An LTL Semantics of Business Workflows with Recovery

Luca Ferrucci[1,2], Marcello M. Bersani[2] and Manuel Mazzara[3]
[1]*ISTI-CNR, Pisa, Italy*
[2]*Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy*
[3]*Innopolis University, Russia and ETH Zürich, Zürich, Switzerland*

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

Abstract: We describe a business workflow case study with abnormal behavior management (i.e. recovery) and demonstrate how temporal logics and model checking can provide a methodology to iteratively revise the design and obtain a correct-by construction system. To do so we define a formal semantics by giving a compilation of generic workflow patterns into LTL and we use the bound model checker $\mathbb{Z}$ot to prove specific properties and requirements validity. The working assumption is that such a lightweight approach would easily fit into processes that are already in place without the need for a radical change of procedures, tools and people's attitudes. 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.

1 INTRODUCTION

Nowadays Internet access is widespread and people use the internet for a wide range of activities, among others to purchase goods and services. In Europe in 2012, 75% of individuals aged 16 to 74 had used the internet in the previous 12 months, and nearly 60% of these reported that they had shopped online (data collected by http://epp.eurostat.ec.europa.eu/). Presently Internet purchases represents the most important way of doing E-business while older systems are either been canceled or improved in such a way that they are able to run over the Internet infrastructure.

Together with the emerging of E-business and the exigency of exchanging business messages between trading partners, the concept of business integration arose. Business integration is becoming necessary to allow partners to communicate and exchange documents as catalogs, orders, reports and invoices, overcoming architectural, applicative, and semantic differences, according to the business processes implemented by each enterprise. In order to be effective, a business integration solution must also deal with (non-functional) requirements such as dependability, security, availability and compatibility.

In this work, our focus will be limited to the dependability aspect of business integration and the analysis of recovery solutions. In particular, we will give a formal semantics to business workflows enriched with abnormal behaviour and recovery and will address a methodology to define and verify specific causal properties defined by designers. Causal properties allows to specify order relationships between events and activities.

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 LTL-based semantics of workflow execution and use $\mathbb{Z}$ot (Pradella et al., 2008) 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 (Lucchi and Mazzara, 2007) where a similar problem was approached in term of Process Algebra. A comparison of approaches is left as future work.

Differently from several others formalizations only offering languages without methods – for a detailed discussion see (Mazzara, 2009) and (Mazzara,

Ferrucci L., Bersani M. and Mazzara M..
An LTL Semantics of BusinessWorkflows with Recovery.
DOI: 10.5220/0005110000290040
In *Proceedings of the 9th International Conference on Software Paradigm Trends* (ICSOFT-PT-2014), pages 29-40
ISBN: 978-989-758-037-6
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

2011) – 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 (Gmehlich et al., 2013) 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 to designers 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, Linear Temporal Logic (LTL) is often used with this goal in the verification of finite-state models, e.g., in model checking (Baier and Katoen, 2008). Second, temporal logic can be used as a descriptive approach for specifying and modelling systems (see, e.g., (Morzenti and San Pietro, 1994; Ferrucci et al., 2012)). 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 conjunction of the model and of (the negation of) its desired properties. For example, in Bounded Satisfiability Checking (BSC) (Pradella et al., 2013), Metric Temporal Logic (MTL) specifications on *discrete* time and properties are translated into propositional logic, in an approach similar to Bounded Model Checking of LTL properties of finite-state machines.

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* $\mathrm{throw}(e)$ 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 does not require extending LTL and also allows modelling of two sorts of exceptions (punctual and non-punctual, see Section 3) in a coherent and uniform way by providing events that represent exceptions a suitable semantics.

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 (Lucchi and Mazzara, 2007) that this work intend to complement. Limitations of process algebras approaches like the previous ones and, for example, (Vaz and Ferreira, 2012) 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 (Calzolai et al., 2008)) and tool support is very limited (see (Mazzara and Bhattacharyya, 2010)). On the other side, other works like (Montesi et al., 2014) 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) (Aalst, 1997) 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 (Yamaguchi et al., 2009) 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 crit-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

ical systems. The B-model is one of the most popular together with its reactive-systems extension Event-B (Augusto et al., 2003). B and Event-B are not lightweight methods. They do come with a refinement-based methodology, but cannot easily be embedded into already existing industrial processes (Gmehlich et al., 2013).

In the domain of temporal logics, CTL has been used to specify and enforce intertask dependencies (Attie and Singh, 1993), and LTL for UML activity graphs verification (Eshuis and Wieringa, 2002). Other temporal logics have also been used for similar objectives. In paricular, in (Baresi et al., 2012) a complete and coherent semantics based on the TRIO logic (Ghezzi et al., 1990) has been proposed for a more consistent set of UML diagrams.

Recovery frameworks have been more rarely formalized in similar manners instead. This has to be intended as another major contribution of the paper. One of the first works formally discussing business recovery in terms of long-running transactions is (Butler and Ferreira, 2004). In (Dragoni and Mazzara, 2009) a simplified and clarified semantics of WS-BPEL recovery framework has been presented in terms of Process Algebras. In (Eisentraut and Spieler, 2009) 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 (Díaz et al., 2012) is the integration of WS-BPEL with WSRF (Foster et al., 2004), a resource management language, taking into account the main structural elements of WS-BPEL with event handling and fault handling.

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 (Gmehlich et al., 2013), (Romanovsky and Thomas, 2013).

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 LTL 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 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 certain 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 (Harel, 1987), UML Activity Diagrams (OMG, 2005) and Petri Nets) or specialized to represent business processes, such as BPEL (OASIS, 2007). 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 which is 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 $T \subseteq A \times A$. Elements of T are pairs (p,q), with $p,q \in A$, that are called *transitions* (later indicated by t_{pq}). Let a be a place of A. 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 $(a_0,a_1)\dots(a_{n-1},a_n)$ 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 $a_0 = a$. Throughout the paper, we assume that workflows are structurally correct, that is, such that there exists at least one path from place *start* to (any) place *end*. Informally, an *execution* of a workflow is the superposition of paths of the workflow starting from the initial place. The LTL modelling allows us to define precisely all the executions of a workflow.

Each activity of a workflow may have a specific semantics that forces the execution to be compliant with some specific rules. In general, endowing activities with a semantics is not achievable only by considering workflows as graphs and the definition of complex behaviours may require more specific and expressive formalisms. This is the case of *conditional cases* and *split-join* activities that we consider in this paper whose semantics can be easily obtained by defining concise LTL formulae. 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. In this paper, we do not model guards explicitly, though conditional expressions over finite domains can be easily introduced, as the effort of the work is focused on the exception recovery mechanism. 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-join are fictitious activities with non relevant time duration.

We consider workflows that are endowed with *exceptions*, i.e., specific events (or signals) representing erroneous configurations that occur during the execution and that may prevent the workflow 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 by the workflow, forces the running activities that monitor the exception not to terminate. Under such assumption, the termination of an execution, and then of all the activities occurring therein, can only be guaranteed if *end* is reached. However, the assumption 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.

We assume that the set of exceptions associated with a workflow is partitioned into the set of permanent (i.e., non-punctual) exceptions and the set of punctual exceptions. 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 and it lasts from a position where it is raised until a position where it expires. Each activity $a \in A$ can be associated with three, possibly empty, sets of exceptions. Set $throw(a)$ is the set of exceptions that activity a 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. Set $catch(a)$ contains exceptions that activity a can handle, that is, that the activity may take on responsibility of remedying the fault. Set $probe(a)$ is the set of exceptions that may compromise the workflow execution because they let activity a switch to an error state, if no activity catching them is active at the same time.

In Section 3, we define formally, through LTL formulae, the semantics of correct workflows executions with respect to the specific semantics of conditional cases and split-join activities and recovery exceptions.

The case study approached in this paper is depicted in Figure 1 and describes a typical office workflow for order processing that is commonly found in large and medium-sized organizations (for more details (Ellis et al., 1995)). Although the example may appear too simple, most of the online purchase systems are actually of comparable complexity, apart abstracting from several details. The workflow we provide is a simple example of wrong design as some exceptions are handled incorrectly and may cause infinite executions. To demonstrate the effectiveness of our approach, in Section 4, we verify (the LTL model of) the workflow and we show that discovering wrong executions allows us to enforce model refinement and obtain a correct design. The workflow consists of the ten activities, depicted within rectangles, and is endowed with two permanent exceptions *HF* (Hardware Failure), *SF* (Software Failure) and one punctual exception *TF* (Transport Failure). An ingoing arrow into an activity labelled with exceptions defines the set *probe* of the activity while an outgoing arrow, labelled with exceptions, defines the set *throw*. We have $throw(InternalCreditCheck) = \{HF, SF\}$ and $throw(Shipping) = \{TF\}$. The set of exceptions that an activity catches is defined within square brackets and it is written beside the name of the activity in the rectangle. We have $catch(Recovery) = \{SF\}$ and $catch(Reject_2) = \{TF\}$. Conditional cases are graphically represented by diamonds, and labelled with ? and *SF recovery*. Split-join activities are depicted by diamonds labelled with $\|$.

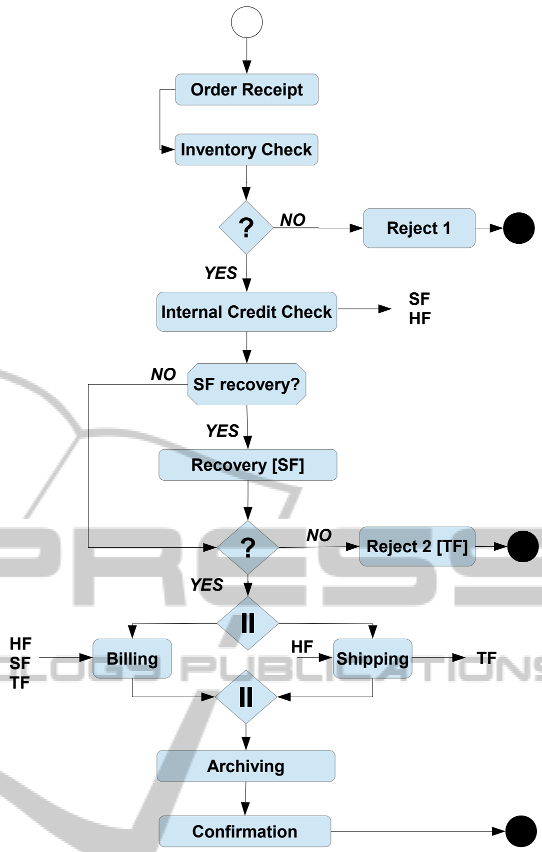


Figure 1: Workflow Case Study.

3 FORMAL SEMANTICS

LTL (Lichtenstein et al., 1985) is one of the most popular descriptive language for defining temporal behaviours that are represented as sequences of observations. The time model adopted in this logic is a totally ordered set (e.g., $(\mathbb{N},<)$) whose elements are the positions where the behaviour is observed. LTL allows the expression of positional orders of events both towards the past and the future. Let *AP* be a finite set of atomic propositions. Well-formed LTL formulae are defined as follows:

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

where $p \in AP$, **X**, **Y**, **U** and **S** are the usual “next”, “previous”, “until” and “since” modalities. The dual operator “release” **R** is defined as usual, i.e., $\phi\mathbf{R}\psi$ is $\neg(\neg\phi\mathbf{U}\neg\psi)$. Useful operators can be defined from the previous ones. “Eventually” is defined as $\mathbf{F}(\phi) = true\mathbf{U}\phi$ and “Globally" $\mathbf{G}\phi$ operator is $false\mathbf{R}\phi$. Informally, $\mathbf{F}(\phi)$ means that ϕ will eventually occur in the future, including the current position, and $\mathbf{G}(\phi)$ means that ϕ holds indefinitely from the current position.

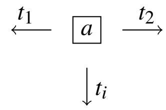
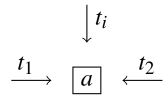
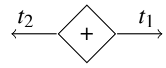
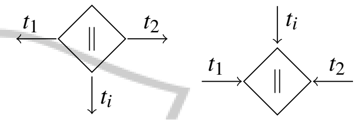
The semantics of LTL formulae is defined with respect to a strict linear order representing time $(\mathbb{N},<)$. Truth values of propositions in *AP* are defined by interpretation $\pi : \mathbb{N} \rightarrow \wp(AP)$ associating a subset of the set of propositions with each element of $\mathbb{N}$. The semantics of a LTL formula ϕ at instant $i \geq 0$ over a linear structure π is recursively defined as in Table 1,

Table 1: Semantics of LTL.

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

A formula $\phi \in$ LTL is *satisfiable* if there exists an interpretation π such that $\pi, 0 \models \phi$.

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

$[a]_{t_{out}(a)}$	$a \Rightarrow (a \wedge \neg t_{\text{out}}(a))\mathbf{U}(t_{\text{out}}(a)) \vee \mathbf{G}(a)$	(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)	
	$\bigwedge_{t \in in(a)} (t \Rightarrow \mathbf{X}(a) \wedge \neg a)$	(4)	
$[\cdot + \cdot]$	$t_1 \Rightarrow \neg t_2$	(5)	
	$\oplus \Rightarrow \neg \mathbf{Y}(\oplus) \wedge \neg \mathbf{X}(\oplus)$	(6)	
$[\cdot \mid \cdot]$	$\bigwedge_{t_1, t_2 \in out(\Vert)} (t_1 \Leftrightarrow t_2)$ or $\bigwedge_{t_1, t_2 \in in(\Vert)} (t_1 \Leftrightarrow t_2)$	(7)	
	$\Vert \Rightarrow \neg \mathbf{Y}(\Vert) \wedge \neg \mathbf{X}(\Vert)$	(8)	

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 LTL 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. Therefore, our LTL translation allows modelling of workflows as sequences of activities and transitions, in strict alternation.

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 $\Vert$ 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 $\bigvee_{t \in in(a)} t$.

Formula (1) states that if activity a holds at the current position, then it is true, at that position, that either a holds forever, which is the case of a workflow that is blocked because of an exception, or a but not $t_{\text{out}}(a)$ holds until at least one transition in $out(a)$ holds. In this way, activity a lasts until one of its outgoing transition fires. Formula (2) imposes that if transition $t_i \in out(a)$ holds at position i, then in the previous one $i-1$, activity a holds. This necessary condition enforces that a transition fires only if the activity from which it originates has just been terminated.

Formula (3) states that if activity a holds at the current position, then it is true, at that position, that a but not $t_{\text{in}}(a)$ holds since at least one transition in $in(a)$ has been fired. In this way, activity a has lasted since one of its ingoing transition fired. Formula (4) imposes that if transition $t \in in(a)$ holds at position i, then, in the next one $i+1$, activity a holds. This necessary condition enforces that a transition fires only if the activity to which it leads will be performed in the next position.

Given an activity $a \in A$, the LTL 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. In fact, formulae (1)-(4) only state that the execution following activity a is realized by some (at least one) transitions of $out(a)$ that fire when a terminates and, symmetrically, that the execution preceding a lets some (at least one) transitions of $in(a)$ fire to start the execution of a.

Conditional case $[\cdot + \cdot]$ and split-join $[\cdot \mid \cdot]$ activities are modelled by using the formulae (1)-(4)

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

and, in addition, specific constraints which enforce the proper execution flow. To model the flow of the conditional cases, we force the execution of the two branches to be exclusive as for the *if-then-else* construct of programming languages. The translation of the split (resp. join) activity is similar yet it enforces the synchronization of all the transitions starting from (resp. yielding to) it.

The conditional case $[\cdot \mid \cdot]$ is translated compositionally. For each conditional activity we introduce a new fresh atomic proposition $\oplus$. Formulae (1), (2), (3) and (4) define the semantics of the activity and Formula (5) imposes the uniqueness of the execution, i.e., that only one branch is executed, by requiring the strict complementarity between t_1 and t_2. Formula (6) enforces the punctuality of $\oplus$ and states that, if $\oplus$ holds at position i then, in the next and in the previous positions, it does not hold.

The translation of the split-join activities is similar to the one defined for the conditional case. For each split-join activity we introduce a new fresh atomic proposition $\|$ and we use formulae (1), (2), (3) and (4) to define the semantics of the activity and Formula (6) to enforce the punctuality of $\|$, similarly to the previous case. Formula (7) is divided into two parts that are used exclusively. Both of them impose strict contemporaneity of all the transitions associated with activity $\|$. The first one, on the left side, is only defined for the split activity and states that all the outgoing transitions starting from it, occur at the same time. The second one, on the right side, is similar but it is defined for the join activity, where all the parallel computations must join before proceeding further. It states that all the ingoing transitions, leading to it, occur at the same time.

We assume that when a workflow terminates, it never resumes, by adding to the model formula $end \Rightarrow \mathbf{G}(end)$. The assumption is realistic because business process executions are unique, always have a starting point, where inputs are collected and fed to the process, and possibly terminate by producing an outcome. Infinite repetitions of finite (correct) executions of a workflow are not meaningful for our purpose as our intention aims at modelling infinite executions only when they represent wrong exception handling.

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. Let $s \in S$. To model punctual exception s we introduce the following Formula (9) that forces exception s to be false in the next position of the one where exception s occurs.

$$\bigwedge_{e \in S} (e \Rightarrow \neg \mathbf{X}(e)) \tag{9}$$

Let a be an activity and $catch(a)$ be the set of exceptions that activity a can restores. Non-punctual exceptions may hold continuously over some adjacent positions. When such an exception occurs, at some position, then it holds until an activity a such that $e \in catch(a)$ restores the exception. The following Formula (10) states that if, at the current position, e holds then there is a position in the future where an activity restores it, otherwise it will hold indefinitely. In fact, the consequent of the implication imposes that if $\bigvee_{a:e \in catch(a)} (e\mathbf{U}a)$ holds then $\neg\mathbf{G}(e)$ must hold, that is, e will not hold indefinitely. Conversely, if $\bigvee_{a:e \in catch(a)} (e\mathbf{U}a)$ does not hold then $\neg\mathbf{G}(e)$ must not holds, that is, e will hold indefinitely.

$$\bigwedge_{e \in P} (e \Rightarrow (\neg\mathbf{G}(e) \Leftrightarrow \bigvee_{\substack{a \in A \\ e \in catch(a)}} (e\mathbf{U}a)) \tag{10}$$

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 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 $e \in catch(b)$. The abortion represents a configuration of error that can not be restored, i.e., a loops infinitely or terminates with a system error. Formula (11) states that, if at the current position, activity a holds and exception e occurs and no activity managing e is active, i.e., $e \in catch(b)$, then activity a will never terminate.

$$\bigwedge_{e \in probe(a)} (a \wedge e \wedge \bigwedge_{\substack{b \in A \\ e \in catch(b)}} \neg b) \Rightarrow \mathbf{G}(a) \tag{11}$$

Formula (11) is defined for all activities $a \in A$ of the workflow with a non empty set $probe(b)$.

Following Formulae (12) and (13) define the necessary conditions to have infinite execution. Formula (12) is specific for punctual exceptions. At a certain position, if activity a is active and it never terminates, i.e., $\mathbf{G}(a)$ holds at that position, then 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. This allows modelling the fact that an infinite execution of an activity may be enforced by a different activity that goes into an error state. Moreover, the faulty activity c may start its execution when activity a is already running and the occurrence of the exception that induces the infinite looping error state of c may occur even later its starting position. This explains the **F** in the consequent of the formula that holds when there is a position i, possibly following the position where $\mathbf{G}(a)$ begins to hold, such that, from that position i, there is a position in the past throughout the execution of c where an exception $e \in probe(c)$ occurred and no activity managing e was active.

$$\mathbf{G}(a) \Rightarrow \mathbf{F}\left(\bigvee_{c \in A} (c)\mathbf{S}(c \wedge \bigvee_{e \in probe(c)} (e \wedge \bigwedge_{\substack{b \in A \\ e \in catch(b)}} \neg b))\right) \tag{12}$$

Formula (12) does not apply to non-punctual exception because **S** may hold only in one position (and this is enough to have $\mathbf{G}(a)$) because non-punctual exceptions are not forced to hold indefinitely when no activity of the workflow can eventually handle them. Next Formula (13) remedies the problem and requires that 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 ever catches e.

$$\mathbf{G}(a) \Rightarrow \mathbf{F}\left(\bigvee_{\substack{c \in A \\ e \in probe(c)}} \mathbf{G}\left(c \wedge e \wedge \bigwedge_{\substack{b \in A \\ e \in catch(b)}} \neg b\right)\right) \tag{13}$$

Both formulae (12) and (13) are defined for all activities $a \in A$ appearing in the workflow. Observe that when $probe(c)$, for some $c \in A$, is empty then the second formula of **S**, in Formula (12), and the formula within **F**, in Formula (13), are 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. Next Formula (14) defines the necessary condition so that an internal exception is thrown. Let $throw(a)$ be the set of exceptions that activity a may rise. The first formula states that if exception $e \in S$ holds then there exists an activity a that is active at the same position such that e belong to the set of exceptions which a can raise, i.e., $e \in throw(a)$. The second formula requires that if a permanent exception e holds then there is an activity b and a position in the past where b raised e.

$$\begin{gathered} \bigwedge_{e \in S} (e \Rightarrow \bigvee_{\substack{a \in A \\ e \in throw(a)}} a) \\ \bigwedge_{e \in P} (e \Rightarrow (e)\mathbf{S}(e \wedge \bigvee_{\substack{b \in A \\ e \in throw(b)}} b)). \end{gathered} \tag{14}$$

Next Formula (15) defines the necessary condition for external exceptions. The first part imposes that an external punctual exception e occurs when at least one activity of A is underway. This guarantees that the exception can not happen in correspondence of positions where only transitions hold (and special activities) because transitions are assumed to have no real duration. The second part is similar to the first one and requires that when an external non-punctual exception occurs then there is an activity that is active at some position in the past. In fact, a non-punctual exception may be active even in some positions, between the current one and the one where the exception was generated, where no activity of the workflow is performed. We require however that an exception raises when at least one activity is active (again to avoid meaningless occurrences one temporal unit long, in correspondence of positions where only transitions hold).

$$\begin{gathered} \bigwedge_{e \in S} (e \Rightarrow \bigvee_{a \in A} a) \\ \bigwedge_{e \in P} (e \Rightarrow (e)\mathbf{S}(e \wedge \bigvee_{a \in A} a)). \end{gathered} \tag{15}$$

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

4 EXPERIMENTAL RESULTS

With the introduction of exceptions, designers can represent different scenarios by specifying and verifying functional properties and controlling the behaviour of a workflow when different kinds of abnormal behaviours occur. In this section, some examples of such type of LTL functional properties are defined and analysed (referring to the workflow of Figure 1). The objective is showing the usefulness of the LTL based-semantics approach and how it can be used in practical cases when abnormal behaviours do exist. With the approach shown in this paper, a formal demonstration of correctness does not make

$$\oplus \Rightarrow (\oplus \wedge \neg(t_{yes} \vee t_{no}))\mathbf{U}(t_{yes} \vee t_{no}))$$
$$t_{yes} \Rightarrow \mathbf{Y}(\oplus) \wedge \neg\oplus$$
$$t_{no} \Rightarrow \mathbf{Y}(\oplus) \wedge \neg\oplus$$
$$t_{yes} \Rightarrow \neg t_{no}$$
$$\oplus \Rightarrow \neg\mathbf{Y}(\oplus) \wedge \neg\mathbf{X}(\oplus)$$

$$\|_{start} \Rightarrow (\|_{start} \wedge \neg t_{yes})\mathbf{S}\, t_{yes}$$
$$t_{yes} \Rightarrow \mathbf{X}(\|_{start}) \wedge \neg\|_{start}$$
$$\|_{start} \Rightarrow (\|_{start} \wedge \neg(t_1 \vee t_2))\mathbf{U}(t_2 \vee t_2))$$
$$t_2 \Rightarrow \mathbf{Y}(\|_{start}) \wedge \neg\|_{start}$$
$$t_1 \Rightarrow \mathbf{Y}(\|_{start}) \wedge \neg\|_{start}$$
$$t_1 \Leftrightarrow t_2$$
$$\|_{start} \Rightarrow \neg\mathbf{Y}(\|_{start}) \wedge \neg\mathbf{X}(\|_{start})$$

$$Bill \Rightarrow (Bill \wedge \neg t_1)\mathbf{S}\, t_1$$
$$t_1 \Rightarrow \mathbf{X}(Bill) \wedge \neg Bill$$
$$Bill \Rightarrow (Bill \wedge \neg t_1)\mathbf{U}\, t_3$$
$$t_3 \Rightarrow \mathbf{Y}(Bill) \wedge \neg Bill$$

$$(Bill \wedge tf \wedge \neg Reject_2) \Rightarrow \mathbf{G}(Bill)$$
$$(hf \Rightarrow (\neg\mathbf{G}(hf) \Leftrightarrow false)$$
$$(sf \Rightarrow (\neg\mathbf{G}(sf) \Leftrightarrow (sf)\mathbf{U}(Recovery))$$

$$\mathbf{G}(Bill) \Rightarrow \mathbf{F}((Bill)\mathbf{S}(Bill \wedge (tf \wedge \neg Reject_2))$$
$$\mathbf{G}(Bill) \Rightarrow \mathbf{F}\begin{pmatrix} \mathbf{G}(Bill \wedge hf) \vee \\ \mathbf{G}(Bill \wedge sf \wedge \neg Recovery) \vee \\ \mathbf{G}(Ship \wedge hf) \end{pmatrix}$$
$$Bill \wedge sf \wedge \neg Recovery \Rightarrow \mathbf{G}(Bill)$$

$$tf \Rightarrow \neg\mathbf{X}(tf)$$
$$tf \Rightarrow Ship$$
$$Bill \wedge hf \Rightarrow \mathbf{G}(Bill)$$

Figure 2: Some formulae modelling the portion of workflow in Figure 1. All formulae are conjuncted (symbol $\wedge$ is omitted for brevity) and globally quantified over time by $\mathbf{G}$. Observe that, since the workflow does not have an activity catching exception hf, whenever it occurs at the same time as activity *Bill* then *Bill* enters in an error state, i.e., infinite looping.

sense, since a *semantics* of business workflows is precisely what is being introduced. At the end of the section, we present an analysis of the results in terms of performances of the verification process and satisfiability of properties. For the sake of brevity, we only report a partial translation of the workflow depicted in Figure 1, starting from the second conditional case marked with a symbol ?.

We introduce the following atomic propositions to represent activities: $\oplus$ for modelling the conditional block, $\|_{start}$ and $\|_{end}$ for activities defining starting and ending point of the join block and *Bill* for *Billing*. Similarly, we introduce a proposition for all other activities. We use t_1 to indicate the transition reaching *Billing* which starts from $\|_{start}$ and t_2 to indicate the transition reaching *Shipping* which starts from $\|_{start}$. Finally, we use *sf*, *hf* and *tf* to model exceptions *SoftwareFailure*, *HardwareFailure* and *TransportFailure*. Formulae in the first row of Figure 2 are the translation of the conditional case, the split-join and *Billing* activities.

To validate the LTL model, we exploit the Bounded Satisfiability Checking (BSC) (Pradella et al., 2013) approach. Other approaches are of course possible, like the traditional non-symbolic LTL Model Checking based on the construction of classical Büchi automata (Vardi and Wolper, 1986). The key idea behind BSC is to build a finite representation, of length k, of an infinite ultimately periodic LTL model of the form $\alpha\beta^{\omega}$, where α and β are finite words over the alphabet 2^{AP}. BSC tackles the complexity of checking the satisfiability for LTL formulae by avoiding the unfeasible construction of the Büchi automata. (Bersani et al., 2011) proves that BSC problem for LTL and its extension is complete and that it can be reduced to a decidable Satisfiability Modulo Theory (SMT) problem.

All tests were carried out by using $\mathbb{Z}$ot and Microsot Z3 SMT solver (Microsoft Research, 2009). However, it is also possible to perform analysis with other model checkers that take LTL as their input modelling language. $\mathbb{Z}$ot is a Bounded Model/Satisfiability checker, written in Lisp, that takes as input specifications written in a variety of temporal logics, and determines whether they are satisfiable or not. It performs the checks by encoding temporal logic formulae into the input language of various solvers, in particular SAT and SMT solvers. SAT solvers are capable of taking, as input, formulae written in propositional logic. SMT solvers instead accept formulae written in logics (fragments of First-Order Logic) that are richer than the simple propositional one. Suitable modules, interface $\mathbb{Z}$ot with the underlying SAT or SMT solvers. $\mathbb{Z}$ot scripts, which contain both the model to be analysed and the necessary commands to invoke the desired solver, are a collection of Lisp statements.

Table 3 shows time – in seconds – required by $\mathbb{Z}$ot to verify the set of functional user-defined properties, memory occupation – in MBytes – and the result, i.e. whether the property is satisfied or not. Let S be the formula which translates the model of our workflow in Figure 1. If S is fed to $\mathbb{Z}$ot "as is", $\mathbb{Z}$ot will look for one of its execution; if it does not find one – i.e., if the model is unsatisfiable – then our translation is contradictory, hence it contains some flaws. Now, let P be one of the LTL formulae which model one of the functional properties we want to check on the workflow. If $S \wedge \neg P$ is unsatisfiable, this means that there is no execution that satisfies the workflow, that also satisfies $\neg P$, that is, that violates the property P, so P holds; otherwise this means that there is at least one execution that satisfies both S and $\neg P$; that is, there is at least one execution of the workflow that violates the property, so the property does not hold. If $\mathbb{Z}$ot determines that a formula is satisfiable, then the tool produces an execution that satisfies it that designer can use to check the correctness of the modelling, i.e. a counterexample *trace* that is compatible with the workflow model but that violates the property.

We now provide the LTL formulae of the properties that we have considered for verifing the workflow of Figure1.

$$\mathbf{G}(\neg tf \wedge \neg hf \wedge \neg sf) \Rightarrow \mathbf{F}(end) \tag{16}$$

Formula (16) states that, if no exceptions occur, the workflow must terminate. Feeding the model checker with the negation of this property and the LTL model of the workflow, we can check if **all** possible executions of the workflow reach *end* state. As reported in Table 3, the property holds, since there are no traces which satisfy its negation.

$$(\mathbf{G}(\neg tf \wedge \neg sf) \wedge \mathbf{F}(hf)) \Rightarrow \mathbf{F}(end) \tag{17}$$

$$(\mathbf{G}(\neg hf \wedge \neg sf) \wedge \mathbf{F}(tf)) \Rightarrow \mathbf{F}(end) \tag{18}$$

Formula (18) is similar to formula (17), except that it checks the occurrence of the *TransportFailure* punctual exception, which is thrown by *Shipping* activity and models a shipping problem, such as a truck accident. As reported in Table 3, property (18) does not hold: the counterexample trace shows again that the activities *Billing* and *Shipping* loop forever. Having a look at the workflow, we can note that the problem comes from the exception *tf* being caught inside the wrong activity, *Reject2*, which is not executed in parallel with *Shipping*.

$$\mathbf{F}(\mathbf{G}(Bill)) \Leftrightarrow \mathbf{F}(\mathbf{G}(Ship)) \tag{19}$$

$$\mathbf{F}(\mathbf{G}(Bill)) \Leftrightarrow \mathbf{F}(\mathbf{G}(Arch)) \tag{20}$$

Formulae (19) and (20) model different type of properties respect to the other ones: Formula (19) states that activity *Billing* loops forever if and only if activity *Shipping* do the same, while Formula (20) states the same for the activities *Billing* and *Archiving*. *Ship* and *Arch* are the abbreviations respectively of *Shipping* and *Archiving*. Table 3 shows that the property modeled by Formula (19) holds, while the other one does not hold: in fact, analysing the counterexample trace and the workflow, we can note that if activity *Billing* loops forever, it is not possible to reach activity *Archiving*.

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 $\mathbb{Z}$ot, 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 1, 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 Figure 2 and defined through rules of Section 3).

Table 3: Test results.

Formula	Time (s)	Memory (Mb)	Result
Formula (16)	2.836	16	UNSAT
Formula (17)	3.883	60	SAT
Formula (18)	3.443	60	SAT
Formula (19)	3.843	60	UNSAT
Formula (20)	3.785	60	SAT

Resources (in terms of time and memory) needed to perform the analysis are limited despite the facts that the size of the LTL model is not trivial. In particular, to verify properties like the ones modelled by Formulae (16) and (19), $\mathbb{Z}$ot 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 $\mathbb{Z}$ot, to perform formal verification of non-trivial functional properties, in a limited amount of resources, allowing designer to execute the analysis in an interactive real-time manner.

We left as future work the implementation and testing of more interesting properties, such as hard real-time properties.

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. Our approach is lightweight and allows the reuse of existing tool support. The working assumption is that a lightweight solution would easily fit into processes that are already in place without the need for

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

a radical change of procedures, tools and people's attitudes. The complexity of formalisms and invasiveness of methods have been indeed demonstrated to be one of the major drawback and obstacle for deployment of formal engineering techniques into mundane projects. The case study is a purchase workflow, but the results can be extended to other systems with emphasis on dependability and abnormal behavior management. The treatment of exception handling, and more in general of recovery, is another substantial contribution that has been less frequently investigated with similar techniques and tools.

The workflow patterns here analyzed are limited with respect to a real scenario. Workflow patterns as presented in (van der Aalst et al., 2003) need to be investigated and encoded. 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 (Aalst, 1997).

Future work aims at extending the current translation of workflows by using more expressive logics. In particular, we plan to extend the basic definition of workflow by adding timing constraints on activities and transitions. To model timed workflow we may exploit CLTLoc (Bersani et al., 2013), which is an LTL based logic where atomic formulae are both atomic propositions and constraints over dense clocks. Zero-time modelization 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. (Ferrucci et al., 2012) introduces a new metric temporal logic called *X-TRIO*, which exploits the concepts of *Non-Standard Analysis* (Robinson, 1996). The way to "glue" together CLTLoc with X-TRIO is a promising research strand.

BPMN (OMG, 2011) is the one of the most widespread technique to model business workflows. In (Mazzara and Dragoni, 2012) it has been exploited for workflow design, which has then been implemented in WS-BPEL. In particular, BPMN (as well as UML activity diagrams) includes the concept of partition (modeled as pools and swimlanes), that is essential for business processes modeling and which has not been considered here. This will need to be investigated later.

Finally, runtime evolution in business processes (Baresi et al., 2014) 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 Anirban Bhattacharyya, Alexandr Naumchev, 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

Aalst, W. M. P. v. d. (1997). Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, ICATPN '97, pages 407–426, London, UK, UK. Springer-Verlag.

Attie, P. C. and Singh, M. P. (1993). Specifying and enforcing intertask dependencies. In *In Proceedings of the 19th VLDB Conference*, pages 134–145.

Augusto, J. C., Howard, Y., Gravell, A. M., Ferreira, C., Gruner, S., and Leuschel, M. (2003). Model-based approaches for validating business critical systems. In *STEP*, pages 225–233.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Baresi, L., Guinea, S., and Manna, V. P. L. (2014). Consistent runtime evolution of service-based business processes. In Anna Liu, John Klein, A. T., editor, *Working IEEE/IFIP Conference on Software Architecture (WICSA)*.

Baresi, L., Morzenti, A., Motta, A., and Rossi, M. (2012). A logic-based semantics for the verification of multi-diagram uml models. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8.

Bersani, M. M., Frigeri, A., Rossi, M., and San Pietro, P. (2011). Completeness of the bounded satisfiability problem for constraint LTL. In *Reachability Problems*, volume 6945 of *LNCS*, pages 58–71.

Bersani, M. M., Rossi, M., and San Pietro, P. (2013). A tool for deciding the satisfiability of continuous-time metric temporal logic. In *Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME)*, pages 99–106.

Butler, M. J. and Ferreira, C. (2004). An operational semantics for stac, a language for modelling long-running business transactions. In Nicola, R. D., Ferrari, G. L., and Meredith, G., editors, *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer.

Calzolai, F., Nicola, R. D., Loreti, M., and Tiezzi, F. (2008). Tapas: A tool for the analysis of process algebras. *T. Petri Nets and Other Models of Concurrency*, 1:54–70.

Díaz, M., Valero, V., Macía, H., Mateo, J., and Díaz, G. (2012). 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*.

Dragoni, N. and Mazzara, M. (2009). A formal semantics for the ws-bpel recovery framework - the pi-calculus way. In Laneve, C. and Su, J., editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 92–109. Springer.

Eisentraut, C. and Spieler, D. (2009). Web services and formal methods. chapter Fault, Compensation and Termination in WS-BPEL 2.0 – A Comparative Analysis, pages 107–126. Springer-Verlag, Berlin, Heidelberg.

Ellis, C., Keddara, K., and Rozenberg, G. (1995). Dynamic change within workflow systems. In *Proceedings of Conference on Organizational Computing Systems*, COCS '95, pages 10–21, New York, NY, USA. ACM.

Eshuis, R. and Wieringa, R. (2002). Verification support for workflow design with uml activity graphs.

Ferrucci, L., Mandrioli, D., Morzenti, A., and Rossi, M. (2012). A metric temporal logic for dealing with zero-time transitions. In *Proc. of 19th International Symposium on Temporal Representation and Reasoning*, pages 81–88. IEEE Computer Society.

Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Storey, T., and Weerawaranna, S. (2004). Modeling stateful resources with web services.

Ghezzi, C., Mandrioli, D., and Morzenti, A. (1990). Trio: A logic language for executable specifications of real-time systems. *J. Syst. Softw.*, 12(2):107–123.

Gmehlich, R., Grau, K., Iliasov, A., Jackson, M., Loesch, F., and Mazzara, M. (2013). Towards a formalism-based toolkit for automotive applications. In *Formal Methods in Software Engineering (FormaliSE)*.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274.

Lichtenstein, O., Pnueli, A., and Zuck, L. (1985). The glory of the past. In *Proc. of Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer.

Lucchi, R. and Mazzara, M. (2007). A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96–118.

Mazzara, M. (2009). Deriving specifications of dependable systems: toward a method. In *12th European Workshop on Dependable Computing (EWDC)*.

Mazzara, M. (2011). On methods for the formal specification of fault tolerant systems. In *Proceedings of the 4th International Conference on Dependability (DEPEND 2011)*.

Mazzara, M. and Bhattacharyya, A. (2010). On modelling and analysis of dynamic reconfiguration of dependable real-time systems. *DEPEND, International Conference on Dependability*.

Mazzara, M. and Dragoni, Nicola Zhou, M. (2012). Implementing workflow reconfiguration in ws-bpel. *Journal of Internet Services and Information Security*, 2(1/2):73–92.

Microsoft Research (2009). Z3: An efficient SMT solver. Available at: http://research.microsoft.com/en-us/um/redmond/projects/z3/.

Montesi, F., Guidi, C., and Zavattaro, G. (2014). Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107.

Morzenti, A. and San Pietro, P. (1994). Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(1):56–98.

OASIS (2007). Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

OMG (2005). Unified modeling language 2.0. http://www.omg.org/spec/UML/2.0/.

OMG (2011). Business process model and notation (bpmn). http://www.bpmn.org/.

Pradella, M., Morzenti, A., and San Pietro, P. (2008). Refining real-time system specifications through bounded model- and satisfiability-checking. In *ASE*, pages 119–127.

Pradella, M., Morzenti, A., and San Pietro, P. (2013). Bounded satisfiability checking of metric temporal logic specifications. *ACM Trans. on Soft. Eng. and Meth. (TOSEM)*.

Robinson, A. (1996). *Non-standard analysis*. Princeton University Press.

Romanovsky, A. and Thomas, M., editors (2013). *Industrial Deployment of System Engineering Methods*. Springer.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Vardi, M. Y. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *Proc. of LICS*, pages 332–344.

Vaz, C. and Ferreira, C. (2012). On the analysis of compensation correctness. *J. Log. Algebr. Program.*, 81(5):585–605.

Yamaguchi, M., Yamaguchi, S., and Tanaka, M. (2009). A model checking method of soundness for workflow nets. *IEICE Transactions*, 92-A(11):2723–2731.

SCIENCE AND TECHNOLOGY PUBLICATIONS