# From Extraction of Logical Specifications to Deduction-Based Formal Verification of Requirements Models

Radosław Klimek

AGH University of Science and Technology,
al. A. Mickiewicza 30, 30-059 Krakow, Poland
rklimek@agh.edu.pl

**Abstract.** The work relates to formal verification of requirements models using deductive reasoning. Elicitation of requirements has significant impact on the entire software development process. Therefore, formal verification of requirements models may influence software cost and reliability in a positive way. However, logical specifications, considered as sets of temporal logic formulas, are difficult to specify manually by inexperienced users and this fact can be regarded as a significant obstacle to practical use of deduction-based verification tools. A method of building requirements models, including their logical specifications, is presented step by step. Requirements models are built using some UML diagrams, i.e. use case diagrams, use case scenarios, and activity diagrams. Organizing activity diagrams into predefined workflow patterns enables automated extraction of logical specifications. The crucial aspect of the presented approach is integrating the requirements engineering phase and the automatic generation of logical specifications. Formal verification of requirements models is based on the deductive approach using the semantic tableaux reasoning method. A simple yet illustrative example of development and verification of a requirements model is provided.

**Keywords:** requirements engineering, formal verification, deductive reasoning, use case diagrams, use case scenarios, activity diagrams, workflows patterns, temporal logic, logical specifications, semantic tableaux method.

## 1 Introduction

Software modeling enables better understanding of the domain problem and of the system under development. Requirements engineering is an important part of software modeling. Requirements elicitation should lead into a coherent structure of requirements and have significant impact on software quality and costs. Thinking of requirements must precede the analysis, design, and code generation acts. Requirements models are descriptions of delivered services in the context of operational constraints. Identifying software requirements of the system-as-is, gathering requirements and formulation of requirements by users allows defects to be identified earlier in a life cycle.

UML, i.e. the Unified Modeling Language [16], which is ubiquitous in the software industry can be a powerful tool for the requirements engineering process. Use cases are central to UML since they strongly affect other aspects of the modeled system and, after joining the activity diagrams, may constitute a good vehicle to discover and write down requirements. Temporal logic is a well established formalism which allows to describe properties of reactive systems, also visualized in UML. The semantic tableaux method, which is a proof formalization for assessing logical satisfiability, seems intuitive and may be regarded as goal-based formal reasoning.

Formal methods enable precise formulation of important artifacts arising during software development and help eliminate ambiguity. There are two well established approaches to formal reasoning and system verification [5]. The first is based on state exploration ("model checking") and the second is based on deductive reasoning. However, model checking is an operational rather than analytic approach.Deductive inference enables the analysis of infinite computation sequences. On the other hand, one important problem of the deductive approach is the lack of automatic methods for obtaining logical specifications considered as sets of temporal logic formulas. The need to build logical specifications manually can be recognized as a major obstacle to untrained users. Thus, the automation of this process seems particularly important. Moreover, application of the formal approach to the entire requirements engineering phase may increase the maturity of requirements models.

**Motivation, Contributions and Related Works.** The motivation for this work is the lack of tools and practical applications of deductive methods for formal verification of requirements models. Another motivation, which is associated with the previous one, is the lack of tools for automatic extraction of logical specifications from software models. However, requirements models built using use case and activity diagrams seem to be suitable for such an extraction process. All of the aforementioned aspects of the formal approach seem to be an intellectual challenge in software engineering.

The contribution of the work is a method for building formal requirements models, including their logical specification, based on some UML diagrams. A complete deduction-based system which enables the automated and formal verification of requirements models is proposed. Another contribution is a method for automating the generation of logical specifications. The generation algorithm for selected workflow patterns is presented. The reasoning process is performed using the semantic tableaux method for temporal logic. The proposed method is characterized by the following advantages: introducing workflow patterns as primitives to requirements engineering and logical modeling, scaling up to real-world problems, and logical patterns once they are defined and widely used. All these factors are discussed in the work and summarized in the last section.

There are some fundamental works on requirements engineering, c.f. the work by van Lamsweerde [15], which is a comprehensive study of many fundamentals of this area. The work by Chakraborty et al. [4] discusses some social processes associated with requirements engineering. In the work by Rauf et al. [17], a method for extracting logical structures from documents is presented. In the work by Kazhamiakin [11], a method based on formal verification of requirements using temporal logic and model checking approach is proposed, and a case study is discussed. Hurlbut [10] provides a very detailed survey of selected issues concerning use cases. The informal character of scenario documentation implies several difficulties in reasoning about the system behavior and validating the consistency between the diagrams and scenario descriptions. Barrett et al. [2] presents the transition of use cases to finite state machines. Zhao and Duan [20] shows formal analysis of use cases; however, the Petri Nets formalism is used. Eshuis and Wieringa [8] addresses the issues of activity diagram workflows but the goal is to translate diagrams into a format that allows model checking. There are some other works in the area of the formal approach and UML-based requirements engineering but there is a lack of works for deduction-based formal verification with temporal logic and the semantic tableaux method for UML-based requirements models. The work [12] is a (very) preliminary version of this one, and the differences include: a lower level of formalization, differences in predefined workflow patterns, more casual algorithm for generating logical specifications, and the lack of an accurate case study.

## 2 Methodology

The outline of the procedure and guidelines used for the construction of a requirements model, as it is understood in the work, is briefly discussed below. It constitutes a kind of methodology and its subsequent steps are presented in Fig. 1. The entire procedure can be summarized in the following items:

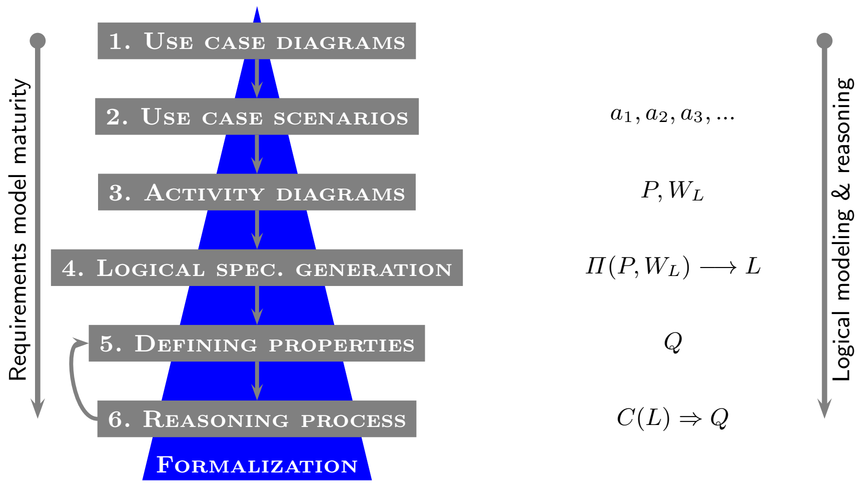


**Fig. 1.** Software requirements modeling and deduction-based verification

1. use case diagrams – use case modeling to understand functions and goals of a system;
2. use case scenarios – identifying and extracting atomic activities;
3. activity diagrams – modeling workflows using predefined patterns;
4. automatic generation of logical specifications from requirements models;
5. manual definition of the desired model properties;
6. formal verification of a desired property using the semantic tableaux method.

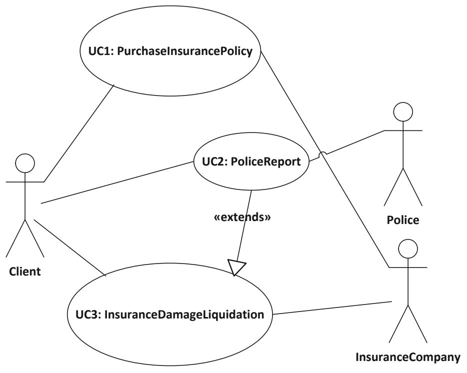
All steps are shown on the left side of Fig. 1. The first three steps involve the requirements modeling phase but the last three steps involve generation of logical specification and analysis of requirements model properties. The loop between the last two steps refers to a process of both identifying and verifying more and more new properties of the examined model. Some symbols and notation resulting from the introduced formalization are on the right side of Fig. 1 and they are discussed in further sections of the work. Generally, it leads, step by step, from an abstract view of a system to more and more detailed and reliable and, finally, verified requirements models.

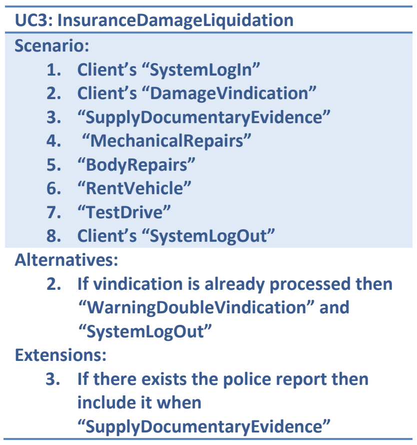
## 3 Use Cases and Identification of Activities

Defining use cases and scenarios is important not only to understand the functionalities of a system but also to identify elementary activities. The activities play an important role when building logical specifications, i.e. the logical specification is modeled over atomic activities. The *use case diagram* consists of actors and use cases. *Actors* are objects which interact with a system and create the system's environment, thus providing interaction with the system. *Use cases* are services and functionalities which are used by actors. The use case diagrams are a rather descriptive technique and do not refer to any details of the system implementation [18].

Let us present it more formally. In the initial phase of a system modeling, use case diagrams $UCD_1, ..., UCD_n$ are built. Every $UCD_i$ diagram contains some use cases $UC_1, ..., UC_m$ which describe the desired functionality of a system. A typical and sample use case diagram is shown in Fig. 2. It consists of three actors and three use cases, $UC_1$, $UC_2$ and $UC_3$, modeling a system of car insurance and damages liquidation. The diagram seems to be intuitive and is not discussed in detail.

Use cases are commonly used for capturing requirements through *scenarios* which are brief narratives that describe the expected use of a system. A scenario is a possible sequence of steps which enables the achievement of a particular goal resulting from the functionality of a use case. Every use case $UC_i$ has its own scenario. From the point of view of the approach presented here, scenarios play an additional important role, which is identification of atomic activities used to build individual scenario steps. An *activity* is the smallest unit of computation. Thus, every scenario contains some activities $a_1, ..., a_n$. The set of *atomic activities* $AA$ contains all activities identified and defined for all scenarios. The most valuable situation is when the entire use case scenario involves identified activities and the narrative does not dominate and is limited to model behavior, which is later formally shown in activity diagrams.





**UC3: InsuranceDamageLiquidation**

**Scenario:**

1. Client's "SystemLogIn"
2. Client's "DamageVindication"
3. "SupplyDocumentaryEvidence"
4. "MechanicalRepairs"
5. "BodyRepairs"
6. "RentVehicle"
7. "TestDrive"
8. Client's "SystemLogOut"

**Alternatives:**

2. If vindication is already processed then "WarningDoubleVindication" and "SystemLogOut"

**Extensions:**

3. If there exists the police report then include it when "SupplyDocumentaryEvidence"

**Fig. 2.** A sample use case diagram $UCD$ "CarInsuranceLiquidatingDamages" (left) and a scenario for the use case $UC_3$ "InsuranceDamageLiquidation" (right)

A sample scenario for the use case $UC_3$, i.e. "InsuranceDamageLiquidation", is shown in Fig. 2. It contains some atomic activities which are identified when preparing the scenario. The alternative and extension points are defined. The "DamageVindication" activity represents the registration process in the insurer system and the start of the process of recovery damages. While the car repair process is carried out ("MechanicalRepairs" and "BodyRepairs"), the client can hire a replacement vehicle ("RentVehicle"). The level of formalization presented here, i.e. when discussing use cases and their scenarios, is intentionally not very high. This assumption seems realistic since this is an initial phase of requirements modeling. Dynamic aspects of activities are to be modeled strictly when developing activity diagrams, c.f. section 5.

## 4 Logical Background

*Temporal logic* TL introduces symbolism for reasoning about truth and falsity of formulas throughout the flow of time, taking the changes of their valuations into consideration. Two basic and unary operators are $\Diamond$ for "sometime (or eventually) in the future" and $\Box$ for "always in the future"; these are dual operators. Temporal logic exists in many varieties; however, these considerations are limited to the *linear-time temporal logic* or *linear temporal logic* LTL. Linear temporal logic refers to infinite sequences of computations and attention is focused on the *propositional linear time logic* PLTL. These sequences refer to the Kripke structure which defines the semantics of TL, i.e. a syntactically correct formula can be satisfied by an infinite sequence of truth evaluations over a set of *atomic propositions* AP. It should be pointed out that atomic propositions are identical to atomic activities defined in section 3, i.e. $AA = AP$. The basic issues related to the TL syntax and semantics are discussed in many works, e.g. [7,19].

The properties of the time structure are fundamental to a logic. Of particular significance is the *smallest*, or *minimal*, *temporal logic*, e.g. [3], also known as temporal logic of class K. The minimal temporal logic is an extension of the classical propositional calculus of the axiom $\Box(\Phi \Rightarrow \Psi) \Rightarrow (\Box\Phi \Rightarrow \Box\Psi)$ and the inference rule $\vdash \Phi \Longrightarrow \vdash \Box\Phi$. The essence of the logic is the fact that there are no specific assumptions for the properties of the time structure. The following formulas may be considered as typical examples: $action \Rightarrow \Diamond reaction$, $\Box(send \Rightarrow \Diamond receive)$, $\Diamond alive$, $\Box\neg(badevent)$, etc. The considerations of the work are limited to this logic since it allows to define many system properties (safety, liveness); it is also easier to build a deduction engine, or use existing verified provers, and to quickly verify the approach proposed in the work.

*Semantic tableaux* is a decision-making procedure for checking satisfiability of a formula. The method is well known in classical logic but it can also be applied in modal and temporal logics [6]. The method is based on formula decompositions. In the semantic tableaux method, at the end of the decomposition procedure, all branches of the received tree are searched for contradictions. When all branches of a tree have contradictions, it means that the inference tree is *closed*. If a negation of the initial formula is placed in the root, this leads to the statement that the initial formula is true. This method has some advantages over the traditional axiomatic approach. In the classical reasoning approach, starting from axioms, longer and more complicated formulas are generated and derived. Formulas become longer and longer step by step, and only one of them will lead to the verified formula. The method of semantic tableaux is characterized by a reverse strategy. The method provides, through so-called *open* branches of the semantic tree, information about the source of an error, if one is found, which is another and very important advantage of the method. Summing up, the tableaux are global, goal-oriented and "backward", while resolution is local and "forward".

A simple yet illustrative example of an inference tree is shown in the left side of Fig. 3. The relatively short formula gives a small inference tree, but shows how the method works. The label $[i, j]$ means that it is the $i$-th formula, i.e. the $i$-th decomposition step, received from the decomposition transformation of a formula stored in the $j$-th node. The label "1 :" represents the initial world in which a formula is true. The label "$1.(x)$", where $x$ is a free variable, represents all possible worlds that are consequences of world 1. On the other hand, the label "$1.[p]$", where $p$ is an atomic formula, represents one of the possible worlds, i.e. a successor of world 1, where formula $p$ is true. The decomposition procedure adopted and presented here refers to the first-order predicate calculus and can be found, for example, in the work [9]. All branches of the analyzed trees are closed ($\times$). There is no valuation that satisfies the root formula. This, consequently, means that the formula before the negation is always satisfied.

An outline architecture of the proposed deduction-based verification system is presented in Fig. 3. A similar system is proposed in work [13]. The system works automatically and consists of some important elements. The $\boxed{\text{G}}$ component generates logical specifications which are sets of a usually large number of temporal logic formulas (of class K). Formula generation is performed automatically from

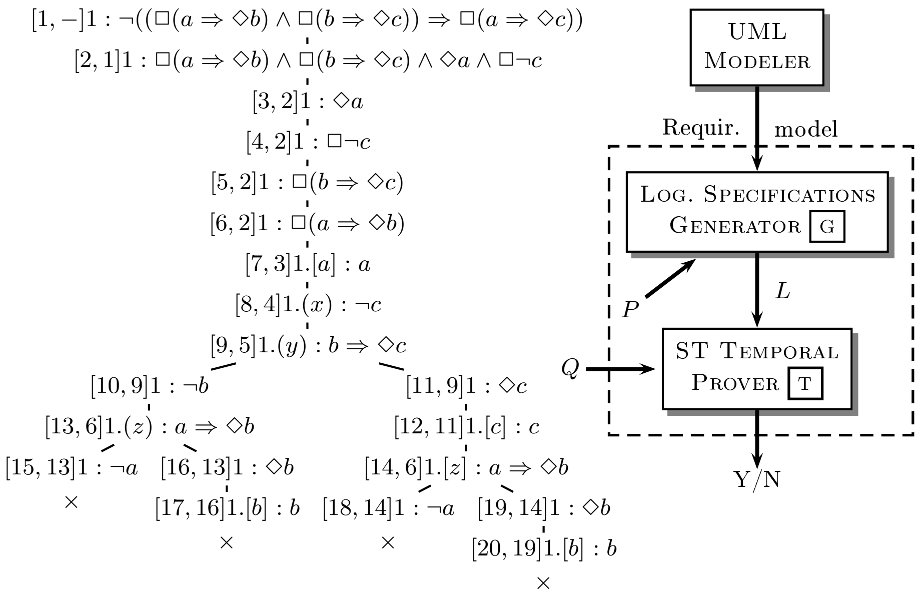

$[1,-]1 : \neg((\Box(a \Rightarrow \Diamond b) \wedge \Box(b \Rightarrow \Diamond c)) \Rightarrow \Box(a \Rightarrow \Diamond c))$
$[2,1]1 : \Box(a \Rightarrow \Diamond b) \wedge \Box(b \Rightarrow \Diamond c) \wedge \Diamond a \wedge \Box\neg c$
$[3,2]1 : \Diamond a$
$[4,2]1 : \Box\neg c$
$[5,2]1 : \Box(b \Rightarrow \Diamond c)$
$[6,2]1 : \Box(a \Rightarrow \Diamond b)$
$[7,3]1.[a] : a$
$[8,4]1.(x) : \neg c$
$[9,5]1.(y) : b \Rightarrow \Diamond c$
$[10,9]1 : \neg b$
$[11,9]1 : \Diamond c$
$[13,6]1.(z) : a \Rightarrow \Diamond b$
$[12,11]1.[c] : c$
$[15,13]1 : \neg a$
$[16,13]1 : \Diamond b$
$[14,6]1.[z] : a \Rightarrow \Diamond b$
$\times$
$[17,16]1.[b] : b$
$[18,14]1 : \neg a$
$[19,14]1 : \Diamond b$
$\times$
$\times$
$[20,19]1.[b] : b$
$\times$
UML Modeler
Requir. model
Log. Specifications Generator G
P
L
ST Temporal Prover T
Q
Y/N

**Fig. 3.** A sample inference tree (left) and a deduction-based verification system (right)

workflow models, which are constructed from predefined patterns for activity diagrams. The extraction process is discussed in section 6. The whole specification $L$ can be treated as a conjunction of formulas $f_1 \wedge \ldots \wedge f_n = C(L)$, where every $f_i$ is a formula generated during the extraction process. The $Q$ formula is a desired property for a requirements model. Both the system specification and the examined properties are input to the $\boxed{\text{T}}$ component, i.e. *Semantic Tableaux Temporal Prover*, or shortly *ST Temporal Prover*, which enables the automated reasoning in temporal logic using semantic tableaux. The input for this component is the formula $C(L) \Rightarrow Q$, or, more precisely:

$$f_1 \wedge \ldots \wedge f_n \Rightarrow Q \tag{1}$$

Due to the fact that the semantic tableaux method is an indirect proof, i.e. *reductio ad absurdum*, 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, it means that the initial formula 1 is true. The output of the $\boxed{\text{T}}$ component, and therefore also the output of the entire deductive system, is the answer Yes/No. This output also realizes the final step of the procedure shown in Fig. 1. However, the verification procedure can be performed for the further properties, c.f. the loop in Fig. 1.

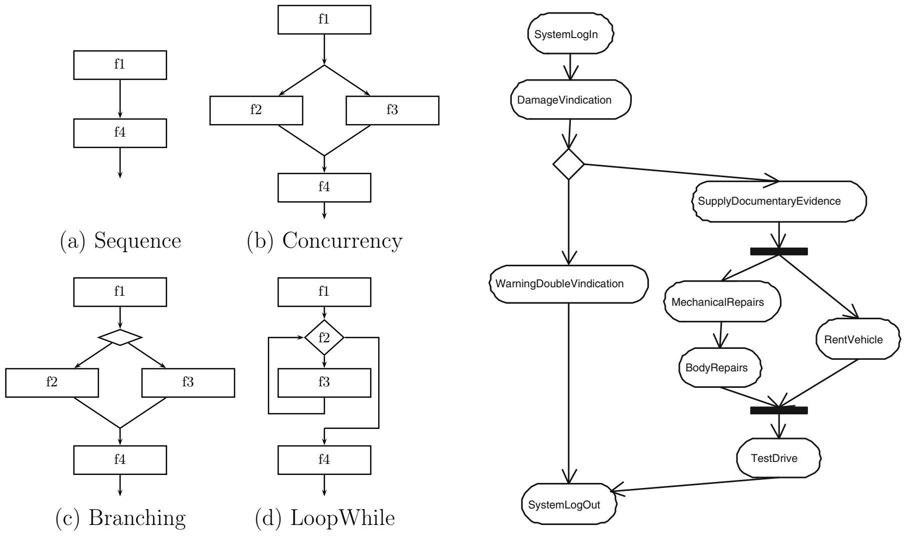
The verification procedure which results from the deduction system in Fig. 3 can be summarized as follows:

1. automatic generation of system specifications (the $\boxed{\text{G}}$ component);
2. introduction of the property $Q$ of the system;
3. automatic inference using semantic tableaux (the $\boxed{\text{T}}$ component) for the whole complex formula, c.f. formula 1.

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

## 5 Workflow Patterns and Modeling Activities

Activity diagrams constitute a closure of the development phase for requirements models, by introducing dynamic aspects for models. This aspect is subjected to the correctness analysis for safety and liveness properties. The *activity diagram* enables modeling of workflow activities. It constitutes a graphical representation of workflow showing the flow of control from one activity to another. It supports choice, concurrency and iteration. The important goal of activity diagrams is to show how an activity depends on others [16].



(a) Sequence (b) Concurrency (c) Branching (d) LoopWhile

**Fig. 4.** Workflow patterns for activities (left) and a sample activity diagram $AD_3$ for use case $UC_3$ "InsuranceDamageLiquidation" (right)

From the viewpoint of the approach presented in the work, it is important to introduce a number of predefined workflow patterns for activities that provide all workflows in a structural form. A *pattern* is a generic description of the structure of some computations. Nesting of patterns is permitted. The following workflow patterns are predefined: *sequence*, *concurrent fork/join*, *branching* and *loop while* for iteration as they are shown in Fig. 4. It is assumed that only predefined patterns can be used for modeling of activity behavior. Such structuring is not a limitation when modeling arbitrarily complex sets of activities.

For every use case $UC_i$ and its scenario, a activity diagram $AD_i$ is developed/-modeled. The activity diagram workflow is modeled only using atomic activities which are identified when building a use case scenario. Furthermore, workflows are composed only using the predefined design patterns shown in Fig. 4. A sample activity diagram $AD_3$ is shown in Fig. 4. It models behavior of the $UC_3$ use case shown in Fig. 2, using activities from the scenario in Fig. 2. After the start of the vindication process, i.e. "DamageVindication", it is checked whether it is already being processed. If yes, the decision to register this fact is made, as it is likely another attempt at vindication of the same event, c.f. "Warning-DoubleVindication". The scenario analysis and the nature of other activities, i.e. "MechanicalRepairs", "BodyRepairs" and "RentVehicle", leads to the conclusion that they can and should be performed concurrently.

## 6 Generating Logical Specifications

The phase of modeling requirements is complete when all activity diagrams for all scenarios are built, c.f. Fig. 1 and section 5. Then, the phase of generating logical specifications and formal analysis of the desired properties begins. The logical specification generation process must be performed in an automatic way. Such logical specifications usually consist of a large number of temporal logic formulas and their manual development is practically impossible since this process can be monotonous, error-prone and the creation of such logical specifications is difficult for inexperienced analysts. On the other hand, the verified properties of the system constitute usually easier formulas, not to mention the fact that they are rather individual temporal logic formulas.

The proposed algorithm for automatic extraction of logical specifications is based on the assumption that all workflows for activity diagrams are built using only well-known workflow patterns, c.f. Fig. 4. The process of building a logical specification can be presented in the following steps:

1. analysis of activity diagrams to extract all predefined workflow patterns,
2. translation of the extracted patterns to a logical expression $W_L$,
3. generating a logical specification $L$ from logical expressions, i.e. receiving a set of temporal logic formulas.

Predefined workflow patterns constitute a kind of primitives which are defined using temporal logic formulas. Therefore, an *elementary set* $pat()$ of formulas over atomic formulas $a_i$, where $i > 0$, which is also denoted $pat(a_i)$, is a set of temporal logic formulas $f_1, ..., f_m$ such that all formulas are syntactically correct (and restricted to the logic K). For example, an elementary set $pat(a, b, c, d) = \{a \Rightarrow \Diamond b, b \Rightarrow \Diamond(c \vee d), \Box\neg((a \vee b) \wedge \neg c)\}$ is a three-element set of PLTL formulas, created over four atomic formulas. Let $\Sigma$ be a set of *predefined design patterns*, i.e. $\Sigma = \{Sequence, Concurrency, Branching, LoopWhile\}$. The proposed temporal logic formulas should describe both safety and liveness properties of each pattern. Let us introduce some aliases: *Seq* as *Sequence*, *Concur* as *Concurrency*, *Branch* as *Branching* and *Loop* as *LoopWhile*.

Every activity workflow is designed using only predefined design patterns. Every design pattern has a predefined and countable set of linear temporal logic formulas. The workflow model can be quite complex and it may contain nesting patterns. Let us define a logical expression, which is similar to well known regular expressions, to represent any potentially complex structure of the activity workflow but also to have a literal representation for these workflows. The *logical expression* $W_L$ is a structure created using the following rules:

– every elementary set $pat(a_i)$, where $i > 0$ and every $a_i$ is an atomic formula, is a logical expression,
– every $pat(A_i)$, where $i > 0$ and every $A_i$ is either
  • an atomic formula, or
  • a logical expression $pat()$,
  is also a logical expression.

Examples of logical expressions are given in the section 7.

```
                                                   /* ver. 6.04.2013 */
Sequence(f1,f4):
f1 => <>f4 / ~f1 => ~<>f4 / []~(f1 & f4)
Concurrency(f1,f2,f3,f4):
f1 => <>f2 & <>f3 / ~f1 => ~(<>f2 & <>f3)
f2 & f3 => <>f4 / ~(f2 & f3) => ~<>f4
[]~(f1 & (f2 | f3)) / []~((f2 | f3) & f4) / []~(f1 & f4)
Branching(f1,f2,f3,f4):
f1 => (<>f2 & ~<>f3) | (~<>f2 & <>f3)
~f1 => ~((<>f2 & ~<>f3) | (~<>f2 & <>f3))
f2 | f3 => <>f4 / ~(f2 | f3) => ~<>f4 / []~(f1 & f4)
[]~(f2 & f3) / []~(f1 & (f2 | f3)) / []~((f2 | f3) & f4)
LoopWhile(f1,f2,f3,f4):
f1 => <>f2 / ~f1 => ~<>f2
f2 & c(f2) => <>f3 & ~<>f4 / ~(f2 & c(f2)) => ~(<>f3 & ~<>f4)
f2 & ~c(f2) => ~<>f3 & <>f4 / ~(f2 & ~c(f2)) => ~(~<>f3 & <>f4)
f3 => <>f2 / ~f3 => ~<>f2
[]~(f1 & f2) / []~(f1 & f3) / []~(f1 & f4)
[]~(f2 & f3) / []~(f2 & f4) / []~(f3 & f4)
```

**Fig. 5.** A predefined set of patterns $P$ and their temporal properties

The last step is to define a logical specification which is generated from logical expressions. The *logical specification* $L$ consists of all formulas derived from a logical expression $W_L$ using the algorithm $\Pi$, i.e. $L(W_L) = \{f_i : i \geq 0 \wedge f_i \in \Pi(W_L, P)\}$, where $f_i$ is a TL formula. Generating logical specifications is not a simple summation of formula collections resulting from a logical expression. The generation algorithm has two inputs. The first one is a logical expression $W_L$ which is a kind of variable, i.e. it varies for every (workflow) model, when the workflow is subjected to any modification. The second one is a predefined set $P$ which is a kind of constant, i.e. once defined then widely used. The example of such a set is shown in Fig 5. However, the formulas are not discussed in the work because of its limited size. They might be a subject of consideration in a separate work. Moreover, the formulas can and should be prepared by an expert with skills and theoretical background. It guarantees that an inexperienced software analyst or engineer will be able to obtain correct logical models. Most elements of the predefined $P$ set, i.e. comments, two temporal logic operators, classical logic operators, are not in doubt. The slash allows to place more formulas in a single line. $f_1$, $f_2$ etc. are atomic formulas for a pattern. They constitute a kind of formal arguments for a pattern. $\Diamond f$ means that sometime (or eventually in the future), activity $f$ is satisfied, i.e. the token reaches the activity. $c(f)$ means that the logical condition associated with activity $f$ has been evaluated and is satisfied. All formulas describe both safety and liveness properties for a pattern [1].

The output of the generation algorithm is a logical specification understood as a set of temporal logic formulas. The algorithm ($\Pi$) is as follows:

1. at the beginning, the logical specification is empty, i.e. $L = \emptyset$;
2. the most nested pattern or patterns are processed first, then, less nested patterns are processed one by one, i.e. patterns that are located more towards the outside;
3. if the currently analyzed pattern consists only of atomic formulas, the logical specification is extended, by summing sets, by formulas linked to the pattern currently being analyzed $pat()$, i.e. $L = L \cup pat()$;
4. if any argument is a pattern itself, then
   (a) firstly, the $f1$ formula, and next
   (b) the $f4$ formula
   of this pattern (if any), or otherwise considering only the most nested nesting far left, or right, respectively, are substituted separately in the place of the pattern as an argument.

Let us supplement the algorithm by some examples. The example for the step 3: $Concur(a, b, c, d)$ gives $L = \{a \Rightarrow \Diamond b \wedge \Diamond c, \neg a \Rightarrow \neg(\Diamond b \wedge \Diamond c), b \wedge c \Rightarrow \Diamond d, \neg(b \wedge c) \Rightarrow \neg\Diamond d, \Box\neg(a \wedge (b \vee c)), \Box\neg((b \vee c) \wedge d), \Box\neg(a \wedge d)\}$. The example for the step 4: $Branch(Seq(a, b), c, d, e)$ leads to $L = \boxed{\{} a \Rightarrow \Diamond b, \neg a \Rightarrow \neg\Diamond b, \Box\neg(a \wedge b)\} \cup \boxed{\{} a \Rightarrow (\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d), \neg a \Rightarrow \neg((\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d)), c \vee d \Rightarrow \Diamond e, \neg(c \vee d) \Rightarrow \neg\Diamond e, \Box\neg(c \wedge d), \Box\neg(a \wedge (c \vee d)), \Box\neg((c \vee d) \wedge e), \Box\neg(a \wedge e)\} \cup \boxed{\{} b \Rightarrow (\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d), \neg b \Rightarrow \neg((\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d)), c \vee d \Rightarrow \Diamond e, \neg(c \vee d) \Rightarrow \neg\Diamond e, \Box\neg(c \wedge d), \Box\neg(b \wedge (c \vee d)), \Box\neg((c \vee d) \wedge e), \Box\neg(b \wedge e)\} \boxed{= \{} a \Rightarrow (\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d), \neg a \Rightarrow \neg((\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d)), c \vee d \Rightarrow \Diamond e, \neg(c \vee d) \Rightarrow \neg\Diamond e, \Box\neg(c \wedge d), \Box\neg(a \wedge (c \vee d)), \Box\neg((c \vee d) \wedge e), \Box\neg(a \wedge e), b \Rightarrow (\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d), \neg b \Rightarrow \neg((\Diamond c \wedge \neg\Diamond d) \vee (\neg\Diamond c \wedge \Diamond d)), \Box\neg(b \wedge (c \vee d)), \Box\neg(b \wedge e)\}$. The first set follows from the $\boxed{\text{nested}}$ pattern, the second set follows directly from the algorithm point $\boxed{\text{4a}}$, and then the third set follows from the algorithm point $\boxed{\text{4b}}$, while the $\boxed{\text{final}}$ specification is the sum of all generated sets.

**Remarks.** Formulas $f1$ and $f4$ play an important role for every pattern, i.e. they are certainly the first and the last, respectively, active activity/task for a pattern. In the case of nested patterns, $f1$ and $f4$ enable considering the pattern as a whole, which is the goal of the last step of the algorithm. It is mandatory for every two patterns to have disjoint sets of atomic activities; moreover, every two patterns contained in a logical expression are either disjointed or completely contained in one another, c.f. formula 2, which, in conjunction with a particular role of $f1$ and $f4$, does not lead to potential contradictions. Formulas $f1$ and $f4$ must be considered separately, c.f. 4a and 4b of the algorithms, in order to guarantee access to a pattern both to/from the "front" and to/from the "back" of a pattern with respect to both the preceding and the following pattern. It may cause some redundancy of generated formulas, but on the other hand it covers all properties of combined patterns, i.e. it guarantees reachability (liveness), if necessary, of all (individual) activities.

## 7 Reasoning and Verification

Let us summarize the entire method proposed in the work. The first phase, let us call it the *modeling phase*, enables development of requirements models and includes the following steps:

– modeling of all use case diagrams $UCD_1, ..., UCD_m$, where $UC_1, ..., UC_n$ are all use cases contained in all use case diagrams;
– modeling of scenarios for all use cases $UC_1, ..., UC_n$ and identification of atomic activities $AA = \{a_1, ..., a_l\}$;
– modeling of activity diagrams $AD_1, ..., AD_n$ for all scenarios using predefined workflow patterns, c.f. Fig. 4, and using the identified atomic activities.

All the above steps require the assistance of an engineer and cannot be done automatically. The next phase, let us call it the *analytical phase*, introduces a certain degree of automation and includes the following steps:

– translation of all activity diagrams $AD_1, ..., AD_n$ (and their workflows) to logical expressions $W_{L,1}, ..., W_{L,n}$;
– generation of logical specifications $L_1, ..., L_n$ for all logical expressions using the $\Pi$ algorithm, i.e. $\Pi(P, W_{L,i}) \longrightarrow L_i$ for every $i = 1, ..., n$;
– summing of specifications, i.e. $L = L_1 \cup ... \cup L_n$;
– (manual) definition of the desired property $Q$;
– start of the process of automatic reasoning using the semantic tableaux method for formula $f_1 \wedge ... \wedge f_k \Rightarrow Q$, where $f_1, ..., f_k$ are formulas which belong to the logical specification $L$.

The above steps illustrate the entire operation of the system shown in Fig. 3. The loop between the last two steps, c.f. Fig. 1, refers to a process of both introducing and verifying more and more new properties (formula $Q$) of the examined model.

Let us consider the activity diagram $AD_3$ shown in Fig. 4 for use case $UC_3$ "InsuranceDamageLiquidation". Activity diagrams constitute the input for the deduction system shown in Fig. 3. The logical expression $W_{L,3}$ for $AD_3$ is

$Seq(SystemLogIn, Branch(DamageVindication, Concur($
$SupplyDocumentaryEvidence, Seq(MechanicalRepairs, BodyRepairs),$
$RentVehicle, TestDrive), WarningDoubleVindication, SystemLogOut))$

Substituting letters of the Latin alphabet in places of propositions: $a$ – SystemLogIn, $b$ – DamageVindication, $c$ – SupplyDocumentaryEvidence, $d$ – MechanicalRepairs, $e$ – BodyRepairs, $f$ – RentVehicle, $g$ – TestDrive, $h$ – WarningDoubleVindication, and $i$ – SystemLogOut, then the expression $W_{L,3}$ is

$$Seq(a, Branch(b, Concur(c, Seq(d, e), f, g), h, i)) \tag{2}$$

Replacing propositions (atomic activities) by Latin letters is a technical matter. In the real world, original names of the activities would be used.

A logical specification $L$ for the logical expression $W_{L,3}$ is built in the following steps. At the beginning, the specification of a model is $L = \emptyset$. Most nested pattern is *Seq*. The next considered pattern is *Concurrency*, and then *Branching*. The most outside situated pattern is once again *Seq*. The resulting logical specification contains the formulas

$$\begin{aligned} L = \{&d \Rightarrow \Diamond e, \neg d \Rightarrow \neg\Diamond e, \Box\neg(d \wedge e), c \Rightarrow \Diamond d \wedge \Diamond f, \neg c \Rightarrow \neg(\Diamond d \wedge \Diamond f),\\ &d \wedge f \Rightarrow \Diamond g, \neg(d \wedge f) \Rightarrow \neg\Diamond g, \Box\neg(c \wedge (d \vee f)), \Box\neg((d \vee f) \wedge g),\\ &\Box\neg(c \wedge g), c \Rightarrow \Diamond e \wedge \Diamond f, \neg c \Rightarrow \neg(\Diamond e \wedge \Diamond f), e \wedge f \Rightarrow \Diamond g,\\ &\neg(e \wedge f) \Rightarrow \neg\Diamond g, \Box\neg(c \wedge (e \vee f)), \Box\neg((e \vee f) \wedge g),\\ &b \Rightarrow (\Diamond c \wedge \neg\Diamond h) \vee (\neg\Diamond c \wedge \Diamond h), \neg b \Rightarrow \neg((\Diamond c \wedge \neg\Diamond h) \vee (\neg\Diamond c \wedge \Diamond h)),\\ &c \vee h \Rightarrow \Diamond b, \neg(c \vee h) \Rightarrow \neg\Diamond b, \Box\neg(c \wedge h), \Box\neg(b \wedge (c \vee h)),\\ &\Box\neg((c \vee h) \wedge i), \Box\neg(b \wedge i), b \Rightarrow (\Diamond g \wedge \neg\Diamond h) \vee (\neg\Diamond g \wedge \Diamond h),\\ &\neg b \Rightarrow \neg((\Diamond g \wedge \neg\Diamond h) \vee (\neg\Diamond g \wedge \Diamond h)), g \vee h \Rightarrow \Diamond b, \neg(g \vee h) \Rightarrow \neg\Diamond b,\\ &\Box\neg(g \wedge h), \Box\neg(b \wedge (g \vee h)), \Box\neg((g \vee h) \wedge i),\\ &a \Rightarrow \Diamond b, \neg a \Rightarrow \neg\Diamond b, \Box\neg(a \wedge b), a \Rightarrow \Diamond i, \neg a \Rightarrow \neg\Diamond i, \Box\neg(a \wedge i)\} \end{aligned} \tag{3}$$

Formula 3 represents the output of the $\boxed{\text{G}}$ component in Fig. 3.

Formal *verification* is the act of proving the correctness of a system (liveness, safety). *Liveness* means that the computational process achieves its goals, i.e. something good eventually happens. *Safety* means that the computational process avoids undesirable situations, i.e. something bad never happens. The liveness property for the model can be

$$b \Rightarrow \Diamond f \tag{4}$$

which means that **if the damage vindication is satisfied then sometime in the future the replacement car is reached**, formally $DamageVindication \Rightarrow \Diamond RentVehicle$. The safety property for the examined model can be

$$\Box\neg(h \wedge f) \tag{5}$$

which means that **it never occurs that the rental of a vehicle and the double vindication are satisfied in the same time**, or more formally $\Box\neg(WarningDoubleVindication \wedge RentVehicle)$. When considering the property expressed by formula 4 then the whole formula to be analyzed using semantic tableaux, providing a combined input for the $\boxed{\text{T}}$ component in Fig. 3, is

$$((d \Rightarrow \Diamond e) \wedge (\neg d \Rightarrow \neg\Diamond e) \wedge ... \wedge (\neg a \Rightarrow \neg\Diamond i) \wedge (\Box\neg(a \wedge i))) \Rightarrow (b \Rightarrow \Diamond f) \quad (6)$$

When considering the property expressed by formula 5 then the whole formula is constructed in a similar way as

$$((d \Rightarrow \Diamond e) \wedge (\neg d \Rightarrow \neg\Diamond e) \wedge ... \wedge (\neg a \Rightarrow \neg\Diamond i) \wedge (\Box\neg(a \wedge i))) \Rightarrow (\Box\neg(h \wedge f)) \quad (7)$$

In both cases, i.e. formulas 6 and 7, after the negation of the input formula within the prover, the inference trees are built. Presentation of a full inference tree for both cases exceeds the size of the work. (The simple inference tree from Fig. 3 gives an idea how it works.) All branches of the semantic trees are closed, i.e. formulas 4 and 5 are satisfied in the considered requirements model. In the case of falsification of the semantic tree the open branches are obtained and provide information about the source of an error what is another advantage of the method.

Although the logical specification was generated for only one activity diagram $AD_3$, that is $L = L_3$, c.f. formula 3, the method is easy to scale up, i.e. extending and summing up logical specifications for other activity diagrams and their scenarios. Then, it will be possible to examine logical relationships (liveness, safety) for different activities coming from different activity diagrams.

## 8 Conclusion

The work proposes a two-phase strategy for formal analysis of requirements models. The first one is carried out by an engineer using a defined methodology and the second one can be (in most) automatic and enables formal verification of the desired properties (liveness, safety). Introducing logical patterns as logical primitives allows for breaking of some barriers and obstacles in receiving logical specifications as a set of a large number of temporal logic formulas in an automated way. Application of formal verification, which is based on deductive inference, helps to significantly increase the maturity of requirements models considering infinite computations and using a human-intuitive approach.

Future works may include the implementation of the logical specification generation module and the temporal logic prover. Considering graph transformations [14] is encouraging for requirements models involving distributed representation of knowledge and their efficient implementation. It should result in a CASE software which could be a first step involved in creating industrial-proof tools, i.e. implementing another part of formal methods, hope promising, in industrial practice.

**Acknowledgement.** The author would like to thank the anonymous Reviewers for their valuable comments that helped to improve the work. This work is co-financed by the EU, Human Capital Operational Programme, SPIN project no. 502.120.2066/C96 and co-financed by the AGH Research Fund no. 11.11 120.859.

## References

1. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21(4), 181–185 (1985)
2. Barrett, S., Sinnig, D., Chalin, P., Butler, G.: Merging of use case models: Semantic foundations. In: 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009), pp. 182–189 (2009)
3. van Benthem, J.: Temporal Logic. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 4, pp. 241–350. Clarendon Press (1993–1995)
4. Chakraborty, S., Sarker, S., Sarker, S.: An exploration into the process of requirements elicitation: A grounded approach. Journal of the Association for Information Systems 11(4), 212–249 (2010)
5. Clarke, E., Wing, J., et al.: Formal methods: State of the art and future directions. ACM Computing Surveys 28(4), 626–643 (1996)
6. d'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J.: Handbook of Tableau Methods. Kluwer Academic Publishers (1999)
7. Emerson, E.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier, MIT Press (1990)
8. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. IEEE Transactions on Software Engineering 30(7), 437–447 (2004)
9. Hähnle, R.: Tableau-based Theorem Proving. ESSLLI Course (1998)
10. Hurlbut, R.R.: A survey of approaches for describing and formalizing use cases. Tech. Rep. XPT-TR-97-03, Expertech, Ltd. (1997)
11. Kazhamiakin, R., Pistore, M., Roveri, M.: Formal verification of requirements using spin: A case study on web services. In: Proceedings of 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, pp. 406–415 (September 28-30, 2004)
12. Klimek, R.: Proposal to improve the requirements process through formal verification using deductive approach. In: Filipe, J., Maciaszek, L. (eds.) Proceedings of 7th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE 2012), Wrocław, Poland. pp. 105–114. SciTePress (June 29–30, 2012)
13. Klimek, R.: A Deduction-based System for Formal Verification of Agent-ready Web Services. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems, Frontiers of Artificial Intelligence and Applications, vol. 252, pp. 203–212. IOS Press (2013), `http://ebooks.iospress.nl/publication/32843`
14. Kotulski, L.: Supporting software agents by the graph transformation systems. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3993, pp. 887–890. Springer, Heidelberg (2006)
15. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. John Wiley & Sons (2009)
16. Pender, T.: UML Bible. John Wiley & Sons (2003)
17. Rauf, R., Antkiewicz, M., Czarnecki, K.: Logical structure extraction from software requirements documents. In: 19th IEEE International Requirements Engineering Conference (RE 2011), Trento, Italy, August 29-September 2, pp. 101–110. IEEE Computer Society (2011)
18. Schneider, G., Winters, J.: Applying use cases: a practical guide. Addison-Wesley (2001)
19. Wolter, F., Wooldridge, M.: Temporal and dynamic logic. Journal of Indian Council of Philosophical Research XXVII(1), 249–276 (2011)
20. Zhao, J., Duan, Z.: Verification of use case with petri nets in requirement analysis. In: Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2009, Part II. LNCS, vol. 5593, pp. 29–42. Springer, Heidelberg (2009)