*Int. J. Appl. Math. Comput. Sci., 2014, Vol. 24, No. 4, 941–956*
DOI: 10.2478/amcs-2014-0069

# A SYSTEM FOR DEDUCTION–BASED FORMAL VERIFICATION OF WORKFLOW–ORIENTED SOFTWARE MODELS

RADOSŁAW KLIMEK

Department of Applied Computer Science
AGH University of Science and Technology, al. A. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: rklimek@agh.edu.pl

The work concerns formal verification of workflow-oriented software models using the deductive approach. The formal correctness of a model's behaviour is considered. Manually building logical specifications, which are regarded as a set of temporal logic formulas, seems to be a significant obstacle for an inexperienced user when applying the deductive approach. A system, along with its architecture, for deduction-based verification of workflow-oriented models is proposed. The process inference is based on the semantic tableaux method, which has some advantages when compared with traditional deduction strategies. The algorithm for automatic generation of logical specifications is proposed. The generation procedure is based on predefined workflow patterns for BPMN, which is a standard and dominant notation for the modeling of business processes. The main idea behind the approach is to consider patterns, defined in terms of temporal logic, as a kind of (logical) primitives which enable the transformation of models to temporal logic formulas constituting a logical specification. Automation of the generation process is crucial for bridging the gap between the intuitiveness of deductive reasoning and the difficulty of its practical application when logical specifications are built manually. This approach has gone some way towards supporting, hopefully enhancing, our understanding of deduction-based formal verification of workflow-oriented models.

**Keywords:** formal verification, deductive reasoning, temporal logic, semantic tableaux, workflow patterns, logical primitives, generating logical specifications, business models, BPMN.

## 1. Introduction

Software modeling enables better understanding of domain problems and developed systems through goal-oriented abstractions in all phases of software development. Software models require careful verification using mature tools to make sure that the received software products are reliable. Formal methods are intended to systematize and introduce a rigorous approach to software modeling and development by providing precise and unambiguous description mechanisms. A formal approach can be applied at any phase of the software-life cycle (Woodcock *et al.*, 2009), i.e., from the requirements of engineering to verification/validation as well as testing (Hierons *et al.*, 2009). A key issue in formal methods and software engineering is the correctness problem. “Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra, 1972, Corollary). Formal specification and formal verification are two important and closely related parts of the formal approach. Formal *specification* establishes fundamental system properties and invariants. Formal *verification* is the act of proving the correctness of the system. The importance of the formal approach increases and there are many examples of its successful application (e.g., Abrial, 2007).

This work concerns logical inference used for formal verification of software models and practical possibilities of building tools for an appropriate verification procedure. There are two fundamental and well-established approaches to formal verification of systems (Clarke and Wing, 1996). The first one is algorithmically oriented and based on state exploration, and the second one is logically oriented and based on deductive reasoning. Now, the state exploration approach, i.e., model checking (Clarke *et al.*, 1999), wins over the deductive approach due to the significant progress observed during recent years in the field of model checking. However, model checking is a kind of simulation for all reachable paths of computation and constitutes an operational rather than an analytic approach. On the other hand, deductive reasoning plays an important role in the formal approach as a “top-down” and sustainable way of thinking, with reasoning moving from more general facts to more specific ones to reach a certain conclusion which is logically valid. Let us consider some arguments in favor of the deductive approach:

- The main argument is the fact that deductive reasoning enables the analysis of infinite sequences of computations.
- Another argument is naturalness and common use of deductive reasoning in everyday life. It also dominates in scientific works.
- A kind of informal argument is an analogy between natural languages and the logical approach, i.e., the application and knowledge of strict and formal grammatical rules, although not necessary, raises the quality and culture of statements in a natural language, while, by analogy, there is no doubt that applying strict logical rules for reasoning increases the quality of verification procedures and makes them more reliable.

Obtaining logical specifications which are regarded as a set of temporal logic formulas $\{F_1, \ldots, F_n\}$ is an important and crucial issue for any deductive system. If $n$ is large, which is not a rare situation even in the case of an average-size system, then in practice it is not possible to build a logical specification manually, and therefore there is a need to automate this process. However, software models could be organized into some predefined workflow patterns which constitute a kind of primitives that enable the transformation of software models to logical specifications. Automation of this process allows bridging the gap between the naturalness of deductive verification and the difficulty of its practical application. The lack of automation is a significant obstacle to the practical use of logical inference for formal verification. The choice of a deductive system, which is natural and intuitive enough for inexperienced users, is another important aspect. Although the work is not based on a particular method of reasoning, the semantic tableaux method for temporal logic is selected since it is intuitive and has some advantages in comparison with other deduction strategies.

Business models are considered in this work. The significance of business models and their workflows increases in the context of the service oriented architecture (SOA), which is a paradigm that gained important attention within information technology (IT) and business communities. All arguments mentioned in this section are important for research and constitute a challenge to the deductive approach.

### 1.1. Motivation and contribution.

The motivation behind this work is the lack of tools for automatic generation/extraction of logical specifications, regarded as sets of temporal logic formulas, as well as practical use of deduction-based formal verification for workflow-oriented models. Business models expressed in BPMN (business process modeling notation), a standard and dominant notation for business processes, are an important class of systems suitable for the discussed method of deductive reasoning about system properties and seem to be an intellectual challenge that software engineers are faced with when they try to obtain trustworthy and reliable models.

The aim of this work is to provide a conceptual theoretical framework supporting deduction-based formal verification of workflow-oriented models. The contribution is a complete deduction-based system, including its architecture and components, which enables automated and formal verification of business models. The main contribution is an algorithm for the generation of logical specifications providing the method of extracting logical specifications from workflow models. Theoretical possibilities of such automation and the completeness issue for this process are discussed. The application of a non-standard method for deduction which is the semantic tableaux method for temporal logic in the area of business models is another contribution. The proposed approach is characterized by the following advantages: introducing predefined patterns as primitives to logical modeling, and logical patterns once defined, e.g., by a logician or a person with good skills in logic, then widely used, e.g., by analysts and developers with fewer skills in logic.

This work shows theoretical solutions to some problems as outlined above, allowing for future preparation of workable practical solutions. It also opens new research areas as shown in the last section.

### 1.2. Related works.

Workflow technologies are always important for the scientific world (cf. Barker and Van Hemert, 2008), providing a kind of glue for distributed services, for example, service-oriented architectures which constitute a number of loosely coupled and independent services, to obtain more flexible than traditional and strictly coupled applications. Thus, the importance of workflow technologies increases both for scientific and business domains. Dehnert and van der Aalst (2004) present a kind of bridge between business process modeling and workflow specification. The proposed methodology consists of some steps which are designed to provide and include a remedy for intuitive description and informal languages. A proliferation of business process management modeling languages is discussed by Ko *et al.* (2009). Languages and notation are classified into groups of execution, interchange, graphical standards, and diagnostics providing identification and an answer for some common misunderstandings, and also discussing future trends. The dominant language, and *de facto* standard, for business process modeling becomes BPMN, see remarks at the beginning of Section 6. Formal semantics of a subset of BPMN using the process algebra CSP formalism are proposed by Wong and Gibbons (2011). Such a formalism allows comparing BPMN models prepared by developers. A pattern-based method expressing behavioural properties is considered in the work. A translation into a bounded fragment of linear temporal logic is also presented. In the work of Dijkman *et al*. (2008), a mapping from BPMN to Petri nets is proposed to obtain analysis techniques using existing Petri net-based tools, and to enable the static analysis of BPMN models. Leuxner *et al*. (2010) present a formal model for workflows based on a transition system and discuss some algebraic properties. A meta-model for formal specification of functional requirements in business process models, which is not well covered in literature, is proposed by Frece and Juric (2012). Specific extensions to the BPMN semantic and diagram elements are introduced. YAWL (van der Aalst and ter Hofstede, 2005) is a workflow language supporting complex data transformations. It is a graphical language but has a well-defined formal semantics defined as a transition system providing a firm basis for by formal analysis of real-world services.

Business models are also subject to formal verification. Dury *et al*. (2007) discuss business workflows for formal verification using model checking. Eshuis and Wieringa (2004) address the issues of workflows, but they are specified in UML activity diagrams and the goal is to translate diagrams into a format that allows model checking. Some aspects of workflows and temporal logic are considered by Brambilla *et al*. (2005), but the formulas are created manually and formal verification is not discussed very widely. However, these considerations may constitute a kind of starting point for this work. Another important direction of research is verifying business processes using Petri nets (van der Aalst, 2002). In the work of Zha *et al*. (2011), a translation of workflows to Petri nets is proposed to perform analysis using existing tools.

An interesting direction of analysis is $\pi$-calculus, which enables efficient reasoning (e.g., Ma *et al*., 2008), and is designed for business processes and BPEL. The paper by Bryans and Wei (2010) is another work that considers an algorithmic translation from BPMN to Event-B notation, which is based on abstract machine notation, for system modeling and analysis. Morimoto (2008) presents a survey of formal verification for business processes. The author discusses automata, model checking, communicating sequential processes, Petri nets, and Markov networks. All these issues are discussed in the context of business process management and web services. In the general work by Shankar (2009), automated deduction for verification is discussed. There are studied some important issues for symbolic logical reasoning, e.g., satisfiability procedures, automated proof search, and a variety of applications in the case of propositional and fragments of first-order logic. However, even though the work contains a review of symbolic reasoning, modal and temporal logics are omitted. Xu *et al*. (2012) discuss formal verification of workflows. A special language is developed but algorithms refer only to propositional logic. A deductive system for workflow models is proposed by Rasmussen and Brown (2012). Even though for present a solid mathematical framework and some deductive work is done, the theoretical background is like for Petri nets and not formal logic.

In the work of Duan and Ma (2005), a method and a management system for specification workflows by temporal logic based workflow specification model are proposed. Yu and Li (2007) put forward a workflow and a linear temporal logic model. It enables formal verification of workflows and is oriented towards model checking. Rao *et al*. (2008) propose a process model of a workflow management system for which specification of constraints is expressed in linear temporal logic. Another paper that focuses on constraint specification using linear temporal logic is that by Maggi *et al*. (2011). A translation of declarative workflow languages to linear temporal logic and finite automata are considered by Westergaard (2011). Improved algorithms for such a translation process are proposed. In an interesting paper by Taibi and Ngo (2003), design patterns are discussed. A simple language for pattern specification, combining first-order and temporal logic of actions, is proposed.

However, all of the research themes mentioned above are different from the approach presented in this paper, which focuses on formal verification of business processes using deductive-based reasoning with temporal logic. While formal verification is discussed in some of the papers, the application of temporal logic for this purpose is relatively rare. Moreover, the deductive approach used for this domain is quite rare.

### 1.3. Structure.

The rest of the paper is organized as follows. Logical preliminaries, which are temporal logic and logical inference using the semantic tableaux method, are discussed in Section 2. Temporal logic is an established standard for the specification and verification of reactive systems, and the semantic tableaux method is a natural and valuable method of inference. The deduction system and its architecture are proposed in Section 3. The system enables formal verification of business models. It consists of several software components, and some of them may be treated as interchangeable. Workflow patterns are discussed in Section 4. They are treated as (logical) primitives which allow automating the entire process of generating logical specifications. The algorithm for extracting logical specifications is proposed in Section 5. A general example of generating logical specifications is presented in Section 6. The work is summarized and further research is discussed in Section 7.

## 2. Logical preliminaries

Formal logic is a symbolic language that supports the reasoning process with statements to be evaluated as true or false. There is a need for a rigorous and logic-based tool that enables formal reasoning about software models. Natural languages that do not belong to formal logic can be expressive, but they are very imprecise and ambiguous. On the other hand, formal languages, such as formal logic, are not expressive but they are precise, and program properties expressed formally are clearly and commonly understood.

*Temporal logic* (TL) is a branch of symbolic logic that focusses on statements whose evaluations depend on time flows, i.e., it is a formal language which allows expression of temporal properties. Temporal logic is a valuable formalism (e.g., Venema, 2001; Wolter and Wooldridge, 2011), which is widely applied in the area of software engineering for the specification and verification of software models and reactive systems. It is used for system analysis where behaviors of events are of interest. TL exists in many varieties, but the discussion in this paper is limited to *linear temporal logic* (LTL), i.e., the logic for which the time structure is considered linear. This means that each state has exactly one future.

The syntax of LTL is formulated over a countable set of *atomic formulas* $AP = \{p, q, r, \dots\}$ and the set of *temporal operators* $\mathcal{M} = \{\Diamond, \Box\}$. Atomic formulas are those with no propositional sub-structure, or with no sub-formulas, or variables from propositional calculus. Syntax rules allow the definition of syntactically correct, or well-formed, temporal logic formulas.

**Definition 1.** An *LTL formula* is the one which is built using the following rules:

- if $p \in AP$, then $p$ is an LTL formula,
- if $p$ and $q$ are formulas, then $\neg p$, $p \vee q$, $p \wedge q$, $p \Rightarrow q$, $p \Leftrightarrow q$ are LTL formulas
- if $p$ is a formula, then $\boxast p$, where $\boxast \in \mathcal{M}$, is also an LTL formula.

Thus, the whole *LTL alphabet* consists of the following symbols: $AP$, $\mathcal{M}$ and classical logic symbols like $\neg$, $\vee$, $\wedge$, etc. It is relatively easy to introduce other symbols, e.g., parentheses, which are omitted here to simplify the presentation. The $\mathcal{M}$ set consists two fundamental and unary temporal logic operators, where $\Diamond$ means "sometime (or eventually) in the future" and $\Box$ means "always in the future". The operators are dual, i.e., $\neg\Diamond$ is, informally, equal to $\Box\neg$, and $\Diamond$ to $\neg\Box\neg$ and $\Box$ to $\neg\Diamond\neg$. The $\mathcal{M}$ set can be extended to other temporal logic operators. The discussion in the work are focused on the LTL, and particulary on *propositional linear temporal logic* (PLTL). Propositions are statements that could affirm something about members of a class, i.e., workflow activities considered in the work. Thus, propositions $AP$ are used as atomic formulas in Definitions 1 and 4, as well as atomic formulas in the predefined set $P$ in Figs. 3 and 4. However, notions introduced in these figures are described in Section 6.

The semantics of LTL are traditionally defined using the concept of the *Kripke structure*, which is considered a graph, or a path, whose nodes represent the reachable states $w = s_0, s_1, s_2, \dots$ or, in other words, the reachable worlds, and a labeling function which maps each node to a set of atomic formulas $2^{AP}$ that are satisfied in a state. A *valuation* function $\nu(w(i)) \longrightarrow 2^{AP}$, where $i \geqslant 0$, and $w(i)$ means the $i$-th element of the path $w$, allows defining the *satisfaction* $\models$ relation between a path and an LTL formula, e.g., $w \models p$ iff $p \in w(0)$, $w \models \neg p$ iff it is not $p \in w(0)$ and $w \models \Diamond p$ iff $p \in w(i)$, where $i0$, etc. Theorems and laws of LTL can be found in the work of Emerson (1990).

Deductive reasoning is a kind of "top-down" way of thinking that links premises and conclusions. This is a typical and natural procedure in everyday life. Logic and reasoning are cognitive skills. Logical reasoning is the process of applying sound mathematical procedures to given statements to arrive at conclusions. Formal and logic-based inference enables reliable verification of the desired properties. There are some techniques, or proof procedures, which are systematic methods producing proofs in some calculus, or provable statements. In other words, they are decision procedures for logic which enable determining formula satisfiability. There are some examples of deductive reasoning: sequent calculi, resolution-based techniques or semantic tableaux. The resolution technique is based on the observation that every logical formula can be transformed into a conjunctive normal form. The interesting feature of the resolution method is that it has only one inference rule, the resolution rule. On the other hand, the method can be employed to formulas (sub-formulas) in conjunctive normal form. The essence of the procedure is to prove the validity of a sub-formula by establishing that the negation of this sub-formula is unsatisfiable. Another proof procedure is the semantic tableaux method, which is based on the observation that it is not possible for an argument to be true while the conclusion is false. The essence of the procedure is finding counterexamples in branches of a tree after breaking down formulas. Semantic tableaux are global, goal-oriented and "backward", while resolution is local and "forward".

Although the work is not based on any particular method of reasoning, the method of semantic tableaux is presented in a more detailed way. The method of *semantic tableaux*, or the *truth tree*, is well known in classical logic but it can be applied in modal logic (d'Agostino *et al.*, 1999). It is a decision procedure for formula sat-

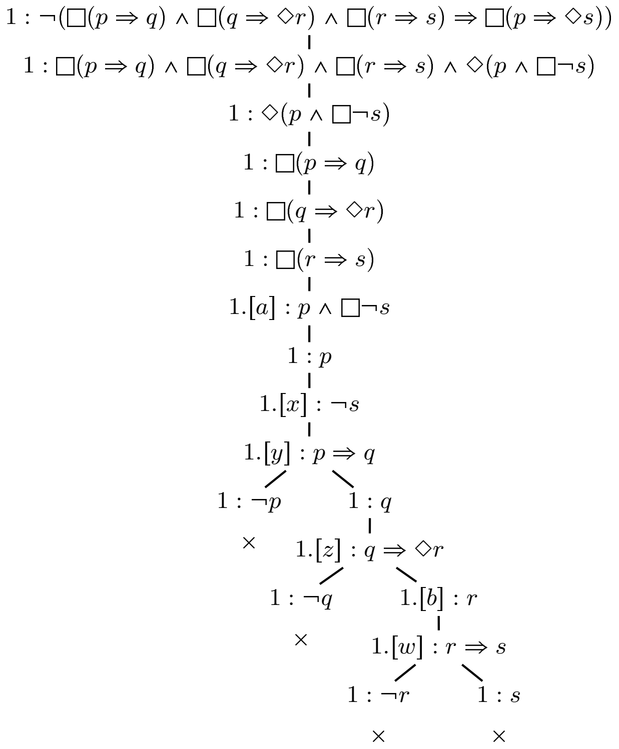
$1 : \neg(\Box(p \Rightarrow q) \wedge \Box(q \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow s) \Rightarrow \Box(p \Rightarrow \Diamond s))$
$1 : \Box(p \Rightarrow q) \wedge \Box(q \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow s) \wedge \Diamond(p \wedge \Box\neg s)$
$1 : \Diamond(p \wedge \Box\neg s)$
$1 : \Box(p \Rightarrow q)$
$1 : \Box(q \Rightarrow \Diamond r)$
$1 : \Box(r \Rightarrow s)$
$1.[a] : p \wedge \Box\neg s$
$1 : p$
$1.[x] : \neg s$
$1.[y] : p \Rightarrow q$
$1 : \neg p$
$1 : q$
×
$1.[z] : q \Rightarrow \Diamond r$
$1 : \neg q$
$1.[b] : r$
×
$1.[w] : r \Rightarrow s$
$1 : \neg r$
$1 : s$
×
×

Fig. 1. Truth tree of the semantic tableaux method.

isfiability checking and represents reasoning by contradiction, i.e., *reductio ad absurdum*. The method is based on formula decomposition using predefined decomposition rules. At each step of the well-defined procedure, formulas become simpler as logical connectives are removed. The tree is *finished* if every (sub-)formula is decomposed and every leaf contains an atomic formula or the negation of an atomic formula. At the end of the decomposition procedure, all branches of the received tree are searched for contradictions. When the branch of the truth tree contains a contradiction, it means that the branch is *closed*. When the branch of the truth tree does not contain a contradiction, it means that the branch is *open*.

When all branches are closed, it means that the tree is closed. In the classical approach, starting from axioms, longer and more complicated formulas are generated and derived. Formulas are getting longer and longer with every step, and only one of them will lead to the verified formula. The method of semantic tableaux is characterized by the reverse strategy. Though we start with a long and complex formula, it becomes less complex and shorter with every step of the decomposition procedure. The open branches of the semantic tree provide information about the source of an error, if one is found, which is an advantage of this method.

**Example 1.** A simple example of an inference tree for a temporal logic formula is shown in Fig. 1. The formula of *minimal temporal logic* (Chellas, 1980; van Benthem, 1993–95) is considered. The adopted decomposition procedure, as well as labeling, refers to the first-order predicate calculus and can be found in the work of Hähnle (1998). Each node contains a (sub-)formula which is either already decomposed or will be subjected to decomposition in the process of building a tree. Each formula is preceded by a label referring to the current world reference. Label "1 :" represents the initial world in which a formula is true. Label "$1.(x)$", where $x$ is a free variable, represents all possible worlds that are consequent of the world 1. On the other hand, label "$1.[p]$", where $p$ is an atomic formula, represents one of the possible worlds, i.e., a successor of the world 1, where formula $p$ is true. Let us note that all branches of the analyzed trees are closed (×). It means there is no valuation that satisfies the root formula. This consequently means that the formula before the negation, i.e., $\Box(p \Rightarrow q) \wedge \Box(q \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow s) \Rightarrow \Box(p \Rightarrow \Diamond s)$, is always satisfied, i.e., the formula is valid. ◆

The semantic tableaux method can be treated as a *decision procedure*, i.e., an algorithm that can produce the Yes/No answer as a response to some important questions. Let $F$ be an examined formula and $\mathcal{T}$ a truth tree built for a formula. Then the following conclusions can be drawn.

**Corollary 1.** *The semantic tableaux method gives answers to the following questions related to the satisfiability problem:*

- *formula $F$ is not satisfied iff the finished $\mathcal{T}(F)$ is closed,*
- *formula $F$ is satisfiable iff the finished $\mathcal{T}(F)$ is open,*
- *formula $F$ is always valid iff finished $\mathcal{T}(\neg F)$ is closed.*

*Proof.* The semantic tableaux method is based on systematic search for models that satisfy a formula. To show that a formula is unsatisfiable, it needs to show that all branches are closed. Hence, if the tree is closed, this means there is no model that satisfies a formula. To show that a formula is satisfiable, it needs to find one open branch. If the tree is open, this means there exists a model that satisfies a formula. If the tree for the negation of a formula is closed, this means there is no model that satisfies a formula, and, as a result of the fact that this is proving by contradiction, it leads to the conclusion that the initial formula is always valid. ■

## 3. Deduction system

The architecture of the proposed inference system is presented and discussed below. The system consists of some independent components and is shown in Fig. 2. A simpler version of the system is shown by Klimek (2012).

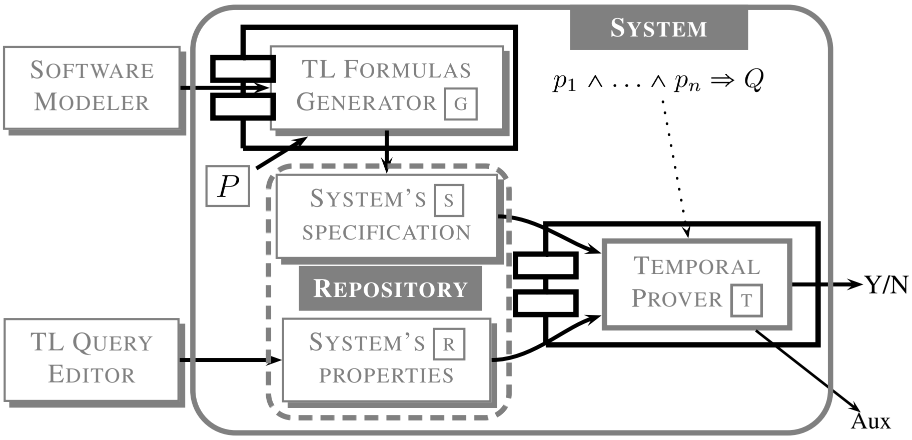

Fig. 2. Architecture of a deduction-based verification system.

The system has two inputs. The data stream with software models to be analyzed is the first input. The approach is based on organizing models into predefined patterns whose temporal properties are once defined, e.g., by a person with good skills in logic, then widely used, e.g., by analysts with fewer skills in logic. The second input is the analyzed property/properties expressed in terms of temporal logic formulas. The easiest way to introduce such formulas is to use a plain text editor and to build them manually. Such a formula, or formulas, is identified by an analyst and describes the expected/desired properties for the investigated software model. Although specifying properties still requires knowledge of temporal logic, formulas for properties are usually much easier to formulate. The output of the whole deductive system is the "Yes/No" answer in response to a new verified property. The whole system can be synthesized informally as $System(Model, Property) \longrightarrow Y/N$. Such a process of inference can be performed many times in response to any new formulas describing the desired and analyzed property. There is another output that is called "Aux". This is a point which enables outputting the auxiliary information depending on the particular method of inference, e.g., open branches in the case of the semantics tableaux method.

The proposed system is based on deductive reasoning and enables examining whether a formula logically "follows" from some statements (formulas).

**Definition 2.** Let $\mathcal{U}$ be a set of formulas and $G$ a formula. If for every model of $\mathcal{U}$, the formula $G$ is satisfied, i.e., the logical value of the formula is equal to the truth, then $G$ is a *logical consequence*, i.e., $\mathcal{U} \models G$.

**Theorem 1.** (Deduction theorem) *Let* $\mathcal{U} = \{F_1, F_2, \ldots, F_n\}$. $\mathcal{U} \models G$ *iff* $\models F \Rightarrow G$, *where* $F_1 \wedge \ldots \wedge F_n \equiv F$.

This is a well-known statement about the equivalence of logical consequence and logical implication. The proof could be found in the work of Kleene (1952). Summing up, the examined formula $G$ is a logical consequence of $F$ iff statement $F \Rightarrow G$ is a *tautology*, i.e., a statement that is always true. It provides the important relation between the notions of logical consequence and validity. The conjunction of all premises leads to the conclusion of the examined formula's validity.

The system works automatically and consists of some important elements. Some of them can be treated as software components/plugins, i.e., they are designed to work as part of a larger system introducing a specific feature, and can be exchanged for one another with similar features if necessary. The first component $\boxed{\text{G}}$ generates logical specifications, i.e., it performs mapping from software models to logical specifications. This process depends also on the predefined workflow property set $P$ which describes the temporal properties for every workflow, and is discussed in the next sections and shown in Fig. 3 and 4. A logical specification is a set of a (usually) large number of temporal logic formulas and is defined in Section 5. The generation of formulas is performed automatically by extracting logical specifications from workflow patterns contained in a workflow model. Formulas regarded as a logical specification are collected in the $\boxed{\text{S}}$ module (data warehouse, i.e., a file or a database) that stores the specification of a system. It can be treated as a conjunction of formulas $p_1 \wedge \ldots \wedge p_n = S$, where $p_i$ is a specification formula generated during the extraction process. The $\boxed{\text{R}}$ module provides the desired and examined properties of the system, as described above, which are expressed in temporal logic. Both the specification of a system and the examined properties constitute an input to the $\boxed{\text{T}}$ component, i.e., *temporal (logic) prover*, which enables the automated reasoning in temporal logic. The input for this component is usually formed in the form of the formula $S \Rightarrow Q$, or, more precisely,

$$p_1 \wedge \ldots \wedge p_n \Rightarrow Q. \quad (1)$$

Since the semantic tableaux method is an indirect proof, after the negation of the formula (1), it is placed at the root of the inference tree and decomposed using well-defined rules of the semantic tableaux method. If the inference tree is closed, this means that the initial formula (1) is true. The output of the $\boxed{\text{T}}$ component and therefore also the output of the whole deductive system are the answer Yes/No in response to any new verified property.

The whole verification procedure can be summarized as follows:

1. automatic generation of system specifications (the $\boxed{\text{G}}$ component), then stored in the $\boxed{\text{S}}$ module,
2. introduction of an examined property of a model (the $\boxed{\text{R}}$ module) as a temporal logic formula (formulas),
3. automatic inference using semantic tableaux (the $\boxed{\text{T}}$ component) for the whole complex formula 1.

Steps from 1 to 3, together or individually, may be processed many times, whenever the specification of the model is changed (step 1) or there is a need for a new inference due to the revised system's specification (steps 2 or 3).

## 4. Workflows as primitives

Workflows regarded as primitives are discussed in this section. *Primitives* are primary or basic units not developed from anything else. In the case of workflows, they can be recognized as a low-level objects that lead to higher-level constructions. In the case of logic, they can be recognized as underived logical elements that lead to more complex logical specifications. A combination of these two primitives is presented below.

Workflows play an important role in computer science and software engineering. Broadly speaking, the *workflow* is a series of tasks, or procedural steps or activities, requiring an input and producing an output, i.e., some added value to the whole activity. In other words, the workflow enables observable progress of the work done by a person, a computer system, or a company. There are many examples of workflows and their notations that influence computer science, and one of them is business models, discussed in Section 6, or activity diagrams of UML (Booch *et al.*, 1999; Pender, 2003). An important feature of workflows is the fact that they are focused on processes rather than documents. This feature is especially important for the approach presented in this paper. One can say that the flow of processes is not disturbed by any data. This gives hope to automate the process of generating logical specifications from workflow-oriented software models which are organized in predefined structures. The main idea is to associate workflows with temporal logic formulas that describe the dynamic aspects of workflows. On the other hand, modeling should be limited to a set of predefined workflows and then models can be developed using only these workflow patterns as discussed in Section 6.

If the last rule of Definition 1 is removed, then the definition of a classical logic formula is obtained. These formulas do not contain modal operators $\mathcal{M}$. Let us present it more formally.

**Definition 3.** The *classical logic*, or *point*, *formula* is a formula which is built using the following rules:

- if $p \in AP$, then $p$ is a point formula,
- if $p$ and $q$ are formulas, then $\neg p$, $p \vee q$, $p \wedge q$ are point formulas.

Point formulas allow describing (logical) circumstances without considering a time flow, i.e., in a point. Only when they are preceded by a temporal operator (e.g., Algorithm 1 or proof in Theorem 2) are they considered in the time context.

Every workflow is linked to logical formulas, both temporal and classical ones. Temporal logic formulas enable describing the internal properties of workflows. Classical logic formulas enable describing workflows from the outside. These aspects are discussed in greater detail below.

**Definition 4.** The *workflow set* of formulas denoted by $\mathit{wrf}(a_1, \ldots, a_n)$, or simply $\mathit{wrf}()$, over atomic formulas $a_1, \ldots, a_n$, is a set of formulas $f_{en}, f_{ex}, f_1, \ldots, f_m$ such that all formulas are syntactically correct, and $f_{en}$ and $f_{ex}$ are point formulas, and $f_1, \ldots, f_m$ are temporal logic formulas, i.e., $\mathit{wrf}() = \{f_{en}, f_{ex}, f_1, \ldots, f_m\}$.

Formulas $a_1, \ldots, a_n$ are arguments of a workflow constituting, informally speaking, its input, i.e., these atomic formulas are used to build both point and temporal formulas of a workflow. Workflow sets are formed in such a way that the first two formulas are classical logic ones and further formulas are LTL ones. The interpretation of such an organization is the following:

1. Classical logic formulas (Definition 3) describe (logical) entry or exit points called *entry formula* $f_{en}$ or *exit formula* $f_{ex}$ of a workflow, i.e., they enable representation of a workflow considered as a whole, in other words, describing the logical circumstances of respectively the start and the termination of the whole workflow execution, or they show which activities of a workflow are executed first or last, respectively, cf. the predefined workflow property set $P$ given in Section 6. Thus, these formulas should not be confused with the well-known precondition or postcondition,

respectively. Let $wrf().f_{\text{en}}$ and $wrf().f_{\text{ex}}$ be entry and exit formulas, respectively, from a workflow set $wrf()$; if it does not lead to ambiguity, then formulas are written shortly as $f_{\text{en}}$ and $f_{\text{ex}}$.

2. Temporal logic formulas (Definition 1) describe the internal behavior of the workflow $f_1, \ldots, f_m$, showing dynamic aspects of a workflow pattern. Every property can be characterized using a liveness property and a safety property (cf. Alpern and Schneider, 1985), thus the aim is to obtain a decomposition expressed in terms of temporal logic formulas.

Summing up, point formulas allow consideration of a workflow as a whole, i.e., from the outside point of view, while temporal formulas show the internal behavior of a workflow.

Some restrictions on atomic formulas $a_1, \ldots, a_n$ of the workflow set $wrf()$ in Definition 4, due to the partial order, are introduced.

**Definition 5.** The set of atomic formulas is divided into three subsets which are pairwise disjointed and the following rules must be valid:

1. the first subset, which contains at least one element, consists of *entry arguments*, and all of these arguments, and no other, form the $f_{en}$ formula,

2. the second subset, which may be empty, consists of *ordinary arguments*,

3. the third subset, which contains at least one element, consists of *exit arguments*, and all these arguments, and no other, form the $f_{ex}$ formula.

**Example 2.** Let us discuss some examples of workflow sets for hypothetical workflow patterns: $W1(a,b) = \{a, b, a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$, $W2(a,b,c) = \{a, b \vee c, a \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$, and $W3(a,b,c,d) = \{a \vee b, d, a \Rightarrow \Diamond c, b \Rightarrow \Diamond c, \Box\Diamond c \Rightarrow \Box\Diamond d, \Box\neg((a \vee b) \wedge (c \vee d))\}$. In the case of $W1$ and $W2$ the $a$ proposition is a (logical) starting point for the whole workflow, i.e., it means that when $a$ is satisfied then workflows are started. $W1$ probably refers to a workflow for a sequence of two tasks $a \Rightarrow \Diamond b$ (liveness), and therefore it is also not possible (safety) that these two task are satisfied simultaneously $\Box\neg(a \wedge b)$. $W2$ probably shows a parallel split of two task, and therefore the $b \vee c$ formula describes the fact that when the workflow ends then $b$ or $c$ are satisfied. In the case of $W3$, the disjunction $a \vee b$ is a (logical) starting point. The $d$ task is always the last activity of the workflow. The set of formulas for $W3$ is a more complex and interesting case. It describes a reactive and fair service (liveness) $\Box\Diamond c \Rightarrow \Box\Diamond d$, i.e., when $c$ is satisfied then always follow $d$. The service is ready to work after the initiation of the whole workflow $(a \vee b)$, and after starting a service (liveness) $a \Rightarrow \Diamond c$ or $b \Rightarrow \Diamond c$. It is mandatory to ensure the safety of the workflow, i.e., the start formulas and service formulas cannot be satisfied at the same time $\Box\neg((a \vee b) \wedge (c \vee d))$. ♦

**Corollary 2.** *The definition of the workflow set $wrf()$ and further remarks lead to the following valid statements:*

- *none of the ordinary arguments of a workflow set are included either in the $f_{en}$ or the $f_{ex}$ formula,*

- *every workflow contains, and its logical formulas describe, the structure that consists of at least two activities (or tasks).*

*Proof.* The proof is relatively simple and, for example, the second statements follows from the fact that the entire set of atomic formulas $a_1, \ldots, a_n$ as arguments for a workflow set must contain at least two arguments which constitute activities (tasks). ■

The whole software model comprising workflows can be quite complex including nesting workflows, and this is why there is a need to define symbolic notation which enables to represent any potentially complex structure.

**Definition 6.** The *workflow expression* $W$ is a structure built using the following rules:

- every workflow set $wrf(a_1, \ldots, a_i, \ldots, a_n)$, where every $a_i$ is an atomic formula, is a workflow expression,

- every $wrf(A_1, \ldots, A_i, \ldots, A_m)$, where every $A_i$ is either
  - an atomic formula $a_k$, where $k > 0$, or
  - a workflow set $wrf(a_j)$, where $j > 0$ and every $a_j$ is an atomic formula, or
  - a workflow expression $wrf(A_j)$, where $j > 0$

  is also a workflow expression.

Every $a_i$ (lower case letters) represents only atomic formulas. Every $A_i$ (upper case letters) represents either atomic formulas or workflows. These rules allow defining an arbitrary complex workflow expression. Due to the partial order relation described above and Corollary 2, it should be noted that, in regard to the workflow expression, there is a similar valid restriction on the number of arguments, i.e., there are at least two arguments for every workflow expression, which is informally shown through the way of indexing for a workflow expression that takes values $i, j = 1, 2, \ldots$.

The notion of aggregated entry/exit formulas is introduced which is a result of nested and complex workflows, as well as the need to transfer, informally speaking, the logical signal to all start/termination points of a nested workflow.

**Definition 7.** Let $w^c$ for a workflow expression $w$ with the upper index $c = e$ (or $x$, respectively) be the *aggregated entry formula* (or the *aggregated exit formula*, respectively) when the aggregated formula is calculated using the following (recursive) rules:

1. if there is no workflow itself in the place of any atomic formula/argument which syntactically belongs to the $f_{en}$ formula (or the $f_{ex}$ formula, respectively) $w$, then $w^e$ is equal to $f_{en}$ ($w^x$ is equal to $f_{ex}$, respectively),
2. if there is a workflow, say $t()$, in place of any atomic argument, say $r$, which syntactically belongs to the $f_{en}$ formula (or the $f_{ex}$ formula, respectively) of $w$, then $r$ is replaced by $t^e$ (or $t^x$, respectively) for every such case.

These rules allow defining aggregated point formulas for an arbitrary complex workflow expression.

**Example 3.** Let us supplement Definitions 6 and 7 by some examples. Let $\Sigma$ be a *predefined workflow set*, e.g.,

$$\Sigma = \{Seq, Concur, Branch, Loop\}, \tag{2}$$

properties whose might be described and stored in the $P$ set, cf. Figs. 3 and 4, modeling sequence, concurrency, branching and iteration, respectively. However, they are defined here in a different (simpler) way compared the $P$ set, i.e., through direct introduction of all necessary formulas. Thus, $Seq(a, b) = \{a, b, a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$, $Concur(a, b, c) = \{a, b \vee c, a \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$, $Branch(a, b, c) = \{a, b \vee c, a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$, and $Loop(a, b, c, d) = \{a, d, a \Rightarrow (\Diamond b \wedge \neg\Diamond d) \vee (\neg\Diamond b \wedge \Diamond d), b \Rightarrow \Diamond c, c \Rightarrow (\Diamond b \wedge \neg\Diamond d) \vee (\neg\Diamond b \wedge \Diamond d)\}$.

The meaning of *Seq* seems obvious. The *Concur* and *Branch* workflows model concurrency and branching, respectively, for two activities $b$ and $c$, which are preceded by another activity $a$. The *Loop* workflow models a while-cycle case that has exactly one input activity $a$ and exactly one output activity $d$, which are located before and after, respectively, the main loop. The sequence of two activities $b$ and $c$ constitutes the entire body of a loop, where $b$ is a main instruction of the body, and $c$ is an incrementation for the body. Formal definitions in terms of temporal logic formulas for these patterns are proposed above. If it is necessary to model concurrency and branching without a preceding activity, then it can be obtained using the provided patterns *Concur* or *Branch*, and assuming that the preceding activity $a$ may be, informally, the *null task*, that is, the execution of which consumes zero time. ♦

Workflow expressions may represent an arbitrary structure, and an example of this is $Seq(a, Seq(Concur(b, c, d), Branch(e, f, g)))$ whose meaning is intuitive, i.e., it might shows the sequence that leads to another sequence of concurrent execution of some activities and then the branch by selecting an activity.

**Example 4.** Examples of aggregated formulas are given as follows. For $w = Seq(a, b)$ the formulas are $w^e = a$ and $w^x = b$ (step 1). For $w = Concur(a, b, Seq(c, d))$ the formulas are $w^e = a$ (step 1) and $w^x = b \vee d$ (step 1 gives "$b\vee$" and step 2 gives $Seq^x$ ="$d$" which is aggregated to "$b \vee d$"). For $w = Concur(a, b, Concur(c, d, e))$ the formula is $w^x = b \vee (d \vee e)$ (step 1 gives "$b\vee$" and step 2 gives $Concur^x$ ="$d \vee e$" which is aggregated to "$b \vee (d \vee e)$"). For $w = Concur(a, Concur(b, c, d), Concur(e, f, g))$ the formula is $w^x = (c \vee d) \vee (f \vee g)$ (step 2 gives $Concur^x = c \vee d$ and $f \vee g$, and after aggregation "$(c \vee d) \vee (f \vee g)$" is obtained). ♦

An important property of workflow expressions is their internal and nested structure. Parentheses are the best illustration for it. Suppose that all instances of "$wrf($" and "$, wrf($", where "$wrf$" is the symbol of an arbitrary workflow, were substituted by "(". Then, for example, the above workflow expression gives the parenthesis structure $(a(b, c, d)(e, f, g))$. This in turn leads to the following theorem.

**Theorem 2.** *For any workflow expression and for any two workflow patterns $wrf_i()$ and $wrf_j()$, where $i \neq j$, only one of the following three situations holds:*

1. *$wrf_i()$ and $wrf_j()$ are completely disjoint,*
2. *$wrf_i()$ is completely contained in $wrf_j()$,*
3. *$wrf_j()$ is completely contained in $wrf_i()$.*

*Proof.* Firstly, let us note that Definition 6 is recursive. For the first case of the definition, a simple pattern with atomic formulas is considered, and it is consistent with the theorem in an obvious way. For the second case, two subcases are considered. For the first subcase, if the argument is an atomic formula, then it is clear that no parentheses are introduced. For the second subcase, recursive application of the rule introduces a new pattern with correctly paired parenthesis, and this subcase guarantees the complete contain (nesting) of patterns. If correctly paired patterns are introduced/substituted in place of different arguments of a pattern, then it guarantees the disjointedness of the paired patterns. ■

## 5. Generating specifications

The process of generating logical specifications is described below. Informally speaking, *logical specification* is a counterpart of formal generalization understood as an act of taking some facts and making broader statements, i.e., formal derivation of a general statement from a particular one. In this work, logical specification is expressed as a set of temporal logic formulas. These formulas are generated from workflow expressions using predefined workflows as logical primitives. Let us define it formally and then present an algorithm.

**Definition 8.** The *logical specification* $L$ is a set of temporal logic formulas derived from a workflow expression $W$ and a predefined set $P$ using the algorithm $\Pi$, i.e., $L(W) = \{f_i : i > 0 \wedge f_i \in \Pi(W, P)\}$, where $f_i$ is an LTL formula.

Generating logical specifications is not a simple union of predefined formula collections resulting from patterns used in a workflow expression. The generation algorithm $\Pi$ is given as Algorithm 1. The generation process has two inputs. The first one is a workflow expression $W$, which is a kind of variable, i.e., it varies for every workflow model. The second one is a workflow property set $P$, which is a kind of constant since it is predefined and fixed containing definitions of workflows in terms of temporal logic formulas. More detailed information about this set including its examples is given in Section 6. The output of the generation algorithm is a logical specification understood as a set of temporal logic formulas.

Let $wrf()^T$ represent a set of all temporal formulas extracted from a workflow set (i.e., without point formulas). Algorithm 1 refers to similar ideas in the works of Klimek (2013) and Klimek *et al.* (2013); however, the case considered here is more general and not focused on specific patterns. All workflows of the workflow expression are processed one by one and the algorithm always halts. All parentheses are paired. Let $p4(h, p2(d, p1(a, b, c), e), p3(f, g))$ be a hypothetical workflow expression, where $p1$, $p2$, $p3$, and $p4$ are workflow patterns. Pattern $p3$ has two arguments, and other patterns have three arguments. Considering the loop in the line 2 of Algorithm 1, the processing order of patterns is the following: $p4$, $p2$, $p1$, and $p3$, where $p4$ and $p2$ are processed in lines 6–12, and $p1$ and $p3$ are processed in lines 3–5.

**Algorithm 1** Generating logical specifications ($\Pi$).

```
Input: Logical expression $W_L$ (non-empty), predefined
    set $P$ (non-empty)
Output: Logical specification $L$
 1: $L := \varnothing$                                  ▷ initiating specification
 2: for every workflow $wrf()$ of $W_L$ from left to right do
 3:     if all arguments of $wrf()$ are atomic then
 4:         $L := L \cup wrf()^T$
 5:     end if
 6:     if any argument of $wrf()$ is a workflow itself then
 7:         for every such an argument, say $r()$, substitute
 8:         disjunction of its aggregated entry and exit
 9:         formulas in all places where the argument
10:         occurs in the $wrf()$ temporal formulas, i.e.,
11:         $L := L \cup ((wrf()^T) \leftarrow$ “$r()^e \vee r()^x$”)
12:     end if
13: end for
```

**Example 5.** Considering the predefined workflow set given by the formula 2 and its definitions of workflows, let us supplement Algorithm 1 by some examples. The example for lines 3–5: $Seq(a, b)$ gives $L = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$ and $Branch(a, b, c)$ gives $L = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$. The example for lines 6–12: $Concur(Seq(a, b), c, d)$ leads to $L = \{(a \vee b) \Rightarrow \Diamond c \wedge \Diamond d, \Box\neg((a \vee b) \wedge (c \vee d))\} \cup \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$. Other examples are shown in Section 6. ♦

Algorithm 1 comprises two main parts. In the first one, lines 3–5, logical specifications are rewritten from a predefined set, cf. Figs. 3 and 4, without any modification and summed with the resulting specification. In the second part, lines 6–12, the workflow formulas $f_{en}$ and $f_{ex}$ are taken into account since they allow consideration of the nested workflow as a whole, i.e., without analyzing its internal behavior, which is itself and separately taken into account in the first part. Consideration of both $f_{en}$ and $f_{ex}$ seems a bit redundant for a single workflow but, on the other hand, informally speaking, these two formulas have equal rights to represent a workflow, and line 11 contains their disjunction which is substituted, and then modified temporal formulas are summed with the resulting specification.

Algorithm 1 allows automating the process of generating logical specifications. Logical expressions are translated into logical specifications, which are expressed in terms of temporal logic formulas. Logical expressions can be arbitrarily complex and nested. Moreover, the list of predefined patterns can be arbitrarily, that is, in any way and at any time extended by new patterns. The only requirement is to define behaviour, cf. Figs. 3 and 4, for new patterns in terms of temporal logic prior to their first use. Thus, the general idea that logical patterns are once defined and then widely used is satisfied.

The completeness problem is a fundamental issue for logical systems and constitutes their key requirement in many fields. Informally speaking, completeness is in some opposition to the fragmentation. In other words, completeness means having all elements and lacking nothing while fragmentation means not having all elements and lacking something. Generally speaking, an object, or a set of objects, is complete if nothing more needs to be added to it. In formal logic systems, *completeness* means that if a formula is valid it can be proven (Gries and Schneider, 1993, p. 128). In algorithms, it refers to the ability of finding a solution if one exists.

This paper discusses both predefined logical specifications and the algorithm for generating logical specifications using predefined specifications. This requires an integrated perspective towards completeness by considering two aspects:

1. completeness of possessed logical specifications, which is contained in a predefined set of patterns, cf. Figs. 3 and 4, and

2. completeness of the generation algorithm, i.e., Algorithm 1.

Firstly, the completeness of the predefined set $P$ is considered. The set consists of logical specifications that refer to particular patterns or, in other words, every pattern is defined in terms of temporal logic formulas. These specifications should be examined, one by one, for compliance with the relevant logical properties. However, as has already been said in Section 1.1 (motivation), logical patterns are predefined by a logician or a person with good skills in logic for further use by an ordinary analyst or a developer. This leads to the conclusion that the logician is responsible for proving the correctness and logical properties of predefined specifications, and some decision procedures, cf. Corollary 1, might be helpful for this process.

Predefined logical specifications constitute an input for the generation of Algorithm 1. Thus, it is reasonable to question whether the algorithm preserves the completeness when generating the resulting logical specification, i.e., obtained as an output of Algorithm 1.

**Definition 9.** The algorithm of generating logical specification is *relatively complete* if it preserves the completeness of the generated logical specification or, in other words, if it does not introduce itself incompleteness to the output logical specifications with respect to predefined input specifications.

**Theorem 3.** *Suppose that a predefined workflow set is non-empty, and every pattern of the $P$ set is non-empty, and every two patterns have disjointed sets of atomic formulas, and the workflow expression $W$ is non-empty. Then the logical specification obtained for Algorithm 1 is relatively complete.*

*Proof.* Let us note that, due to the parentheses in Theorem 2, patterns are nested entirely/completely, i.e., it is not possible to obtain a partial nesting that might provide undesirable crossing of patterns. Furthermore, every two patterns contain disjointed sets of atomic formulas. Every system can be described in terms of safety and liveness properties/formulas (Alpern and Schneider, 1985). If a predefined logical specification is complete, then incompleteness cannot be introduced while generating the output logical specification when using liveness formulas. The most general form for liveness is formula $P \Rightarrow \Diamond Q$. Let us consider two cases for Algorithm 1.

*Case 1 (lines 3–5).* Specifications are only rewritten from a predefined set, cf. Figs. 3 and 4. Then if the input specification is complete, the completeness property is preserved.

*Case 2 (lines 6–12).* The entry and exit formulas are considered. They are generalization for a nested pattern and allow bypassing/skipping its internal behaviour. They enable considering both the beginning and the end of a workflow. Let us note that for any workflow pattern $w()$, due to Corollary 2, there is always satisfied $\Box\neg(w().f_{en} \wedge w().f_{ex})$. On the other hand, there is also valid $\Box(w().f_{en} \Rightarrow \Diamond w().f_{ex})$. However, due to the nature of entry and exit points, they are both either satisfied or not satisfied, mapping a kind of logical propagation, which leads to the third, and additional, formula $\Box(\neg w().f_{en} \Rightarrow \neg\Diamond w().f_{ex})$.

Completeness refers to the reachability of all formulas and properties of a logical specification. Let us consider the sequence of two workflows $Seq(g(), h())$ for the predefined set expressed by the formula 2 and further definitions. Let $g().f_{en} \equiv g_e$, $g().f_{ex} \equiv g_x$, $h().f_{en} \equiv h_e$, and $h().f_{ex} \equiv h_x$. Let us return to the three formulas introduced above. After considering them in the context of workflows $g()$ and $h()$, they are gathered as premises. Now, due to the ordinary liveness formula $a \Rightarrow \Diamond b$, e.g., definition of $Seq$ for the formula (2) where $a$ refers to $g()$ and $b$ refers to $h()$ for the mentioned $Seq(g(), h())$, and the substitution in the 11-th line of Algorithm 1, the following formula that is added to the premises set is obtained: $\Box((g_e \vee g_x) \Rightarrow \Diamond(h_e \vee h_x)))$. Formula $\Box(g_x \Rightarrow \Diamond h_e)$ is a requirement that expresses the demand to pass from one exit point directly to the next entry point that allows covering all properties/formulas from the beginning of the next workflow. Gathering all premises and the demand, the resulting formula is

$$\begin{aligned}(\Box(g_e \Rightarrow \Diamond g_x) \wedge \Box(\neg g_e \Rightarrow \neg\Diamond g_x) \wedge \\ \Box\neg(g_e \wedge g_x) \wedge \Box(h_e \Rightarrow \Diamond h_x) \wedge \\ \Box(\neg h_e \Rightarrow \neg\Diamond h_x) \wedge \Box\neg(h_e \wedge h_x) \wedge \\ \Box((g_e \vee g_x) \Rightarrow \Diamond(h_e \vee h_x))) \\ \Rightarrow \Box(g_x \Rightarrow \Diamond h_e).\end{aligned} \tag{3}$$

While analyzing the above formula using the semantic tableaux method, the obtained truth tree, similar to the small tree from Fig. 1, contains many hundreds of nodes, and is closed, which means that the formula (3) is always satisfied (tautology).

Considering both cases is sufficient for the entire algorithm. ■

## 6. Model analysis and verification

A method of formal verification of business models is discussed in this section. The method follows from the approach provided in this work. Firstly, business systems are modeled using predefined workflow patterns, i.e., processes associated with logical patterns. In other words, workflow patterns are predefined in terms of temporal logic formulas, and then logical specifications are automatically generated using Algorithm 1. The introduced deduction-based verification system allows performing verification of business models in a formal way.

Workflow patterns are crucial for the approach introduced in this work as they lead to the automation of the logical specification generation process. Informally speaking, a pattern is a distinctive formation created and used as an archetype. Creating and using patterns promotes software reuse, which is always a kind of *idée fixe* in software engineering. Riehle and Züllighoven (1996) described *patterns* as "the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts". Patterns might constitute a kind of primitives which enable the mapping of workflow patterns to logical specifications. *Business process modeling notation* is a standard and dominant graphical notation (e.g., OMG, 2011), for the modeling of business processes.

The primary goal of BPMN is to provide notation that is understandable by all business users, from business analysts to technical developers and business people who will manage and monitor these processes. An important part of BPMN is 21 patterns which are introduced by van der Aalst *et al.* (2003). Gradually building in complexity, process patterns were broken down into six categories, and the *Basic Control Flow Patterns* category is considered in this work. The proposed method of automatic extraction of logical specifications is based on the assumption that the whole business model is built using only the well-known workflow patterns of BPMN. This assumption is fundamental to the consideration of the work and is not a restriction since it enables receiving correct and well-composed business models.

Let the predefined workflow set of patterns be $\Sigma = \{\mathit{Sequence}, \mathit{ParallelSplit}, \mathit{Synchronization}, \mathit{ExclusiveChoice}, \mathit{SimpleMerge}\}$. This set might be extended using other patterns described by van der Aalst *et al.* (2003). Definitions of all potentially used workflow patterns are expressed in terms of temporal logic and stored in the set $P$, which is predefined and fixed. It is assumed that the defining process is performed by a person with good skills in logic. The process should contain a discussion and proofs of the logical properties for every pattern. Furthermore, the defining process is performed once, and then logical primitives can be widely used. An example of such a predefined workflow set $P$ is shown in Fig. 3. The way to define formally the individual workflow patterns, the type of the formulas used, is itself an interesting problem. However, it is not discussed here exactly, and should be the subject of research for all patterns in a separate work, cf. remarks in the Section 7, where the syntax of the presentation language shown in the figure is expected to be defined formally. Now, it is presented informally in the following way. Most elements of the $P$ set, i.e., two temporal logic operators, classical logic operators, are not in doubt in understanding. The slash allows placing more than one formula in a single line. Here $f_1$, $f_2$, etc. are atomic formulas and constitute a kind of formal arguments for a pattern. Every pattern has two point formulas which are located at the beginning of the set describing the start and the final, respectively, logical conditions/circumstances of the execution of a pattern. The content of the $P$ set is shown as a plain ASCII text to illustrate its participation in the real processing, cf. Fig. 2. Though the above set contains a relatively small number of patterns, a justification for this is only to present a general idea for pattern-oriented generation of logical specifications, and there is no difficulty with defining a set of workflow formulas for any other process patterns, as well as for the 21 patterns mentioned above (van der Aalst *et al.*, 2003; White, 2004).

```
                        /* version 25.10.2013
/* Basic Control Patterns
Sequence(f1,f2):
f1
f2
[](f1 => <>f2) / [](~f1 => ~<>f2)
[]~(f1 & f2)
ParallelSplit(f1,f2,f3):
f1
f2 | f3
[](f1 => <>f2 & <>f3) / [](~f1 => ~<>f2 & ~<>f3)
[]~(f1&(f2|f3))
Synchronization(f1,f2,f3):
f1 | f2
f3
[](f1 & f2 => <>f3) / []( ~(f1 & f2) => ~<>f3)
[]~((f1|f2)&f3)
ExclusiveChoice(f1,f2,f3):
f1
f2 | f3
[](f1 => (<>f2 & ~<>f3)|(~<>f2 & <>f3))
[](~f1 => ~<>f2 & ~<>f3)
[]~(f1&(f2|f3)) / []~(f2 & f3)
SimpleMerge(f1,f2,f3):
f1 | f2
f3
[](f1|f2 => <>f3) / [](~(f1|f2) => ~<>f3)
[]~(f1|f2) / []~((f1|f2)&f3)

/* ..... [other] Business Patterns
```

Fig. 3. Sample predefined set $P$.

Although formal definitions for all patterns exceed the volume and goal of this paper, the predefined set of workflows $P$ is to be extended by the $\mathit{ArbitraryCycles}$ pattern (cf. White, 2004, pp. 11–12; Fig. 4), which is perhaps the most complex process pattern, and $\Sigma := \Sigma \cup \{\mathit{ArbitraryCycles}\}$. The pattern represents cycles that have more than one entry or exit points. There are no special restrictions on the types of loops used. A new notation linked with tested loop conditions is introduced. If $exp$ is

```
ArbitraryCycles(Alfa,Beta,Chi,A,B,C,D,F,E,G):
Alfa
E | G
/* first loop (Alfa)
[](x(Alfa) & c(Alfa) => <>B & ~<>A)
[](x(Alfa) & ~c(Alfa) => <>A & ~<>B)
[](~x(Alfa) => ~<>A & ~<>B)
[]~(x(Alfa)&(A|B)) / []~(A|B|C)
[](A => <>C) / [](~A => ~<>C)
[](B | C => <>D) / [](~(B | C) => ~<>D)
[]~((B|C)&D)
/* second loop (Beta)
[](D => <>x(Beta)) / [](~D => ~<>x(Beta))
[]~(D&x(Beta))
[](x(Beta) & c(Beta) => <>E & ~<>F)
[](x(Beta) & ~c(Beta) => ~<>E & <>F)
[](~x(Beta) => ~<>E & ~<>F)
[]~(x(Beta)&(E|F)) / []~(E|F)
/* towards outside (Chi,G)
[](F => <>x(Chi)) / [](~F => ~<>x(Chi))
[]~(F&x(Chi))
[](x(Chi) & c(Chi) => <>G & ~<>C)
[](x(Chi) & ~c(Chi) => ~<>G & <>C)
[]~(x(Chi)&(G|C)) / []~(G|C)
```

Fig. 4. *ArbitraryCycles* pattern for a predefined set $P$.

a condition (logical expression) to be tested, which is associated with a certain activity, then $c(exp)$ means that the logical expression $exp$ is evaluated and is true. $x(exp)$ means that the activity associated with the expression $exp$ is satisfied, i.e., the activity is executed (from the rising/positive edge to the falling/negative edge). Here $exp$ can be evaluated only when $x(exp)$ is satisfied, and the following sentence is valid: $x(exp) \wedge (c(exp) \vee \neg c(exp))$; otherwise, when $\neg x(exp)$, the value of the $c(exp)$ expression is undefined. An example of an extended part of the $P$ set is shown in Fig. 4. *Alfa*, *Beta*, *Chi*, $A$, $B$, $C$, etc. are formal arguments for the workflow pattern, where the first three arguments refer to some conditions. The workflow has one entry argument, two exit arguments, and six ordinary arguments. Temporal formulas of the workflow set describe both safety and liveness properties for the pattern.

**Example 6.** Let us consider a simple yet illustrative example to present the approach of the work. The example is somewhat abstract, but the main purpose is to demonstrate the key idea which is the deployment of predefined patterns for modeling and generating logical specifications, and formal verification of business models. Let us suppose that workflow expression $W$ is

$$\begin{gathered} \mathit{Sequence}(\mathit{ExclusiveChoice}(\mathit{Sequence}(a,b),\\ \mathit{Sequence}(c,d), \mathit{Sequence}(\mathit{ParallelSplit}(e,f,g),\\ \mathit{Synchronization}(h,i,j))), \mathit{SimpleMerge}(k,l,m)) \end{gathered}$$

The logical specification $L$ is built in the following steps. At the beginning, the specification is $L = \varnothing$. The patterns are processed in the following order: *Sequence*, *ExclusiveChoice*, *Sequence*, *Sequence*, *Sequence*, *ParallelSplit*, *Synchronization*, and *SimpleMerge*.

The following sub-sets are generated: the first *Sequence* gives $L_1 = \{\Box(a \vee (d \vee j)) \Rightarrow \Diamond((k \vee l) \vee m), \Box(\neg(a \vee (d \vee j)) \Rightarrow \neg\Diamond((k \vee l) \vee m), \Box\neg((a \vee (d \vee j)) \wedge ((k \vee l) \vee m))\}$, *ExclusiveChoice* gives $L_2 = \{\Box((a \vee b) \Rightarrow (\Diamond(c \vee d) \wedge \neg\Diamond(e \vee j)) \vee (\neg\Diamond(c \vee d) \wedge \Diamond(e \vee j))), \Box(\neg(a \vee b) \Rightarrow \neg\Diamond(c \vee d) \wedge \neg\Diamond(e \vee j)), \Box\neg((a \vee b) \wedge ((c \vee d) \vee (e \vee j))), \Box\neg((c \vee d) \wedge (e \vee j))\}$, the second *Sequence* gives $L_3 = \{\Box(a \Rightarrow \Diamond b), \Box(\neg a \Rightarrow \neg\Diamond b), \Box\neg(a \wedge b)\}$, the third *Sequence* gives $L_4 = \{\Box(c \Rightarrow \Diamond d), \Box(\neg c \Rightarrow \neg\Diamond d), \Box\neg(c \wedge d)\}$, the fourth *Sequence* gives $L_5 = \{\Box((e \vee (f \vee g)) \Rightarrow \Diamond((h \vee i) \vee j)), \Box(\neg(e \vee (f \vee g)) \Rightarrow \neg\Diamond((h \vee i) \vee j)), \Box\neg((e \vee (f \vee g)) \wedge ((h \vee i) \vee j))\}$, *ParallelSplit* gives $L_6 = \{\Box(e \Rightarrow \Diamond f \wedge \Diamond g), \Box(\neg e \Rightarrow \neg\Diamond f \wedge \neg\Diamond g), \Box\neg(e \wedge (f \vee g))\}$, *Synchronization* gives $L_7 = \{\Box(h \wedge i \Rightarrow \Diamond j), \Box(\neg(h \wedge i) \Rightarrow \neg\Diamond j), \Box\neg((h \vee i) \wedge j)\}$, and *SimpleMerge* gives $L_8 = \{\Box(k \vee l \Rightarrow \Diamond m), \Box(\neg(k \vee l) \Rightarrow \neg\Diamond m), \Box\neg(k \vee l), \Box\neg((k \vee l) \wedge m)\}$. Thus, the resulting specification is $L = L_1 \cup \cdots \cup L_8$ and contains the formulas

$$\begin{aligned}
L = \{&\Box(a \vee (d \vee j)) \Rightarrow \Diamond((k \vee l) \vee m),\\
&\Box(\neg(a \vee (d \vee j)) \Rightarrow \neg\Diamond((k \vee l) \vee m),\\
&\Box\neg((a \vee (d \vee j)) \wedge ((k \vee l) \vee m)),\\
&\Box((a \vee b) \Rightarrow (\Diamond(c \vee d) \wedge \neg\Diamond(e \vee j)) \vee\\
&\quad (\neg\Diamond(c \vee d) \wedge \Diamond(e \vee j))),\\
&\Box(\neg(a \vee b) \Rightarrow \neg\Diamond(c \vee d) \wedge \neg\Diamond(e \vee j)),\\
&\Box\neg((a \vee b) \wedge ((c \vee d) \vee (e \vee j))),\\
&\Box\neg((c \vee d) \wedge (e \vee j)), \Box(a \Rightarrow \Diamond b),\\
&\Box(\neg a \Rightarrow \neg\Diamond b), \Box\neg(a \wedge b),\\
&\Box(c \Rightarrow \Diamond d), \Box(\neg c \Rightarrow \neg\Diamond d), \Box\neg(c \wedge d),\\
&\Box((e \vee (f \vee g)) \Rightarrow \Diamond((h \vee i) \vee j)),\\
&\Box(\neg(e \vee (f \vee g)) \Rightarrow \neg\Diamond((h \vee i) \vee j)),\\
&\Box\neg((e \vee (f \vee g)) \wedge ((h \vee i) \vee j)),\\
&\Box(e \Rightarrow \Diamond f \wedge \Diamond g), \Box(\neg e \Rightarrow \neg\Diamond f \wedge \neg\Diamond g),\\
&\Box\neg(e \wedge (f \vee g)), \Box(h \wedge i \Rightarrow \Diamond j),\\
&\Box(\neg(h \wedge i) \Rightarrow \neg\Diamond j), \Box\neg((h \vee i) \wedge j),\\
&\Box(k \vee l \Rightarrow \Diamond m), \Box(\neg(k \vee l) \Rightarrow \neg\Diamond m),\\
&\Box\neg(k \vee l), \Box\neg((k \vee l) \wedge m)\} \qquad (4)
\end{aligned}$$

The resulting logical specification can be used for formal verification of a system. ♦

Liveness and safety are a standard taxonomy of properties when specifying and verifying systems. *Liveness* means that the computational process achieves its goals, i.e., something good eventually happens, or its counterexample has a prefix extended to infinity. *Safety* means that the computational process avoids undesirable situations, i.e., nothing bad ever happens, or its counterexample has a finite prefix. The liveness property for the model can be

$$\Box(b \Rightarrow \Diamond j), \tag{5}$$

which means that always if $b$ is satisfied then sometime in the future the $j$ activity is satisfied. The safety property for the examined model can be

$$\Box\neg(c \wedge g), \tag{6}$$

what means that it never happens that $c$ and $g$ are satisfied in the same time.

The whole formula to be analyzed using the semantic tableaux method for the property expressed by the formula (5) is

$$(\Box(a \vee (d \vee j)) \Rightarrow \Diamond((k \vee l) \vee m) \wedge \ldots \wedge \\ \Box\neg((k \vee l) \wedge m)) \Rightarrow \Box(b \Rightarrow \Diamond j). \tag{7}$$

The formula (4) represents the output of the $\boxed{\text{G}}$ component in Fig. 2. The formula (7) provides a combined input for the $\boxed{\text{T}}$ component in Fig. 2. When considering the property expressed by the formula (6), then the whole formula is constructed in a similar way as

$$(\Box(a \vee (d \vee j)) \Rightarrow \Diamond((k \vee l) \vee m) \wedge \ldots \wedge \\ \Box\neg((k \vee l) \wedge m)) \Rightarrow (\Box\neg(c \wedge g)). \tag{8}$$

The full reasoning tree for both cases contains hundreds of nodes. Formulas are valid and the examined properties are satisfied in the model considered.

The prover is an important component of the architecture for the deduction-based system shown in Fig. 2. It enables automation of the inferencing process and formal verification of the developed models. Reasoning engines are more available, especially in recent years when a number of provers for modal logics has become accessible, (cf. Schmidt, 2014). Selection of an appropriate existing prover, or building one's own, constitutes a separate task that exceeds the scope and main objectives of this work, cf. also the concluding remarks in the last Section.

## 7. Conclusions

A method of pattern-oriented automatic generation of logical specifications for business models expressed in BPMN is proposed. Logical specifications are regarded as a set of temporal logic formulas and obtaining it is a crucial aspect in the case of practical use of deduction-based formal verification. An algorithm as a method for automatic generation of logical specifications from predefined logical patterns/primitives is proposed. The architecture of a deduction-based system for formal verification of business models is presented.

The generating method enables a kind of scaling up, migration from small problems to real-world problems in the sense that they are having more and more nesting patterns. This gives hope for practical use in problems of any size. The proposed approach introduces the concept of logical primitives, workflow patterns predefined in terms of temporal logic formulas. They might be once well-defined and could be widely used by an inexperienced user. The proposed system enables formal verification of business models using temporal logic and semantic tableaux provers. The advantage of the method is to provide an innovative concept for process verification, which might be done for any given business model created using BPMN. The aim of this work has been to provide a conceptual theoretical framework to prepare workable solutions for deduction-based formal verification of workflow-oriented models.

Future research should extend the results in some directions, e.g., other logical properties of the approach should be explored. The fundamental issue for the approach is to define formally all workflow patterns (van der Aalst *et al.*, 2003) in terms of temporal logic formulas to provide temporal logic-based semantics for workflows. The literature review argues that there is a lack of such comprehensive and formal definitions. Definitions proposed in Section 6, i.e., the predefined set $P$, might be considered the beginning of such work. Another important issue is detailed analysis of the existing and available provers (Schmidt, 2014) which could be useful for the approach and applied as a prover component (Fig. 2). Future works may also include both the implementation of the generation component (Fig. 2) and its own temporal logic prover (Section 2) using the semantic tableaux method. Implementation works in both of these cases are carried out and are relatively advanced. This should result in being CASE software providing industrial-proof tools, that is, implementing another part of formal methods, hope promising, in industrial practice.

## Acknowledgment

The author wishes to thank the anonymous reviewers for their valuable comments that helped to improve the work.

## References

Abrial, J.-R. (2007). Formal methods: Theory becoming practice, *Journal of Universal Computer Science* **13**(5): 619–628.

Alpern, B. and Schneider, F.B. (1985). Defining liveness, *Information Processing Letters* **21**(4): 181–185.

Barker, A. and Van Hemert, J. (2008). Scientific workflow: A survey and research directions, *in* R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Wasniewski (Eds.), *Proceedings of the 7th International Conference Parallel Processing and Applied Mathematics, PPAM 2008, Gdańsk, Poland, 9–12 September 2007*, Lecture Notes in Computer Science, Vol. 4967, Springer Verlag, Berlin, pp. 746–753.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999). *The Unified Modeling Language Reference Manual*, Addison Wesley, Redwood City, CA.

Brambilla, M., Deutsch, A., Sui, L. and Vianu, V. (2005). The role of visual tools in a web application design and verification framework: A visual notation for LTL formulae, *in* D. Lowe and M. Gaedke (Eds.), *Proceedings of the 5th International Conference on Web Engineering ICWE 2005, July 27–29, 2005, Sydney, Australia*, Lecture Notes in Computer Science, Vol. 3579, Springer Verlag, Berlin, pp. 557–568.

Bryans, J.W. and Wei, W. (2010). Formal analysis of BPMN models using Event-B, *in* S. Kowalewski and M. Roveri (Eds), *15th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2010, 20–21 September 2010, Antwerp, Belgium*, Lecture Notes in Computer Science, Vol. 6371, Springer Verlag, Berlin, pp. 33–49.

Chellas, B.F. (1980). *Modal Logic*, Cambridge University Press, Cambridge.

Clarke, E., Grumberg, O. and Peled, D. (1999). *Model Checking*, MIT Press, Cambridge, MA.

Clarke, E. and Wing, J. (1996). Formal methods: State of the art and future directions, *ACM Computing Surveys* **28**(4): 626–643.

d'Agostino, M., Gabbay, D., Hähnle, R. and Posegga, J. (1999). *Handbook of Tableau Methods*, Kluwer Academic Publishers, Hingham, MA.

Dehnert, J. and van der Aalst, W.M.P. (2004). Bridging the gap between business models and workflow specifications, *International Journal of Cooperative Information Systems* **13**(03): 289–332.

Dijkman, R.M., Dumas, M. and Ouyang, C. (2008). Semantics and analysis of business process models in BPMN, *Information and Software Technology* **50**(12): 1281–1294.

Dijkstra, E.W. (1972). *Structured Programming*, Academic Press, London, pp. 1–82.

Duan, Y. and Ma, H. (2005). Modeling flexible workflow based on temporal logic, *in* W. Shen, A.E. James, K.-M. Chao, M. Younas, Z. Lin and J.-P.A. Barthès (Eds.), *Proceedings of the 9th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2005, 24–26 May 2005, Coventry, UK*, Vol. 1, IEEE Computer Society, Washington, DC, pp. 508–513.

Dury, A., Boroday, S., Petrenko, A. and Lotz, V. (2007). Formal verification of business workflows and role based access control systems, *in* L. Peñalver, O.A. Dini, J. Mulholland and O. Nieto-Taladriz (Eds.), *Proceedings of the 1st International Conference on Emerging Security Information, Systems and Technologies, SecurWare 2007, October 14–20, 2007, Valencia, Spain*, IEEE Computer Society, Washington, DC, pp. 201–210.

Emerson, E. (1990). *Handbook of Theoretical Computer Science*, Vol. B, MIT Press, Cambridge, MA, pp. 995–1072.

Eshuis, R. and Wieringa, R. (2004). Tool support for verifying UML activity diagrams, *IEEE Transactions on Software Engineering* **30**(7): 437–447.

Frece, A. and Juric, M.B. (2012). Modeling functional requirements for configurable content- and context-aware dynamic service selection in business process models, *Journal of Visual Languages & Computing* **23**(4): 223–247.

Gries, D. and Schneider, F.B. (1993). *A Logical Approach to Discrete Math*, Springer Verlag, New York, NY.

Hähnle, R. (1998). Tableau-based theorem proving, *10th European Summer School on Logic, Language and Information, ESSLLI 1998, Saarbrücken, Germany*.

Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R. and Zedan, H. (2009). Using formal specifications to support testing, *ACM Computing Survey* **41**(2): 9:1–9:76.

Kleene, S.C. (1952). *Introduction to Metamathematics*, Bibliotheca Mathematica, North-Holland, Amsterdam.

Klimek, R. (2012). Towards formal and deduction-based analysis of business models for SOA processes, *in* J. Filipe and A. Fred (Eds.), *Proceedings of the 4th International Conference on Agents and Artificial Intelligence, ICAART 2012, 6–8 February 2012, Vilamoura, Algarve, Portugal*, Vol. 2, SciTePress, Lisboa, pp. 325–330.

Klimek, R. (2013). From extraction of logical specifications to deduction-based formal verification of requirements models, *in* R.M. Hierons, M.G. Merayo and M. Bravetti (Eds.), *Proceedings of the 11th International Conference on Software Engineering and Formal Methods, SEFM 2013, 25–27 September 2013, Madrid, Spain*, Lecture Notes in Computer Science, Vol. 8137, Springer Verlag, Berlin, pp. 61–75.

Klimek, R., Faber, Ł. and Kisiel-Dorohinicki, M. (2013). Verifying data integration agents with deduction-based models, *Proceedings of the Federated Conference on Computer Science and Information Systems, FedCSIS 2013, Kraków, Poland*, pp. 1049–1055.

Ko, R.K., Lee, S.S. and Lee, E.W. (2009). Business process management (BPM) standards: A survey, *Business Process Management Journal* **15**(5): 744–791.

Leuxner, C., Sitou, W. and Spanfelner, B. (2010). A formal model for work flows, *in* J.L. Fiadeiro, S. Gnesi and A. Maggiolo-Schettini (Eds.), *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13–18 September 2010*, IEEE Computer Society, Washington, DC, pp. 135–144.

Ma, S., Zhang, L. and He, J. (2008). Towards formalization and verification of unified business process model based on pi calculus, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20–22 August 2008, Prague, Czech Republic*, IEEE Computer Society, Washington, DC, pp. 93–101.

Maggi, F.M., Montali, M., Westergaard, M. and van der Aalst, W.M.P. (2011). Monitoring business constraints with linear temporal logic: An approach based on colored automata, *Proceedings of the 9th International Conference* *on Business Process Management, BPM 2011, 30 August–2 September 2011, Clermont-Ferrand, France*, Lecture Notes in Computer Science, Vol. 6896, Springer Verlag, Berlin, pp. 132–147.

Morimoto, S. (2008). A survey of formal verification for business process modeling, *in* M. Bubak, G.D. van Albada, J. Dongarra and P.M.A. Sloot (Eds.), *Proceedings of the 8th International Conference on Computational Science, ICCS 2008, June 23–25, 2008, Kraków, Poland, Part II*, Lecture Notes in Computer Science, Vol. 5102, Springer Verlag, Berlin, pp. 514–522.

OMG (2011). Business process model and notation (BPMN) version 2.0, *Technical report*, Object Management Group, Needham, MA, `http://www.omg.org/spec/bpmn/2.0`.

Pender, T. (2003). *UML Bible*, John Wiley & Sons, New York, NY.

Rao, M.R., Hildebrandt, T.T. and Tth, J.B. (2008). The resultmaker online consultant: From declarative workflow management in practice to LTL, *in* M. van Sinderen, J.P.A. Almeida, L.F. Pires and M. Steen (Eds.), *12th Enterprise Distributed Object Computing Conference Workshops, EDOCW 2008, 16 September 2008, Munich, Germany*, IEEE Computer Society, Washington, DC, pp. 135–142.

Rasmussen, R. and Brown, R. (2012). A deductive system for proving workflow models from operational procedures, *Future Generation Computer Systems* **28**(5): 732–742.

Riehle, D. and Züllighoven, H. (1996). Understanding and using patterns in software development, *Theory and Practice of Object Systems* **2**(1): 3–13.

Schmidt, R. (2014). Accessible theorem provers, `http://www.cs.man.ac.uk/~schmidt/tools/`.

Shankar, N. (2009). Automated deduction for verification, *ACM Computing Surveys* **41**(4): 20:1–20:56.

Taibi, T. and Ngo, D.C.L. (2003). Modeling of distributed objects computing design pattern combinations using a formal specification language, *International Journal of Applied Mathematics and Computer Science* **13**(2): 239–253.

van Benthem, J. (1993–95). *Handbook of Logic in Artificial Intelligence and Logic Programming*, 4, Oxford University Press, New York, NY, pp. 241–350.

van der Aalst, W.M.P. (2002). Making work flow: On the application of Petri nets to business process management, *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets, ICATPN 2002, 24–30 June 2002, Adelaide, Australia*, Lecture Notes in Computer Science, Vol. 2360, Springer Verlag, Berlin, pp. 1–12.

van der Aalst, W.M.P. and ter Hofstede, A.H.M. (2005). YAWL: Yet another workflow language, *Information Systems* **30**(4): 245–275.

van der Aalst, W.M., ter Hofstede, A., Kiepusewski, B. and Barros, A. (2003). Workflow patterns, *Distributed and Parallel Databases* **4**(1): 5–51.

Venema, Y. (2001). *Blackwell Guide to Philosophical Logic*, Basil Blackwell Publishers, Oxford, pp. 203–223.

Westergaard, M. (2011). Better algorithms for analyzing and enacting declarative workflow languages using LTL, *Proceedings of the 9th International Conference Business Process Management, BPM 2011, 30 August–2 September 2011, Clermont-Ferrand, France*, Lecture Notes in Computer Science, Vol. 6896, Springer Verlag, Berlin, pp. 83–98.

White, S.A. (2004). Process modeling notations and workflow patterns, *BPTrends*, pp. 1–25, `http://www.omg.org/bpmn/Documents/Notations_and_Workflow_Patterns.pdf`.

Wolter, F. and Wooldridge, M. (2011). Temporal and dynamic logic, *Journal of Indian Council of Philosophical Research* **XXVII**(1): 249–276.

Wong, P.Y. and Gibbons, J. (2011). Property specifications for workflow modelling, *Science of Computer Programming* **76**(10): 942–967.

Woodcock, J., Larsen, P.G., Bicarregui, J. and Fitzgerald, J. (2009). Formal methods: Practice and experience, *ACM Computing Survey* **41**(4): 19:1–19:36.

Xu, L.D., Viriyasitavat, W., Ruchikachorn, P. and Martin, A. (2012). Using propositional logic for requirements verification of service workflow, *IEEE Transactions on Industrial Informatics* **8**(3): 639–646.

Yu, Y. and Li, X. (2007). A workflow model with temporal logic constraints and its automated verification, *Proceedings of the 6th International Conference on Grid and Cooperative Computing, GCC 2007, August 16–18, 2007, Urumchi, Xinjiang, China*, IEEE Computer Society, Washington, DC, pp. 681–684.

Zha, H., van der Aalst, W.M.P., Wang, J., Wen, L. and Sun, J. (2011). Verifying workflow processes: A transformation-based approach, *Software & Systems Modeling* **10**(2): 253–264.

**Radosław Klimek** graduated from the AGH University of Science and Technology (AGH UST) in Kraków, Poland, and received a Ph.D. in computer science in 1998. Currently he is with the Department of Applied Computer Science at the AGH UST. His research interest includes formal methods and formal verification, temporal and modal logics, classical and non-classical methods of reasoning, software engineering and development, algorithms, programming languages, pervasive and ubiquitous systems, as well as computing and reasoning in context-aware systems.

Received: 30 November 2013
Revised: 11 April 2014