of workflow technology and specification in different contexts (e.g., BPEL [16], BPM with process algebra [7], and π-calculus [17]) and has evolved into the Workflow Patterns Initiative [18]. For each pattern, we provide a brief description, the formal algebraic expression for an example, and a short discussion. The pattern is *represented* by the algebra, if it provides the functionality to build an expression, equivalent to the graphical pattern representation.

An elementary activity such as a business step can be regarded as atomic unit of work. However, the concept of an atomic action has no explicit equivalent in LOTOS since actions represent synchronizations. Instead, we will use a basic activity process to represent this concept (see Fig. 1).

---

process $K[k_1,\ k_2,\ k_c]$
(k_1; k_2; **exit**) [> (k_c; **stop**)
endproc

Fig. 1. Basic activity

Say, the basic activity process K coordinates the execution of an external process S, responsible for carrying out the work of the activity. S could constitute an interface between LOTOS and a web service, for instance. The action k_1 invokes the elementary activity since K synchronizes with S through k_1. Once invoked S executes the internal work of the activity. After execution the process S synchronizes with K using the action k_2. The successful termination is flagged by ***exit***. During the activity's execution, it is possible to flag its failure

or invalidation with the *disabling* operator. The cancellation is reported through synchronization with the action k_c, which interrupts the main flow and uses ***stop*** instead of ***exit***.

In addition to the concept of elementary activity, the term activity is also used in the pattern context. This results in a two-fold interpretation. An activity is a basic activity as well as a set of activities with an execution order. The term activity will be used to represent elementary activities as described above and processes whose behavior expressions represent an ordered execution of several elementary activities. For facilitated understanding we employ a simplified notation. To instantiate a process in a behavior expression, we use the process name omitting the gates. To define a process, the structure "process name = behavior expression" is used, since it already contains all necessary information.

A. Basic Control-Flow Patterns

Sequence (Pattern 1): Activity B will only be enabled to execute after successful completion of activity A, which constitutes a sequence of these activities. See Fig. 2 (a).

$$P = A >> B$$

In this representation it is important to notice that we use the sequential composition operator (">>") instead of the action prefix (";") because processes may be used as activities.

Parallel Split (Pattern 2): The end of the execution of activity A, which represents a single thread of control, splits into multiple threads enabling the parallel execution of activities B and C. See Fig. 2 (b).

$$P = A >> (B \;|||\; C)$$

The use of the interleaving operator ("|||") reflects the concurrent, independent execution of activities. The sequential composition operator guarantees that the split will only occur after A has terminated its execution.

Synchronization (Pattern 3): Multiple parallel activities represented by activities A and B converge into a single thread of control represented by activity C. C will only be enabled after both A and B complete their executions and synchronize. It is assumed that each incoming branch of the synchronizer is executed only once. See Fig. 2 (c).

$$P = (A \;|||\; B) >> C$$

The synchronization between parallel branches is guaranteed by the interleaving operator imposing synchronization of the successful terminations of A and B. The execution of C can only proceed after this synchronization occurs.

Exclusive Choice (Pattern 4): After the execution of activity A, one of several branches must be chosen (i.e., activity B or C). See Fig. 2 (d).

$$P = A >> (([\mathit{cond1}] \rightarrow B) \;[\,]\; ([\mathit{cond2}] \rightarrow C))$$

The choice operator ("[ ]") allows the representation of this pattern. Mutually exclusive conditions $[\mathit{cond1}]$ and $[\mathit{cond2}]$ relate to guarded expressions and reflects the runtime decision that has to be made based on some control-flow data. This representation illustrates a clear advantage of using LOTOS over simpler process algebras due to its capability of representing encapsulated data.

Simple Merge (Pattern 5): Although, both activities A and B are initially enabled, it is assumed that only one of the alternative branches will execute. After the chosen activity finishes execution, activity C is enabled. See Fig. 2 (e).

$$P = (A \;[\,]\; B) >> C$$

or

$$P = (([\mathit{cond1}] \rightarrow A) \;[\,]\; ([\mathit{cond2}] \rightarrow B)) >> C$$

Considering the existing patterns of choice, the Simple Merge can only occur after an Exclusive or a Deferred Choice. The first algebraic expression indicates a merge situation after a Deferred Choice. The second defines a merge after an Exclusive Choice. Despite the difference, the merged itself remains the same and is represented by a sequential composition with C.

B. Advanced Branching and Synchronization Patterns

Multi-Choice (Pattern 6): After executing activity A, a number of branches are chosen based on decision or workflow control data. Therefore, one of the following three options will be enabled: B, C, or both in parallel. Differently than in the Exclusive Choice, it is possible to execute more than one branch in parallel. See Fig. 2 (f).

$$P = A >> (([\mathit{cond1}] \rightarrow B \;[\,]\; [\neg \mathit{cond1}] \rightarrow \textbf{\textit{exit}}) \;|||\; ([\mathit{cond2}] \rightarrow C \;[\,]\; [\neg \mathit{cond2}] \rightarrow \textbf{\textit{exit}}))$$

We employ the interleaving operator to represent the possibility of parallel execution of all branches. However, it is necessary to evaluate for each branch whether it will actually be executed or not. This is done by the structure $([\mathit{cond}] \rightarrow X \;[\,]\; [\neg \mathit{cond}] \rightarrow \textbf{\textit{exit}})$. Depending on the condition *cond*, this structure handles the execution decision of activity X. If the condition evaluates to 'true', the activity will be executed. Otherwise ($\neg$*cond* evaluates to true), and ***exit*** guarantees synchronization with other branches at the end of the parallel execution. In order to continue with the execution flow, all branches have to synchronize.

Synchronizing Merge (Pattern 7): After executing activity A in Multi-Choice, the Synchronizing Merge has to wait for the termination of all activated paths. If more than one path are activated (i.e., activities B and C), they have to be synchronized into a single thread before the next activity (D) can be executed. In case only one of them has been activated, D will be enabled after the chosen activity has terminated execution. It is assumed that each branch can only execute once. See Fig. 2 (g).

$$P = A >> (([\mathit{cond1}] \rightarrow B \;[\,]\; [\neg \mathit{cond1}] \rightarrow \textbf{\textit{exit}}) \;|||\; ([\mathit{cond2}] \rightarrow C \;[\,]\; [\neg \mathit{cond2}] \rightarrow \textbf{\textit{exit}})) >> D$$

This pattern is easily representable, due to the behavior of the Multi-Choice structure, which already enforces the required synchronization. To represent, that D follows the synchronization, we use the sequential composition operator to merge the threads.

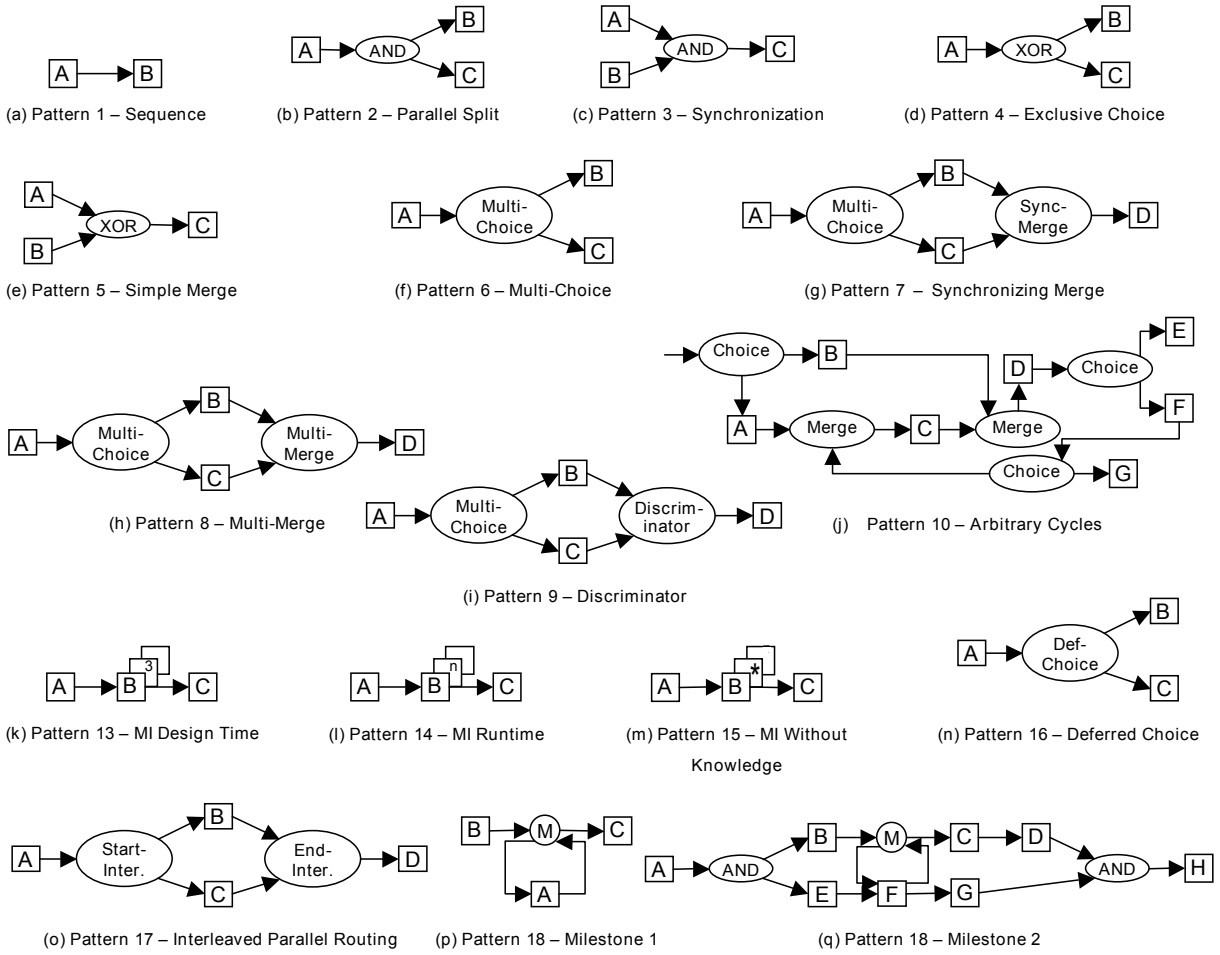

Fig. 2. Control-flow patterns (adapted from [18]).

Multi-Merge (Pattern 8): After executing activity A in Multi-Choice, more than one branch might get activated (i.e., activity B and C). In this case activity D, which follows the merge, is invoked each time one of the parallel branches finishes its execution, and there is no synchronization between the parallel branches. In case only one branch is initially activated (i.e., B or C), D will be invoked only once after the chosen branch has terminated its execution. See Fig. 2 (h).

$$P = A >> (([cond1] \rightarrow (B >> D)\ [\,]\ [\neg cond1] \rightarrow \textbf{\textit{exit}}) \\ |||\ ([cond2] \rightarrow (C >> D)\ [\,]\ [\neg cond2] \rightarrow \textbf{\textit{exit}}))$$

The structure used to represent this pattern is very similar to the Synchronizing Merge. The only difference concerns the location of D, which is located inside of each parallel branch of the Multi-Choice in this pattern. This ensures that D is executed every time a branch is activated. Note that D can be interpreted as subprocess, which makes this pattern a central means of composition in more complex structures.

Discriminator (Pattern 9): After executing activity A, in Multi-Choice, the discriminator waits for one of the incoming branches (i.e., activity B or C) to complete execution before activating activity D. Subsequently, each terminating branch is ignored until all activated branches have terminated execution. This resets the pattern, so that it can be triggered again. Note that unlike the Multi-Merge D is not enabled again once a potential subsequent branch finishes execution. See Fig. 2 (i).

$$P = \textbf{\textit{hide}}\ q\ \textbf{\textit{in}}\ P_1 \\ P_1 = A >> (([cond1] \rightarrow (B >> q)\ [\,]\ [\neg cond1] \rightarrow \textbf{\textit{exit}}) \\ |||\ ([cond2] \rightarrow (C >> q) \\ [\,]\ [\neg cond2] \rightarrow \textbf{\textit{exit}}))|[q]|\ (q; (D\ |||\ P_2)) \\ P_2 = (q; P_2)\ [\,]\ \textbf{\textit{exit}}$$

To express this pattern, it is necessary to use an additional gate q and the additional process P_2. q is responsible for invoking the subprocess $(q; (D\ |||\ P_2))$ that enables activity D. P_2, which is executed in parallel recursively, waits for other potentially enabled activities and ignores their termination. Once no more active branches remain, the P_2 exit operator is used to terminate the execution of P_2. The gate q remains hidden inside the discriminator avoiding the need of external synchronizations.

C. Structural Patterns

Arbitrary Cycles (Pattern 10):

A loop in a process allows activities to be performed repeatedly. After executing activity F, a cyclical repetition of

activities can occur depending on which branches are selected during process execution. See Fig. 2 (j).

$$
\begin{aligned}
P_1 &= ([\textit{cond1}] \rightarrow (A >> P_2)) \\
&\quad [\,] ([\neg \textit{cond1}] \rightarrow (B >> P_3)) \\
P_2 &= C >> P_3 \\
P_3 &= D >> (([\textit{cond2}] \rightarrow E) \\
&\quad [\,] ([\neg \textit{cond2}] \rightarrow (F >> ([\textit{cond3}] \rightarrow G \\
&\quad [\,] [\neg \textit{cond3}] \rightarrow P_2)))) \\
&\text{or} \\
P_1 &= (A >> P_2) \; [\,] \; (B >> P_3) \\
P_2 &= C >> P_3 \\
P_3 &= D >> (E \; [\,] \; (F >> (G \; [\,] \; P_2)))
\end{aligned}
$$

To represent the cyclical repetition of loops, it is necessary to use recursive processes. The presence of choices are required as well, in order to allow the choice between continuing the repetition or leaving the loop structure. Both Exclusive Choice and Deferred Choice (see Sect. III-E) can be used for this purpose. Therefore, both cases are represented above. The main difference is the presence of conditions (e.g., *cond1*) when using the Exclusive Choice and the absence of such when using the Deferred Choice.

Implicit Termination (Pattern 11): A given subprocess should be terminated when there is nothing else to be done. This means, that no more activities are classified as active in the workflow and no other activities can be invoked and activated. In LOTOS the termination idea is represented through the keyword ***stop***, which indicates inaction and deadlock situations. Even after successful termination symbolized by the exit operator ***exit***, a ***stop*** is generated to symbolize that the process is inactive at that moment.

D. Patterns Involving Multiple Instances

Multiple Instances Without Synchronization (Pattern 12): In the case of single process instances, multiple instances of an activity (A) can be created through new independent threads. There is no need for synchronization, and so other activities or subprocesses (e.g., $B >> C$) are allowed to execute in parallel with the threads of activity A.

$$
\begin{aligned}
P &= P_1 \; ||| \; (B >> C) \\
P_1 &= (P_1 \; ||| \; A) \; [\,] \; A
\end{aligned}
$$

The role of subprocess P_1 is to represent all possible instances of activity A. Being recursive, P_1 can spawn new parallel threads. The last parallel thread has to be created using the right second option of the P_1 choice operator, which stops the recursion. The process P is responsible for representing the parallel execution of the threads of A and the subprocess $(B >> C)$.

Multiple Instances with a Priori Design Time Knowledge (Pattern 13): When activity A terminates its execution, activity B is enabled. However, in this pattern an activity can be enabled multiple times. The number of instances of such an activity is known at design time (e.g., activity B will be executed three times). Once every instance execution is completed, a subsequent activity (C) needs to be started. See Fig. 2 (k).

$$
\begin{aligned}
P &= A >> P_1 >> C \\
P_1 &= B \; ||| \; B \; ||| \; B
\end{aligned}
$$

This pattern is easily represented using the interleaving operator. Process P_1 represents the parallel execution of all three instances of B. This process will solely be enabled after the end of the execution of A. After the execution of all instances of B, C will be enabled to execute.

Multiple Instances with a Priori Runtime Knowledge (Pattern 14): When activity A finishes its execution, activity B is enabled. B can be executed several times, and the number of instances will be determined at runtime, but before the instances of this activity are created. This number may potentially vary for each case and may depend on case characteristics or availability of resources. Once all B instance executions are completed, another activity (C) is enabled. See Fig. 2 (l).

$$
\begin{aligned}
P &= A >> P_1(n) >> C \\
P_1(x) &= ([x <= 0] \rightarrow \textbf{\textit{exit}}) \\
&\quad [\,] \; ([x > 0] \rightarrow (B \; ||| \; P_1(x-1)))
\end{aligned}
$$

This patterns emphasizes the importance of encapsulated data in an algebraic control-flow structure. The number of instances (n) is simply passed as parameter of process P_1. Instances are created through recursion in this process. When all required instances are created, the base of the recursion is reached, which stops the recursive calls. Process P guarantees the ordering between A, all instances of B, and C.

Multiple Instances without a Priori Runtime Knowledge (Pattern 15): After activity A terminates its execution, activity B is enabled. The number of instances of B is not known before the instances are created. The number becomes known during the activities' execution. After the execution of all created instances, C is enabled. The difference to the previous pattern is that new instances of B can be created while others are being executed or have already completed execution. See Fig. 2 (m).

$$
\begin{aligned}
P &= A >> P_1 >> C \\
P_1 &= (P_1 \; ||| \; B) \; [\,] \; B
\end{aligned}
$$

To represent this pattern, two processes are required. P_1 is a recursive process responsible for the parallel execution of instances of B . Note that at least one instance of B must be executed before executing C. Multiple execution of B is allowed in this structure until C terminates its execution.

E. State-based patterns

Deferred Choice (Pattern 16): After ending the execution of activity A, one of several branches (i.e., B or C) have to be chosen. In contrast to an Exclusive Choice, this choice is not made explicitly. Several alternatives are offered to the environment, which will choose one of them. The choice is delayed until the processing in one of the alternative branches is actually started.

$$
P = A >> (B \; [\,] \; C)
$$

This pattern is represented using the choice operator, and its structure is similar to the Exclusive Choice. The difference is that there is no need to use conditions due to the decision being made by the environment.

Interleaved Parallel Routing (Pattern 17): The execution of activity A enables a set of activities (i.e., B and C), which can be executed in an arbitrary order. Each activity in this set is executed, and the ordering is decided at runtime. Activities are not allowed to execute in parallel. After the execution of B followed by C or C followed by B, activity D is enabled for execution. See Fig. 2 (o).

$$
\begin{aligned}
P &= A >> P_1 >> D \\
P_1 &= \textbf{\textit{hide}}\ q, r\ \textbf{\textit{in}}\ ((((q; B) >> (r; \textbf{\textit{exit}})) \\
&\quad ||| ((q; C) >> (r; \textbf{\textit{exit}}))) \ |[q, r]|\ P_2) \\
P_2 &= q; r; P_2\ [\ ]\ \textbf{\textit{exit}}
\end{aligned}
$$

This pattern can be represented using the three processes shown above. Process P is responsible for assuring the sequential ordering between A, process P_1 and D. P_1 handles the execution of the arbitrarily ordered activities. In P_1, each activity is embedded in the structure $((q; \text{Activity}) >> (r; \textbf{\textit{exit}}))$, preceded by a gate q, and succeeded by a gate r. These two gates ensure the exclusive execution of the inner activity at any time. The structures are joined by the interleaving operator, which guarantees that all activities are terminated before executing D. Process P_2 is used as "resource container" to control the interleaved activity execution. Every P_1 activity structure must synchronize with P_2 at q and r. When q in P_2 is used in a synchronization with a P_1 activity (e.g., B), the other activity structures are blocked on q. When B is terminated, its structure is synchronized with P_2 at r flagging the end its execution. This synchronization allows the recursive call of P_2 offering q for the next activity to be executed.

Milestone (Pattern 18): This pattern models the execution of an activity depended on the state of a process instance. The enabling is determined by the state after the end of execution of an activity and the beginning of the next activity. To illustrate this pattern, we show two examples. Firstly, activity A will only be enabled for execution after activity B has been executed, but before activity C has been invoked. This state is called a place M and considered a milestone for A. See Fig. 2 (p).

$$
\begin{aligned}
P &= B >> (\textbf{\textit{hide}}\ p\ \textbf{\textit{in}}\ (P_1\ |[p]|\ P_2)) \\
P_1 &= p; P_1\ [\ ]\ C \\
P_2 &= (p; A >> P_2)\ [\ ]\ \textbf{\textit{exit}}
\end{aligned}
$$

Secondly, we show a different usage example for this pattern. The difference to the first example is the existence of parallel execution threads. One of these threads contains the place M. Different threads contain the milestone-dependent activity F, which can only be executed after the end of activity E. Note that unlike the previous example, if the milestone-dependent activity is not executed, the process goes into deadlock, because activities F, G, and H will never be executed. See Fig. 2 (q).

$$
\begin{aligned}
P &= A >> (\textbf{\textit{hide}}\ p\ \textbf{\textit{in}}\ (P_1\ |[p]|\ P_2)) >> H \\
P_1 &= B >> P_3 \\
P_2 &= E >> ((P_4 >> G)\ [\ ]\ \textbf{\textit{stop}}) \\
P_3 &= (p; P_3)\ [\ ]\ (C >> D) \\
P_4 &= p; F >> (P_4\ [\ ]\ \textbf{\textit{exit}})
\end{aligned}
$$

Both examples are modeled by the principle of using a gate to simulate a milestone. In the first example, we use three processes to represent this pattern. Note that, once C has been chosen, p cannot be offered anymore. P_2 is responsible for "consuming" the milestone p offered by P_1 to execute A. The activity can be executed several times while C is not chosen. Despite of using more processes than the first example, the representation idea remains the same in the second example. A *stop* is used inside of process P_2 to represent the deadlock possibility.

F. Cancellation Patterns

Cancel Activity (Pattern 19): After executing activity A, activity B is enabled with a disabling option before its execution. If this happens, the thread waits for the execution of the activity and activity C will never be executed. If the cancellation is not used, activity C becomes enabled after execution of B.

$$P = A >> B >> C$$

Instead of representing this pattern explicitly, we need to remember that $B = (k_1; k_2; \textbf{\textit{exit}})[> (k_c; \textbf{\textit{stop}})$. If B is cancelled, the ***stop*** guarantees the representation of an inaction state, which allows removing the thread. Since the sequential composition operator depends on a successful termination to proceed, the use of ***stop*** will also ensure that C will never be executed.

Cancel Case (Pattern 20): This pattern models the complete removal of a process instance, which is related to the workflow execution service and not process definition. Therefore, a processes can be simply canceled by the execution controller.

IV. Evaluation

Based on Sect. III, we can see that all proposed patterns can be represented using LOTOS. The results clearly indicate that LOTOS is very suitable for control-flow pattern representation. Unlike simpler algebraic representations, LOTOS represents the set exhaustively. Under the assumption that the set of 20 patterns identifies comprehensive workflow functionality of real-world projects [12], we can conclude that the expressive power of LOTOS suffices to model any required functionality in real-world workflow. When compared to other algebraic notations, LOTOS shows some important advantages, such as the capability of modeling data types, variables and expressions, as well as the presence of indicators of process inactivity. Apart from expressiveness and suitability, it is important to remember that LOTOS expressions, in some patterns representation, are very short compared to simpler process algebras. For instance, ACP [19] generates very long expressions for certain

patterns due to necessary case-enumeration. While this does not influence theoretic representativeness, it may constitute a suitability problem in practice.

V. Related Work

In [20] there was a first attempt to use LOTOS to describe some workflow patterns. The aim of the authors, however, was to provide a real workflow case example described in LOTOS. To accomplish this task, they have mapped to LOTOS a restricted set of patterns (only three of them), showing the importance of the subject. However, they left uncovered the representativeness and mapping of all control-flow patterns. Our work comes to complete all the remaining mappings and to discuss the expressiveness of LOTOS based on each pattern. Formal definition of abstract syntax and operational semantics for workflow patterns as a conceptual foundation for process-aware information systems has also been previously addressed using Petri Nets [21] and π-calculus [22]. The workflow language YAWL [23] is based on Petri Nets and was developed by Aalst et al. after their initial work on workflow patterns [12]. Their pattern repository enables comparison and evaluation of workflow specifications and has evolved in the Workflow Patterns Initiative, currently spanning the areas of control-flow, data and resource patterns [18]. The most recent formalization of this functionality using Petri nets was introduced by Russell as *new*YAWL [24]. In contrast to these graphical approaches Puhlmann et al. [17] employ π-calculus algebra to introduce complete execution semantics in a formal mapping of the initial pattern repository. Despite the close relationship between these efforts and our work, each of them has a distinct contribution and a parallel background. The native features of graphical representation, mobility and abstract data type handling are uniquely attributed to Petri nets, π-calculus, and LOTOS, respectively. It is necessary to establish a common basis, before a sophisticated formalism characterization can be undertaken on the grounds of actual implementation. Our work closes this initial gap and answers the question whether LOTOS is a suitable notation for the workflow domain.

VI. Conclusion

This paper presents formal semantics for comprehensive workflow functionality through precise mapping to algebraic notation in LOTOS to support collaborative applications. Systematic analysis of suitability requirements for real-world workflows was achieved through a workflow pattern approach. This work successfully closes the gap to formalisms, such as π-calculus and Petri nets, providing a foundation for application in the workflow domain.

By showing that LOTOS satisfies all examined behavioral workflow requirements, this work opens the door to taking advantage of the powerful LOTOS data model in collaborative applications. Since the design-features of LOTOS include rigorous data encapsulation in concurrent system specification, it might constitute an innovative way of dealing with challenges in the workflow domain. As a next step this theoretic contribution can be utilized towards rigorous and reliable implementation at the core of collaborative application architectures. Future research topics include automated translation of graphical workflow notation to LOTOS and collaborative applications based on formal workflow structures.

Acknowledgement

This work has been supported by grant# 06/00375-0, from FAPESP (São Paulo State Research Foundation). Additional support is provided by grant# 482139/ 2007-2 from CNPq (Brazilian National Research Council).

References

[1] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[2] S. White, "Business process modeling notation (BPMN)," Business Process Management Initiative (BPMI)–Version 1.0–BPMI.org, 2004.

[3] D. Jordan and J. Evdemon, "Web services business process execution language version 2.0." Public Review Draft OASIS WS-BPEL Technical Committee, April 2007.

[4] W. M. P. van der Aalst, "Pi calculus versus petri nets: let us eat humble pie rather than further inflate the pi hype," Unpublished Discussion Paper, 2003, http://tmitwww.tm.tue.nl/staff/wvdaalst/pi-hype.pdf.

[5] M. H. ter Beek, A. Bucchiarone, and S. Gnesi, "Web service composition approaches: From industrial standards to formal methods," in *ICIW*, 2007, p. 15.

[6] F. Puhlmann, "Why do we actually need the π-calculus for business process management?" in *BIS*, 2006, pp. 77–89.

[7] K. R. Braghetto, J. E. Ferreira, and C. Pu, "Using control-flow patterns for specifying business processes in cooperative environments," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 1234–1241.

[8] ——, "Business processes management using process algebra and relational database model," in *ICE-B*, 2008.

[9] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Comput. Netw. ISDN Syst.*, vol. 14, no. 1, pp. 25–59, 1987.

[10] A. Ferrara, "Web services: a process algebra approach," in *ICSOC*, 2004, pp. 242–251.

[11] K. J. Turner, "Representing and analysing composed web services using CRESS," *J. Network and Computer Applications*, vol. 30, no. 2, pp. 541–562, 2007.

[12] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[13] R. Milner, *Communication and concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[14] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

[15] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1985.

[16] W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Web service composition languages: Old wine in new bottles?" in *EUROMICRO*, 2003, pp. 298–307.

[17] F. Puhlmann and M. Weske, "Using the π-calculus for formalizing workflow patterns," *Business Process Management*, pp. 153–168, 2005.

[18] W. van der Aalst and A. ter Hofstede, "Workflow patterns website," June 2008. [Online]. Available: http://www.workflowpatterns.com

[19] W. Fokkink, *Introduction to Process Algebra*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.

[20] V. Carchiolo, A. Longheu, and M. Malgeri, "Using LOTOS in workflow specification," in *ICEIS (3)*, 2003, pp. 364–369.

[21] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Technische Hochschule Darmstadt, Darmstadt, Germany, 1962.

[22] R. Milner, *Communicating and mobile systems: the π-calculus*. New York, NY, USA: Cambridge University Press, 1999.

[23] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: yet another workflow language," *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005.

[24] N. C. Russell, "Foundations of process-aware information systems," Ph.D. dissertation, Queensland University of Technology, Brisbane, Australia, December 2007.