A Timed Semantics of Workflows

Marcello M. Bersani[1] Salvatore Distefano[1,2] Luca Ferrucci[3] Manuel Mazzara[4]

[1] Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Italy
{marcellomaria.bersani,salvatore.distefano}@polimi.it
[2] Kazan Federal University, Kazan, Russia
[3] ISTI-CNR, Italy
luca.ferrucci@isti.cnr.it
[4] Innopolis University, Kazan, Russia
m.mazzara@innopolis.edu.ru

Abstract. We formalize timed workflow with abnormal behavior management (i.e. recovery) and demonstrate how temporal logics and model checking are methodologies to iteratively revise the design correct-by construction system. We define a formal semantics by compiling generic workflow patterns into an extension of LTL with dense time clocks (CLTLoc). CLTLoc allows us to define the first logical formalization of workflows that can be practically employed in verification tools and to avoid the use of well-known automata based formalisms dealing with real-time. We use an ad-hoc bound model checker to prove requirements validity on a business process. The working assumption is that lightweight approaches easily fit into processes that are already in place so that radical change of procedures, tools and people's attitudes are not needed. The complexity of formalisms and invasiveness of methods have been demonstrated to be one of the major drawback and obstacle for deployment of formal engineering techniques into mundane projects.

Keywords: Workflow, Recovery Framework, Formal Methods, Temporal Logic, Semantics

1 Introduction

Workflows are logical, abstract artifacts able to describe or implement patterns of activities required to perform real works. Any type of resources is considered in this definition, both human and mechanical, including electronic, automation and computer-based devices or systems. The general concept of workflow has been adopted and characterized in several contexts such as economics (knowledge economy, business process), management (manufacturing, job shop), mathematics (queueing systems, decision making, operations research), engineering (quality engineering, service engineering). With specific regard to computer science, workflows are used in several contexts such as algorithms, business processes, Web services, machine learning, service oriented computing, software product lines, human machine interaction.

Workflows are therefore applied in different contexts and also in different ways: in design phase, where their representations are used to evaluate specific properties

(both functional and non-functional) of the corresponding systems; to represent and evaluate the effectiveness of scheduling systems and policies; in document management and imaging; in service composition and orchestration (service oriented architecture/infrastructure, bioinformatics and cheminformatics scientific workflows); to study human machine interactions.

This work can be framed into the workflow representation techniques, proposing a formal approach based on CLTLoc[10] semantics for modelling the real-time workflow executions and for the verification of dependability properties. In fact, as CLTLoc extends LTL [26] by allowing timing constraints into formulae, both instantaneous (zero time) and time consuming activities can be taken into account in the workflow model, thus providing an effective tool for dependability verification of workflows.

On methods and tools Logics and model-checking have been successfully used in the last decades for modelling and verification of various types of hardware and software systems and have a stronger credibility in the scientific community when compared with other formalisms. We here give an CLTLoc-based semantics of workflow execution and use $\mathbb{Z}$ot [37] model checker for requirement verification. By applying model-checking on the case study presented in this paper, we demonstrate the feasibility of the verification approach and how temporal logics can work for both modelling and verification of (simple but realistic) business workflows inclusive of exception handling. This work has to be intended as complementary to what has been done in [27] where a similar problem was approached in term of Process Algebra.

Differently from several others formalizations only offering languages without methods – for a detailed discussion see [28] and [29] – our approach, together with software tools, aims at offering a complete practical toolkit for software and systems engineers working in the field of workflow design. Following the line of [24] this approach goes under the correct-by-construction paradigm and the idea of developing dependable systems by integrating specific approaches well-suited to each development phase.

The ideal process of workflow verification is an iterative process. In this work, we aim at providing an instrument for workflow revision, i.e. a procedure to follow until the requirements are finally met. To do this, we encode the workflow into a formal language and, at the same time, we formally describe specific requirements on the system. This is discussed in Section 3. At this point, as shown in Section 4, correctness can be automatically determined via the $\mathbb{Z}$ot model checker. As an outcome of model checking we may need to revise the workflow in order to meet the requirements.

A descriptive semantics The role of temporal logics in verification and validation is two-fold. First, temporal logic allows abstract, concise and convenient expression of required properties of a system. In fact, temporal logic is often used with this goal in the verification of finite-state models, e.g., in model checking [6]. Second, temporal logic can be used as a descriptive approach for specifying and modelling systems (see, e.g., [33,20]). A descriptive model is based on axioms, written in some (temporal) logic, which define the system through its general properties, rather than by an operational model based on some kind of machine behaving in the desired way. In this case,

verification typically consists of satisfiability checking of the entailment between the model and the desired property [38].

Specifying temporal relations among events that do not inherently behave in an operational way may become rather hard when operational models are employed. This is the case for the system recovery considered here. Exception handling is an event-based paradigm that implements the asynchronous exchange of warning events among actors that are part of the system. The typical implementation of exception handling mechanisms – through logical rules of the form *if* $(cond)$ *then* throw(e) and *try-catch* blocks – requires ad-hoc extensions of operational-based formalisms by means of the definition of message-passing primitives. Specifying exception handling mechanisms through temporal logic can be easily achieved by the logic itself and also allows modelling of classes of exceptions endowed with a specific semantics (see Section 3) in a coherent and uniform way.

Other approaches and novelty Several approaches have been adopted in recent years to provide formal semantics of business processes. Most of them are very much bound to a specific formalism accordingly extended to better cope with modelling issues. These attempts mostly belong (but not limited) to the process algebras, Petri nets or model-based philosophies, with some raid into temporal logics et similia too.

Mobile process algebra have been successfully used in [27] that this work intend to complement. Limitations of process algebras approaches like the previous ones and, for example, [41] are related to the fact that process algebras are based on equational reasoning. From a practical perspective, this makes verification tricky, difficult and certainly not user-friendly, because verification is mainly carried out by specific proof techniques that are used to prove behavioural equivalence among processes. Furthermore, all these approaches mostly focus on the verification of reachability-based properties (with some exceptions like [13]) and tool support is very limited (see [30]). On the other side, other works like [32] provide a methodology and tool support for the modelling phase, but do not cope with the verification phase and either do not belong to the correct-by-construction paradigm.

Petri Nets supporters and van der Aalst approaches like Workflow Petri Nets (WPN) [1] reached the objective of verification and tool support to a much larger extent than other communities. This approach is based on extensions of previously existing formalisms and still represents an operational model, which also inherits the relative overhead. A successful attempt to overcome this issue has been provided by [42] where acyclic WPN are translated into a finite-state automaton and verified against a suitable LTL property in order to verify soundness.

Model-based approaches have also been used, though to a much lesser extent and often in combination with testing, for validation of business critical systems. The B-model is one of the most popular together with its reactive-systems extension Event-B [5]. B and Event-B are not lightweight methods. They do come with a refinement-based methodology, which however cannot easily be embedded into already existing industrial processes [24].

In the domain of temporal logics, CTL has been used to specify and enforce inter-task dependencies [4], and LTL for UML activity graphs verification [19]. Other tem-

poral logics have also been used for similar objectives. In particular, in [8] a complete and coherent semantics based on the TRIO logic [23] has been proposed for a more consistent set of UML diagrams.

Recovery frameworks have been more rarely formalized in similar manners instead. This was the main contribution of [21]. One of the first works formally discussing business recovery in terms of long-running transactions is [12]. In [16] a simplified and clarified semantics of WS-BPEL recovery framework has been presented in terms of Process Algebras. In [17] the state-of-the-art in formalizing fault, compensation and termination mechanisms of WS-BPEL 2.0 has been deeply investigated. More recently, another model has been formulated for the description of composite web services orchestrated by WS-BPEL and with resources associated. The key contribution of [15] is the integration of WS-BPEL with WSRF [22], a resource management language, taking into account the main structural elements of WS-BPEL with event handling and fault handling.

The main contribution of the paper is a timed semantics for workflows which extends the one in [21] with the definition of timing constraints bounding delays of activities and transitions. CLTLoc, being the first extension of LTL with constraints over Reals, allows us to define a descriptive semantics based on a temporal logic, along the line of that we proposed in [21], without resorting to Timed Automata[3], the de-facto standard formalism for dealing with real-time reasoning. So, to the best of the authors' knowledge, the proposed semantics is the first descriptive formalization of workflows with recovery over real-time.

The working assumption is that a lightweight solution would easily fit into processes that are already in place without the need for a radical change of procedures, tools and people's attitudes, which is actually the case for most of the aforementioned techniques. The complexity of formalisms and invasiveness of methods have been demonstrated to be one of the major drawback and obstacle for deployment of formal engineering techniques into mundane projects [24], [40].

The rest of the paper is organized as follows: Section 2 describes the case study of a workflow for order processing. The semantics of workflows and exception handling is given using temporal logic in Section 3 where a general encoding into CLTLoc is provided. In Section 4 the implementation of this translation is illustrated and tests have been carried out to validate its correctness. Finally, Section 5 draws conclusive remarks and focus on future developments.

2 Timed Workflows with Recovery

A business process is a set of logically related tasks performed to achieve a well defined business outcome. Examples of typical business processes are elaborating a credit claim, hiring a new employee, ordering goods from a supplier, creating a marketing plan, processing and paying an insurance claim, and so on. Many computer systems are already available in the commercial marketplace to address the various aspects of Business Process Management (BPM) and automation.

An automated business process is generally called *business workflow*, i.e a choreographed and system-driven sequence of activities directed towards performing a cer-

tain business task to completion. By *activity* we mean an element that performs a specific function within a process. Activities can be as simple as sending or receiving a message, or as complex as coordinating the execution of other processes and activities. A business process may encompass complex activities some of which run on back-end systems such as, for example, a credit check, automated billing, a purchase order, stock updates and shipping, or even such frivolous activities as sending a document and filling a form.

Workflow is commonly used to define the dynamic behaviour of business systems and originates from business and management as a way of modelling business processes that could wholly or partially be automated. It has evolved from the notion of process in manufacturing and offices because these processes are the result of trying to increase efficiency in routine work activities since industrialization.

The view on a workflow which is inherited from the BPM perspective – i.e. the way in which workflow designers may see a system – is somehow different from the way formalists see it. Therefore, to fill the gap between the formal and informal world, we will provide the reader with a precise understanding introducing a formal definition of a business workflow. However, our notation is suitably abstract enough to represent a large number of different modelization formalisms, such as those based on State Machines (Statecharts [25], UML Activity Diagrams [35] and Petri Nets) or specialized to represent business processes, such as BPEL [34]. In fact, one of the purposes of this work is defining a general notation able to include most of the specialized constructs of these languages, by abstraction. How this abstraction is performed is out of the scope of the paper.

A workflow is a directed graph defined by pair (A,T), where A is a finite non empty set of places (or *activities*) and T is a relation that is defined as $T \subseteq A \times A \times \mathbb{N}$. Elements of T are pairs (p,q,d), with $p,q \in A$ and $d \in \mathbb{N}$, that are called *transitions* (later indicated by t^d_{pq}). When d is zero, the transition is called zero-time and it is simply written t_{pq} otherwise d is the delay of executing t^d_{pq}. Let a be a place of A and $time : A \to \mathbb{N}$ be a function labelling activities of the workflow. We write a_c, with $c = time(a)$, for the activity a which lasts c time units. We assume that activities have non null duration, i.e., function *time* never nullifies. Set $out(a)$ is the set of *outgoing transitions* starting from a which is defined as $\{(a,q) \mid q \in A, (a,q) \in T\}$. Set $in(a)$ is the set of *ingoing transitions* leading to p which is defined as $\{(q,a) \mid q \in A, (q,p) \in T\}$.

We assume that $|out(a)| \geq 1$, for all $a \in A$, except for place *end*, and that $|in(a)| \geq 1$, for all $a \in A$, except for place *start*.

A *finite path* from a to a' is a (finite) sequence of pairs $(a_0,a_1)\ldots(a_{n-1},a_n)$ with $a_0 = a$ and $a_n = a'$, such that $(a_i,a_{i-1}) \in T$, for all $1 \leq i \leq n$. An *infinite path* from a is an (infinite) sequence of pairs $(a_i,a_{i-1}) \in E$, for all $i \geq 1$, where $a_0 = a$. As in [21], we assume that workflows are structurally correct, that is, such that there exists at least one path from place *start* to (any) place *end*. The CLTLoc modelling allows us to define precisely all the executions of a timed workflow that, informally, are the superposition of paths of the workflow starting from the initial place.

Conditional cases and *split-join* activities have been already considered in [21]. In this paper, we refine their modelling to make it compliant with the real-time semantics. We briefly recall their intuitive meaning. Conditional cases model *if-then-else* blocks

provided with the usual semantics. If the condition holds the "then" branch is executed otherwise the execution flow follows the "else" branch. Split-join activities model the parallel execution of two (or more) branches of the workflow that starts concurrently when activity *split* is executed and eventually synchronize their computations in correspondence with the associated *join* activity. We assume that conditional cases and split-joins are fictitious activities with non relevant time duration. Therefore, the execution of conditional cases and split-join is considered instantaneous with no time consumption and causes the related activity to start at the same instant where they occur. However, a non null duration may still be associated with these activities by defining a non-zero time incoming transition.

We consider workflows that are endowed with *exceptions*, which are events (or signals) representing erroneous configurations that occur during the workflow execution and that may prevent it from reaching a final place. With no loss of generality, we assume that an exception (raised at some moment throughout the execution) that is not managed forces the running activities that monitor that exception not to terminate. The termination of an execution, and then of all the activities occurring therein, can only be guaranteed if *end* is reached. The assumption is not too strong and does not prevent modelling an activity, say a, that terminates with an error configuration. In fact, one can introduce an exception to represent the wrong termination of a and a special activity that detects it and that is specifically devised for managing faulty termination of a. In addition, workflow executions are not restricted only to finite paths (from *start* to *end*) and infinite iterations of finite paths of the workflow are still allowed. In fact, infinite executions are representative of wrong behaviours only when there is one (ore more) activity, over some paths, that can not terminate and does not allow the workflow to proceed further and reach *end*. To guarantee that a workflow is correctly designed, all the exceptions that may raise during an execution have to be caught and solved. Designers should prevent anomalous situations by defining suitable recovery actions that restore the workflow execution.

Exceptions associated with a workflow are partitioned into the set of permanent (i.e., non-punctual) exceptions and the set of punctual exceptions, as in [21]. Informally, we say that an exception is *punctual* when its duration is negligible, whereas we say that an exception is *non-punctual* when it may have a duration lasts from a position where it is raised until a position where it expires. In this paper, since we extend the modelling through real-time constraints we allow permanent exception to have an exact duration by which it must be handled. If this is not the case, then the workflow does not terminate. Punctual exceptions are not associated with any duration as they must be solved by some activity that is already underway. Activities in the workflow can be associated with three, possibly empty, sets of exceptions: (i) the set of exceptions that activity can notify whenever a potential dangerous error may compromise the workflow execution and that have to be suitably handled by some other activity which is able to repair the fault; (ii) the set of exceptions that activity can handle and the set of exceptions that may compromise the workflow execution because they let activity switch to an error state, if no activity catching them is active at the same time.

3 Formal Semantics

Constraint LTL (CLTL [14,9]) is an extension of LTL allowing atomic formulae over a constraint system. Let V be a finite set of variables and let $\mathcal{D} = (\mathbb{R}, \{<, =\})$ be a constraint system over which formulae are interpreted. In this paper, we consider a fragment of CLTL where temporal terms α are defined as: $\alpha := c \mid x$, where c is a constant in $\mathbb{N}$ and x is a variable in V.

An atomic constraint is a term of the form $\alpha_1 \sim \alpha_2$, where $\sim \in \{<, =\}$, α_1 and α_2 are temporal terms. Well-formed CLTL formulae are defined as follows:

$$\phi := p \mid \alpha_1 \sim \alpha_2 \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}(\phi) \mid \mathbf{Y}(\phi) \mid \phi \mathbf{U} \phi \mid \phi \mathbf{S} \phi$$

where $p \in AP$, every α_i' is a temporal term, $\sim \in \{<, =\}$, **X**, **Y**, **U** and **S** are the "next", "previous", "until" and "since" operators of LTL. The semantics of CLTLoc is defined with respect to $\mathcal{D}$ and the order $(\mathbb{N}, <)$ representing positions in time. An interpretation is a pair (π, σ), where $\sigma : \mathbb{N} \times V \to D$ is a mapping assigning for every variable $x \in V$ its value $\sigma(x, i)$ at each position $i \in \mathbb{N}$ and $\pi : \mathbb{N} \to \wp(AP)$ is a mapping associating a set of propositions with each position in $\mathbb{N}$. The semantics of CLTL at a position $i \in \mathbb{N}$ over an interpretation (π, σ) is defined in Table 1 (Boolean connectives are omitted for brevity). A formula $\phi \in$ CLTL is satisfiable if there exists a pair (π, σ) such that $(\pi, \sigma), 0 \models \phi$. In

$$\begin{aligned}
(\pi,\sigma), i &\models p \Leftrightarrow p \in \pi(i) \text{ for } p \in AP \\
(\pi,\sigma), i &\models \alpha_1 \sim \alpha_2 \Leftrightarrow \sigma(i, x_{\alpha_1}) \sim \sigma(i, x_{\alpha_2}) \\
(\pi,\sigma), i &\models \neg\phi \Leftrightarrow (\pi,\sigma), i \not\models \phi \\
(\pi,\sigma), i &\models \phi \wedge \psi \Leftrightarrow (\pi,\sigma), i \models \phi \text{ and } (\pi,\sigma), i \models \psi \\
(\pi,\sigma), i &\models \mathbf{X}(\phi) \Leftrightarrow (\pi,\sigma), i+1 \models \phi \\
(\pi,\sigma), i &\models \mathbf{Y}(\phi) \Leftrightarrow (\pi,\sigma), i-1 \models \phi \wedge i > 0 \\
(\pi,\sigma), i &\models \phi \mathbf{U} \psi \Leftrightarrow \exists\, j \geq i : (\pi,\sigma), j \models \psi \wedge (\pi,\sigma), n \models \phi \;\forall\, i \leq n < j \\
(\pi,\sigma), i &\models \phi \mathbf{S} \psi \Leftrightarrow \exists\, j \leq i : (\pi,\sigma), j \models \psi \wedge (\pi,\sigma), n \models \phi \;\forall\, j < n \leq i
\end{aligned}$$

Table 1: Semantics of CLTL

this case, we say that (π, σ) is a model of ϕ and we write simply $(\pi, \sigma) \models \phi$.

CLTLoc[10] is a special case of CLTL, where the arithmetical variables behave as clocks, as in Timed Automata [3]. The logic is the first decidable extension of LTL that embeds the notion of dense time through explicit clocks in the language for which there is an implemented decision procedure freely available. Intuitively, a clock x measures the time elapsed since the last time when $x = 0$, i.e., the last "reset" of x. To ensure that time progresses at the same rate for every clock, σ must satisfy the following condition: for every position $i \in \mathbb{N}$, there exists a "time delay" $\delta > 0$ such that for every clock $x \in V$ either time progress, i.e., $\sigma(i+1, x) = \sigma(i, x) + \delta$, or the clock is reset , i.e., $\sigma(i+1, x) = 0$. The initial value of a clock, $\sigma(0, x)$, may be any non-negative value. If needed, a clocks x may be initialized to c just by adding a constraint of the form $x = c$.

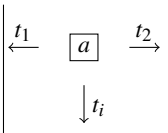
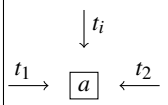
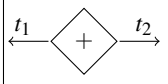
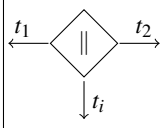
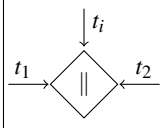
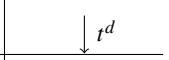
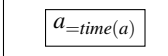
$[a]_{t_{out}(a)}$	$a \Rightarrow (a \wedge \neg t_{\text{out}}(a))\mathbf{U}(t_{\text{out}}(a)) \vee \mathbf{G}(a)$	(1)	
	$\bigwedge_{t \in out(a)} (t \Rightarrow \mathbf{Y}(a) \wedge \neg a)$	(2)	
$t_{in(a)}[a]$	$a \Rightarrow (a \wedge \neg t_{\text{in}}(a))\mathbf{S}(t_{\text{in}}(a))$	(3)	
	$\bigwedge_{t^d \in in(a)} (t^d \Rightarrow \mathbf{X}(a) \wedge \neg a)$	(4)	
	$\bigwedge_{t \in in(a)} (t \Rightarrow \neg\mathbf{Y}(a) \wedge a)$	(5)	
$[\cdot + \cdot]$	$t_1 \Rightarrow \neg t_2$	(6)	
	$\oplus \Rightarrow \neg\mathbf{Y}(\oplus) \wedge \neg\mathbf{X}(\oplus)$	(7)	
	$t_1 \vee t_2 \Leftrightarrow \oplus$	(8)	
$[\cdot \mid \cdot]$	$\bigwedge_{t_1,t_2 \in out(\|)} (t_1 \Leftrightarrow t_2)$	(9)	
	$\| \Rightarrow \neg\mathbf{Y}(\|) \wedge \neg\mathbf{X}(\|)$	(10)	
	$t_i \Leftrightarrow \|$	(11)	
$[\cdot \mid \cdot]$	$\bigwedge_{t_1,t_2 \in in(\|)} (t_1 \Leftrightarrow_2)$	(12)	
	$\| \Rightarrow \neg\mathbf{Y}(\|) \wedge \neg\mathbf{X}(\|)$	(13)	
	$t_i \Leftrightarrow \|$	(14)	
t^d	$t^d \Leftrightarrow x_t = 0 \wedge \mathbf{X}(x_t = d)$	(15)	
	$a \wedge \neg\mathbf{Y}(a) \Leftrightarrow x_a = 0$	(16)	
	$\mathbf{Y}(a) \wedge \neg a \Rightarrow x_a = time(a)$	(17)	

Table 2: Workflow LTL encoding. For convenience, transitions are labeled with numeric pedices.

Workflow model Workflows model execution of systems as sequences of activities. Transitions, conditional case and split-join interleave the activities and determine uniquely the execution flow, i.e., the sequence of activities that realizes the computation. An activity is an abstraction of a compound of actions that are performed by the real workflow. Although they can be modelled as atomic computations, we adopt a different perspective for which the activities, being actions in the real world, have a non-punctual duration. To translate workflows into an CLTLoc formula, we assume that all the activities (except for the activities *start* and *end*) are always followed by a transition, and viceversa, and that conditional case and split-join are special activities that have punctual duration. When an activity is performed, the firing of the outgoing transition lets the system change allowing it to execute the next activity.

Let (A,T) be a workflow. With no loss of generality, we assume that no element in the graph is duplicated. By this assumption, we associate each activity with an atomic proposition that uniquely identifies it. We write t_{pq} to indicate an element $(p,q) \in T$, i.e., a transition between activities $p,q \in A$. If activity $a \in A$ holds at position i then the workflow is performing activity a at that position; similarly for t. We introduce $\|$ and $\oplus$ to indicate a split-join activity and a conditional case activity, respectively; *start* and *end* to indicate the starting and the final activity of the workflow. Workflow diagrams are translated according to rules in Table 2.

Let $t_{\text{out}}(a)$ be the disjunction $\bigvee_{t \in out(a)} t$ and $t_{\text{in}}(a)$ be the disjunction $\bigvee_{t \in in(a)} t$.

Table 2 summarizes the CLTLoc formulae defining the translation of the workflow. We slightly depart from the formalization provided in [21] to model zero-time transitions and the duration of activities. We now describe all formulae defining such aspects while motivations for formulae that are not detailed here can be found in [21].

Zero-time transitions are modelled through Formula 5. Let t be a transition reaching activity a, i.e., such that $t \in in(a)$. When t fires at position i then a is true in i but it does not at position $i-1$ where an activity b such that $t \in out(b)$ was active. Formula 4 and 5 are different, as the former, combined with Formula 2, forces the absence of the two activities a and b when t_{ab} occurs; the latter lets t_{ab} fire exactly at the first position where b holds. In other words, a transition t_{ab} between activity a followed by b, is non zero-time when, if i is the position where it fires, activity a holds at position $i-1$, where it ends, and activity b holds at position $i+1$, where it starts, but none of them at position i. Conversely, a transition t_{ab} between activity a followed by b, is zero-time when activity a holds at position $i-1$ and activity b holds at position i, where it starts exactly at the same position when t_{ab} holds.

Fig. 1 shows an example of a zero-time and non zero-time transition.

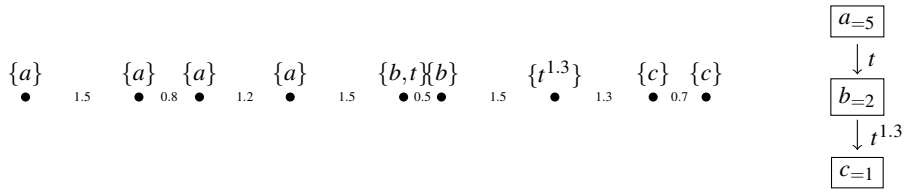


Fig. 1: Portion of an execution (left) of the sequence of activities a,b and c (right) where t is a zero-time transition between a and b and $t^{1.3}$ is a non zero-time transition that lasts 1.3 time units. The delays between two consecutive positions of the CLTLoc representation of the execution are shown in the picture between any pair of bullets. For instance, the time elapsing between position 2 and 3 is 0.8 time unit. Zero-time transition t occurs in the first position of the execution of activity b which, therefore, begins exactly when activity a terminates. Transition $t^{1.3}$ has a non null duration and then it separates activity b from c.

Given an activity $a \in A$, the CLTLoc semantics that we associated with $[a]_{t_{out}(a)}$ and ${}_{t_{in}(a)}[a]$ does not impose any constraint on the firing of transitions in the sets $out(a)$ and $in(a)$, making the execution flow non-deterministically determined. Formulae (1)-(4)

are the same of those presented in [21]. The only difference is in Formula (4) which is now written for non zero-time transitions.

Conditional case $[\cdot + \cdot]$ and split-join $[\cdot \mid \cdot]$ activities are modelled similarly to [21] by formulae (1)-(5) . However, we introduce formulae (8), (11) and (14) to model the zero duration of split/join activities and conditional cases. They enforce the contemporaneity of the activities $\oplus$ and $\|$ with the outgoing transitions, that are assumed to be zero-time. Therefore, their behaviour is modelled by Formula (5).

Adding time to workflows requires care in modelling split-join activities and the duration of their branches. It is, in fact, possible to write unfeasible split-join blocks that do not allow any execution to reach the join when the branches are not "temporally synchronized", that is, when there is at least a pair of paths in the split-join block whose duration (the sum of the duration of all the activities and non zero-time transitions over the path) is not equal. In such a case, the designer must introduce special activities, that are not functionally related to the workflow, to delay the execution of those paths in the split-join which have different duration. In Section 4, we show how our approach can be used to refine the model and design split-join activities that are correctly synchronized.

Formulae (15)-(17) are new and not part of [21] as they model the real-time temporal behaviour of activities and transitions. Fig. 2 shows an example of constraints on clocks measuring delays for activities and non zero-time transitions. Formula (15) defines the duration of non-zero time transitions. For each non-zero time transition t_i^d, we introduce a clock x_i that is reset when t_i^d fires and whose value is exactly d in the next position. To measure activity delays, we introduce clock x_a for activity $a \in A$. Formula (16) defines the condition for resetting x_a that occurs when activity a starts. The formula states that if, at the current position, activity a holds and in the previous position a was not underway, then clock x_a is reset. This allows the clocks to initiate the measuring of the duration of a. When activity finishes then, at that moment, clock x_a must evaluate to $time(a)$. In a terminating run, all activities terminate and eventually are such that $\mathbf{Y}(a) \wedge \neg a$. Then, all the executions meet the temporal constraints on the duration of the activities by enforcing the consequent of Formula (17). In case of workflow errors, some activities of the workflow may loop forever.

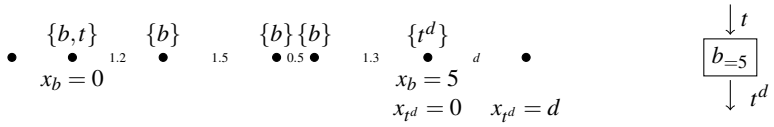


Fig. 2: Execution (left) of activity b and transitions t and t^d depicted on the right, where t is a zero-time transition preceding b and t^d is a non zero-time transition that lasts d time units. The delays between two consecutive positions of the CLTLoc representation of the execution are non-deterministically determined to meet the constraint on the duration of the activity imposed in Formula (17). In position 2, activity b starts and clock x_b is reset. The constraint $x_b = 5$ on the duration of activity b is met in the next position where b terminates (in the example this holds when t^d occurs). Formula (15) models temporal constraints on t^d. When t^d occurs, clock x_{t^d} is reset. In the next position, it meets the constraint $x_{t^d} = d$.

In such a case, the antecedent of the implication is false, as the activity satisfies $\mathbf{G}(a)$, and does not constraint the value of clock x_a to any specific value. Along the execution of activity a, its clock x_a has a value which is less than the duration $time(a)$ of the activity itself.

As in [21], we assume that when a workflow terminates, it never resumes, by adding to the model formula $end \Rightarrow \mathbf{G}(end)$.

Encoding exceptions Let E be a (finite) set of exceptions associated with the workflow and P and S be two subsets of E such that $P \cup S = E$ and $P \cap S = \emptyset$ where P is the set of permanent exceptions and S is the set of punctual exceptions. In this section, with abuse of notation, we restrict set A only to activities that are not *start*, *end*, split-join and conditional activities with which no exception is associated.

Informally, we say that an exception is punctual when it holds exactly one time instant whenever it occurs. Conversely, an exception is non-punctual when it may have a duration and it lasts from a position where it is raised until a position where it expires. In this paper, we focus on non-punctual exceptions for which the modelling is modified with respect to [21] to deal with time constraints. Formulae defining the behaviour of punctual exceptions are not shown here as they are the same as those presented in [21].

Let a be an activity and $catch(a)$ be the set of exceptions that activity a can restores. Since non-punctual exceptions may hold continuously over some adjacent positions, when such an exception occurs, say e, at some position, it holds until an activity a such that $e \in catch(a)$ restores the exception. To store the time elapsed since its generation, we introduce a clock x_e, for all $e \in P$. Formula (18) imposes that clock x_e is reset when exception e is thrown.

$$\bigwedge_{e \in P} (e \wedge \neg \mathbf{Y}(e) \Leftrightarrow x_e = 0). \tag{18}$$

We extend function *time* to element of set E. Formula (19) is different from the one presented in [21] as it also includes the timing constraint on clock x_e to meet the deadline $time(e)$. It states that if, at the current position, e holds then there is a position in the future where an activity restores it before the deadline $time(e)$, otherwise it will hold indefinitely. In fact, if the right-hand formula holds, i.e., the exception is managed correctly before the deadline, then $\neg\mathbf{G}(e)$ holds and e will not hold indefinitely. Conversely, if the right-hand formula does not hold then $\neg\mathbf{G}(e)$ does not holds, that is, e will hold indefinitely.

$$\bigwedge_{e \in P} (e \Rightarrow (\neg\mathbf{G}(e) \Leftrightarrow \bigvee_{\substack{a \in A \\ e \in catch(a)}} (e\mathbf{U}(a \wedge x_e \leq time(e))))). \tag{19}$$

Any non-punctual exception which is not properly resolved by some activities of the workflow before its deadline causes the workflow to fall into an error configuration and lets the execution loop forever. Formula (20) states that in exception e is active and its clock is greater than its deadline then the exception remains active and endures indefinitely. Formula (20), conjoined with Formula(19), avoids the occurrence of an already managed exception e after its deadline, because otherwise $\mathbf{G}(e)$ holds which,

by Formula(19), is equivalent to not having an activity that managed e.

$$\bigwedge_{e \in P} (e \wedge x_e > time(e) \Rightarrow \mathbf{G}(e)). \tag{20}$$

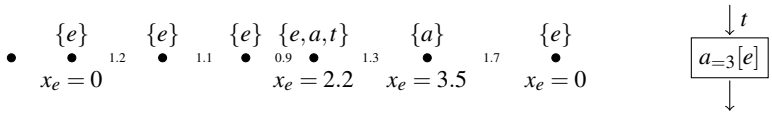


Fig. 3: Portion of the execution (left) of exception e with deadline $time(e) = 3$ caught by activity a, i.e., such that $e \in catch(a)$. In position 5, where $x_e = 2.2$, activity a solves the exception before its deadline. Therefore, e does not occur in the next position, that occurs after 3 time units since the generation of e, because of Formula (20) and Formula 20. Exception e is newly thrown in the last position of the sub-execution where clock x_e is reset.

Let $probe(a)$ be the set of exceptions associated with activity a that may let a loop indefinitely. If a is active at a certain position of the time, then the occurrence of (i) an exception e in $probe(a)$ causes an abortion of a if, at that moment, there is no activity b that restores e, such that $e \in catch(b)$ or (i) an exception e has not met its deadline. The abortion represents a configuration of error that can not be restored, i.e., a loops infinitely or terminates with a system error. Formula (21) is the same as the one in [21] and is defined for all activities $a \in A$ of the workflow with a non empty set $probe(b)$

$$\bigwedge_{e \in probe(a)} (a \wedge (\mathbf{G}(e) \vee e \wedge \bigwedge_{\substack{b \in A \\ e \in catch(b)}} \neg b)) \Rightarrow \mathbf{G}(a) \tag{21}$$

Formula (22), which is introduced for all activities $a \in A$ appearing in the workflow, defines the necessary conditions to have infinite execution. It is modified with respect to the correspondent one in [21] because the disjunct $\bigvee_{e \in P} \mathbf{G}(e)$ is added to the formula. At a certain position, if activity a is active and it never terminates, i.e., $\mathbf{G}(a)$ holds at that position, then (i) there exists an activity c of the workflow, possibly different from a, that eventually loops indefinitely because an exception $e \in probe(c)$ is not correctly handled or (ii) there is an exception e whose deadline has not met which continues indefinitely (due to Formula (20)). If activity a holds forever, then there is an activity c (which may possibly be a) and a non-punctual exception $e \in probe(c)$ that holds indefinitely, because no activity b, that actually could manage e, ever catches it.

$$\mathbf{G}(a) \Rightarrow \mathbf{F}\left(\bigvee_{\substack{c \in A \\ e \in probe(c)}} (\mathbf{G}\left(c \wedge e \wedge \bigwedge_{\substack{b \in A \\ e \in catch(b)}} \neg b\right) \vee \bigvee_{e \in P} \mathbf{G}(e \wedge c))\right) \tag{22}$$

Observe that when $probe(c)$, for some $c \in A$, is empty the formula within $\mathbf{F}$ is trivially false. In this case, the activity appearing in the antecedent of the formula always terminates and no looping executions are admitted for it, because $\mathbf{G}(a)$ is false.

An exception $e \in E$ is *internal* if it is thrown by some activity appearing in the workflow whereas it is *external* otherwise. The same formalization in [21] holds in this context, so the reader may refer to [21] for further details.

We can now, formally, define the executions of a workflow. Let W be a workflow and ϕ_W the CLTLoc formula translating W that is defined by conjunction the formulae above, globally quantified over the time. We define *execution* of W an CLTLoc interpretation (π, σ) for formula ϕ_W such that $(\pi, \sigma), 0 \models \phi_W$.

4 An example

To demonstrate the soundness and effectiveness of the proposed approach, it has been applied to the investigation of an example taken from literature and related to an office process. This way, in the following, we first describe the office workflow and then we report on how to apply our framework to the verification of some basic properties of the overall process.

4.1 The model

A workflow for processing generic good requests, referable to a quite large class of small and medium enterprises (for details see [18]), is described in Fig. 4. Even if simple, it could represent, from a high level perspective, a broad class of actual e-commerce or similar (remote) purchase systems. In the corresponding workflow model we included some design flaws on exception handlers that may drive to neverending executions.

More specifically, the office workflow is composed of ten activities, drawn as rectangles, which could generate three types of exceptions: *HF* (Hardware Failure), *SF* (Software Failure) and *TF* (Transport Failure). The first two of them (HF and SF) are permanent exceptions, while TF ones are punctual exceptions. The *probe* and *throw* exception sets related to an activity are identified by all its ingoing or outgoing labelled arrows, respectively. This way we have $throw(InternalCreditCheck) = \{HF, SF\}$ and $throw(Shipping) = \{TF\}$. The set of exceptions caught by an activity is specified within square brackets close to its name in the rectangle ($catch(Recovery) = \{SF\}$ and $catch(Reject_2) = \{TF\}$). Conditions are represented by diamonds and labelled with ? or with $<$ *Condition Name* $>$? (*SF recovery available?*). Similarly, split-join concurrent activities are depicted by diamonds labelled with $\|$. Both condition and split-join activities are instantaneous activities thus characterized by zero-time durations. Finally, the number specified as subscript of labels represents the time spent in an activity with the corresponding time unit (*m* stands for minutes), i.e. the activity time durations. With regard to exceptions, this time represents a timeout: once expired the exception can no longer be handled and/or recovered.

Verifying the reachability property $\mathbf{G}(\neg Arch)$, we prove that *Archiving* activity is not reachable. So, to synchronize the execution of the two branches, we have introduced a new dummy activity *Delay* to equal the duration of activity *Shipping*.

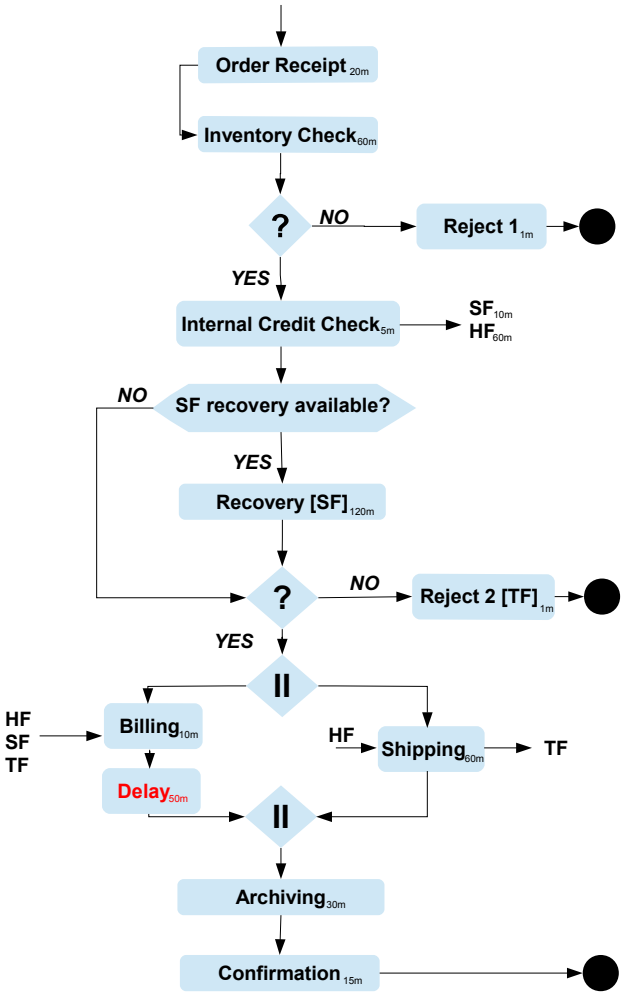


Fig. 4: Office workflow representation.

4.2 Verification

To demonstrate the effectiveness of our approach in this section we verify (the CLTLoc model of) the workflow described above to identify wrong execution patterns taking into account different, alternative scenarios characterized by the exceptions, thus obtaining hints able to drive the model refinement towards a correct design. In this section, we discuss on the performance results of the verification process and the satisfiability of properties.

To validate the CLTLoc model, we exploit the Bounded Satisfiability Checking (BSC) [38] approach similarly to [21].

Table 3 shows time – in seconds – required by ℤot to verify a set of functional user-defined properties, memory occupation – in MBytes – and the result, i.e. whether the property is satisfied or not.

Formula $\mathbf{G}(\neg tf \wedge \neg hf \wedge \neg sf \Rightarrow \mathbf{F}(end))$ states that, if no exceptions occur, the workflow must terminate, as in [21]. As reported in Table 3, the property holds also in the timed version of the workflow, since there are no traces which satisfy its negation.

Formula $(\mathbf{G}(\neg hf \wedge \neg sf) \wedge \mathbf{F}(tf \wedge x_{Shipp} = 5)) \Rightarrow \mathbf{F}(end)$ checks the occurrence of the *TransportFailure* punctual exception, which is thrown by *Shipping* activity and models a shipping problem, such as a truck accident. The variable x_{Shipp} is the clock associated to activity *Shipping*, which counts the time elapsed since the beginning of the execution of the activity: in this formula, we want to simulate the thrown of the *tf* exception after 5 time units – minutes, in the case study – from the beginning of the activity. As reported in Table 3, the property does not hold: the counterexample trace shows, as in [21], that the activities *Billing* and *Shipping* loop forever. Having a look at the workflow, we observe that some activities never terminate since the exception *tf* can be caught only by activity *Reject2*, which can not be executed in parallel with *Shipping*, and because activity *Billing* catches the exception before its deadline, which is 10 time units.

Formula $(\mathbf{G}(\neg hf \wedge \neg sf) \wedge \mathbf{F}(tf \wedge x_{Shipp} = 25)) \Rightarrow \mathbf{F}(end)$ checks, again, the occurrence of the *TransportFailure* punctual exception, but in a scenario where it is thrown after activity *Billing* has finished its execution. In this case, there are no parallel activities that catch the exception, since activity *Delay* is only a placeholder to wait for synchronization, so the workflow terminates.

All tests have been carried out on a 3.3 Ghz quad core PC with 16 Gbytes of RAM. The bound k, which is a user-defined parameter representing the maximal length of runs analysed by ℤot, corresponds to the number of discrete positions that are used to build the bounded representation of the model. The value chosen is $k = 35$. By analysing the longest path of the workflow of Figure 4, one can see that this value for k is big enough to guarantee the definition of meaningful workflow executions, i.e., interpretations over the symbols appearing in the workflow that are model of the LTL formula translating it (partly shown in and defined through rules of Section 3).

Table 3: Test results.

Formula	Time (s)	Memory (Mb)	Result
$\mathbf{G}(\neg tf \wedge \neg hf \wedge \neg sf \Rightarrow \mathbf{F}(end))$	7.545	25	UNSAT
$(\mathbf{G}(\neg hf \wedge \neg sf) \wedge \mathbf{F}(tf \wedge x_{Shipp} = 5)) \Rightarrow \mathbf{F}(end)$	8.536	180	SAT
$(\mathbf{G}(\neg hf \wedge \neg sf) \wedge \mathbf{F}(tf \wedge x_{Shipp} = 25)) \Rightarrow \mathbf{F}(end)$	7.846	28	UNSAT

To verify properties like the one modelled by Formula $\mathbf{G}(\neg tf \wedge \neg hf \wedge \neg sf \Rightarrow \mathbf{F}(end))$, ℤot must exhaustively analyse all possible runs to return UNSAT, which is the worst case in terms of time and memory consumed; taking it into account, we can conclude that it is feasible, using modern model checking tools such as ℤot, to perform for-

mal verification of non-trivial functional real-time properties, in a limited amount of resources, allowing designer to execute the analysis in an interactive real-time manner.

5 Conclusions and future work

The major objective of this paper is demonstrating how temporal logics are effective in giving semantics and iteratively enforce requirements into the process. To this purpose, starting from [21], we extended the previous LTL semantics formalization of workflows to include timed activities. To model timed workflow we exploit CLTLoc [11], which is an LTL based logic where atomic formulae are both atomic propositions and constraints over dense clocks. The implemented solution is able to verify time behavior of a wide class of workflows, as also demonstrated by an example of a generic office business process.

The workflow patterns here analyzed are limited with respect to a real scenario, where more complex patterns, as the ones identified in [2], need to be investigated and encoded into our approach. Once workflows are intended as graphs and transitions are treated like in this paper, similarities emerge with the Petri Nets approach, in particular with Workflow Petri Nets [1]. Other formalisms such as the business process modeling notation (BPMN) [36] could be considered as starting point for our approach. Indeed, in [31] BPMN has been exploited for workflow design since it includes the concept of partition (modeled as pools and swimlanes), an essential features for business processes modeling not considered here. This will need to be investigated later.

Zero-time modeling is also an open issue. When some workflow activities have a negligible duration with respect to the other ones, they may be modeled as having a logical zero time duration. This implies Zeno behaviours and other counterintuitive consequences. [20] introduces a new metric temporal logic called *X-TRIO*, which exploits the concepts of *Non-Standard Analysis* [39]. The way to "glue" together CLTLoc with X-TRIO is a promising research strand.

Finally, runtime evolution in business processes [7] and, more in general, the idea of self-reconfiguring systems are related issues we intend to further explore.

Acknowledgements The authors acknowledge the support and advice given by Marina Carvalho, Miticus Flamejante, Vínicius Pereira, Diego Pérez, Michele Ciavotta, Marco Miglierina and all the other Friends at Politecnico di Milano, which represent a moving force, an actual égrégore capable of always moving ideas forward to the next level.

References

1. Aalst, W.M.P.v.d.: Verification of workflow nets. In: Proceedings of the 18th International Conference on Application and Theory of Petri Nets. pp. 407–426. ICATPN '97, Springer-Verlag, London, UK, UK (1997)
2. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comp. Sci. 126(2), 183–235 (1994)

4. Attie, P.C., Singh, M.P.: Specifying and enforcing intertask dependencies. In: In Proceedings of the 19th VLDB Conference. pp. 134–145 (1993)
5. Augusto, J.C., Howard, Y., Gravell, A.M., Ferreira, C., Gruner, S., Leuschel, M.: Model-based approaches for validating business critical systems. In: STEP. pp. 225–233 (2003)
6. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
7. Baresi, L., Guinea, S., Manna, V.P.L.: Consistent runtime evolution of service-based business processes. In: Anna Liu, John Klein, A.T. (ed.) Working IEEE/IFIP Conference on Software Architecture (WICSA) (2014)
8. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: A logic-based semantics for the verification of multi-diagram uml models. ACM SIGSOFT Software Engineering Notes 37(4), 1–8 (2012)
9. Bersani, M.M., Frigeri, A., Rossi, M., San Pietro, P.: Completeness of the bounded satisfiability problem for constraint LTL. In: Reachability Problems, LNCS, vol. 6945, pp. 58–71. Springer (2011)
10. Bersani, M.M., Rossi, M., Pietro, P.S.: A tool for deciding the satisfiability of continuous-time metric temporal logic. In: 2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, FL, USA, September 26-28, 2013. pp. 99–106 (2013), http://doi.ieeecomputersociety.org/10.1109/TIME.2013.20
11. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. In: Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME). pp. 99–106 (2013)
12. Butler, M.J., Ferreira, C.: An operational semantics for stac, a language for modelling long-running business transactions. In: Nicola, R.D., Ferrari, G.L., Meredith, G. (eds.) COORDINATION. Lecture Notes in Computer Science, vol. 2949, pp. 87–104. Springer (2004)
13. Calzolai, F., Nicola, R.D., Loreti, M., Tiezzi, F.: Tapas: A tool for the analysis of process algebras. T. Petri Nets and Other Models of Concurrency 1, 54–70 (2008)
14. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Information and Computation 205(3), 380–415 (2007)
15. Díaz, M., Valero, V., Macía, H., Mateo, J., Díaz, G.: Bpel-rf tool: An automatic translation from ws-bpel/wsrf specifications to petri nets. In: ICSEA 2012 : The Seventh International Conference on Software Engineering Advances (2012)
16. Dragoni, N., Mazzara, M.: A formal semantics for the ws-bpel recovery framework - the pi-calculus way. In: Laneve, C., Su, J. (eds.) WS-FM. Lecture Notes in Computer Science, vol. 6194, pp. 92–109. Springer (2009)
17. Eisentraut, C., Spieler, D.: Web services and formal methods. chap. Fault, Compensation and Termination in WS-BPEL 2.0 – A Comparative Analysis, pp. 107–126. Springer-Verlag, Berlin, Heidelberg (2009)
18. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of Conference on Organizational Computing Systems. pp. 10–21. COCS '95, ACM, New York, NY, USA (1995)
19. Eshuis, R., Wieringa, R.: Verification support for workflow design with uml activity graphs (2002)
20. Ferrucci, L., Mandrioli, D., Morzenti, A., Rossi, M.: A metric temporal logic for dealing with zero-time transitions. In: Proc. of 19th International Symposium on Temporal Representation and Reasoning. pp. 81–88. IEEE Computer Society (2012)
21. Ferrucci, L., Bersani, M.M., Mazzara, M.: An LTL semantics of businessworkflows with recovery. In: ICSOFT-PT 2014 - Proceedings of the 9th International Conference on Software Paradigm Trends, Vienna, Austria, 29-31 August, 2014. pp. 29–40 (2014)
22. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Storey, T., Weerawaranna, S.: Modeling stateful resources with web services (2004), http://toolkit.globus.org/wsrf/

23. Ghezzi, C., Mandrioli, D., Morzenti, A.: Trio: A logic language for executable specifications of real-time systems. J. Syst. Softw. 12(2), 107–123 (1990)
24. Gmehlich, R., Grau, K., Iliasov, A., Jackson, M., Loesch, F., Mazzara, M.: Towards a formalism-based toolkit for automotive applications. In: Formal Methods in Software Engineering (FormaliSE) (2013)
25. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8(3), 231 – 274 (1987), http://www.sciencedirect.com/science/article/pii/0167642387900359
26. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Proc. of Logics of Programs. LNCS, vol. 193, pp. 196–218. Springer (1985)
27. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
28. Mazzara, M.: Deriving specifications of dependable systems: toward a method. In: 12th European Workshop on Dependable Computing (EWDC) (2009)
29. Mazzara, M.: On methods for the formal specification of fault tolerant systems. In: Proceedings of the 4th International Conference on Dependability (DEPEND 2011) (2011)
30. Mazzara, M., Bhattacharyya, A.: On modelling and analysis of dynamic reconfiguration of dependable real-time systems. DEPEND, International Conference on Dependability (2010)
31. Mazzara, M., Dragoni, Nicola Zhou, M.: Implementing workflow reconfiguration in ws-bpel. Journal of Internet Services and Information Security 2(1/2), 73–92 (2012)
32. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with jolie. In: Web Services Foundations, pp. 81–107 (2014)
33. Morzenti, A., San Pietro, P.: Object-oriented logical specification of time-critical systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 3(1), 56–98 (1994)
34. OASIS: Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (2007)
35. OMG: Unified modeling language 2.0. http://www.omg.org/spec/UML/2.0/ (2005)
36. OMG: Business process model and notation (bpmn). http://www.bpmn.org/ (2011)
37. Pradella, M., Morzenti, A., San Pietro, P.: Refining real-time system specifications through bounded model- and satisfiability-checking. In: ASE. pp. 119–127 (2008)
38. Pradella, M., Morzenti, A., San Pietro, P.: Bounded satisfiability checking of metric temporal logic specifications. ACM Trans. on Soft. Eng. and Meth. (TOSEM) (2013)
39. Robinson, A.: Non-standard analysis. Princeton University Press (1996)
40. Romanovsky, A., Thomas, M. (eds.): Industrial Deployment of System Engineering Methods. Springer (2013)
41. Vaz, C., Ferreira, C.: On the analysis of compensation correctness. J. Log. Algebr. Program. 81(5), 585–605 (2012)
42. Yamaguchi, M., Yamaguchi, S., Tanaka, M.: A model checking method of soundness for workflow nets. IEICE Transactions 92-A(11), 2723–2731 (2009)