Deduction-based Formal Verification of Requirements Models with Automatic Generation of Logical Specifications

Radosław Klimek

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

Abstract. This work concerns requirements gathering and their formal verification using deductive approach. The approach is based on temporal logic and the semantic tableaux reasoning method. Requirements elicitation is carried out with some UML diagrams. A use case, its scenario and its activity diagram may be linked to each other during the process of gathering requirements. Activities are identified in the use case scenario and then their workflows are modeled using the activity diagram. Organizing the activity diagram workflows into design patterns is crucial and enables generating logical specifications in an automated way. Temporal logic specifications, formulas and properties are difficult to specify by inexperienced users and this fact can be a significant obstacle to the practical use of deduction-based verification tools. The approach presented in this paper attempts to overcome this problem. Automatic transformation of workflow patterns to temporal logic formulas considered as a logical specification is defined. The architecture of an automatic generation and deduction-based verification system is proposed.

Keywords: requirements engineering, UML, use case diagram, use case scenario, activity diagram, formal verification, deductive reasoning, semantic tableaux method, temporal logic, workflows, design patterns, logical modeling, generating formulas

1 Introduction

Requirements engineering is an important part of software modeling. Requirements elicitation is an early phase in the general activity of the process of identifying and articulating purposes and needs for a new system. It should lead into a coherent structure of requirements and have significant impacts on software quality and costs. Software modeling enables better understanding of a domain problem and a developed system. Better software modeling is key in obtaining reliable software. Software reliability implies both correctness and robustness, i.e. it meets its functional (and non-functional) specifications and it delivers a service even if there are unexpected obstacles. The process of gathering requirements understood as an extra layer of abstraction improves not only the reliability of

the software development phase but also enhances corporation communication, planning, risk management and enables cost reductions. System requirements are descriptions of delivered services in the context of operational constraints. Identifying software requirements of the system-as-is, gathering requirements and the formulation of requirements by users enables the identification of defects earlier in a life cycle. Thinking of requirements must precede the code generating act. UML, i.e. Unified Modeling Language [20, 19], which is ubiquitous in the software industry can be a powerful tool for the requirements engineering process. UML 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 instrument to discover and write down requirements. UML also appears to be an adequate and convenient tool for the documentation of the whole requirements process. What is more, the use case and activity diagrams may also be used in a formal-based analysis and the verification of requirements. Formal methods in requirements engineering may improve the process through a higher level of rigor which enables a better understanding of the domain problem and deductive verification of requirements. Temporal logic is a well established formalism for describing properties of reactive systems. It may facilitate both the system specifying process and the formal verification of non-functional requirements which are usually difficult to verify. The semantic tableaux method, which is a proof formalization for deciding satisfiability and which might be more descriptively called "satisfiability trees", is very intuitive and may be considered goal-based formal reasoning to support requirements engineering.

Maintaining software reliability seems difficult. Formal methods enable the precise formulation of important artifacts arising during software development and the elimination of ambiguity and subjectivity inconsistency. Formal methods can constitute a foundation for providing natural and intuitive support for reasoning about system requirements and they guarantee a rigorous approach in software construction. There are two important parts to the formal approach, i.e. formal specification and formal reasoning, though both are quite closely related to each other. Formal specification introduces formal proofs which establish fundamental system properties and invariants. Formal reasoning enables the reliable verification of desired properties. There are two well established approaches to formal reasoning and information systems verification [6]. The first is based on the state exploration and the second is based on deductive inference. During recent years there was particularly significant progress in the field of state exploration also called "model checking" [5]. However, model checking is an operational rather than analytic approach. Model checking is a kind of simulation for all reachable paths of computation. Unfortunately, the inference method is now far behind state exploration for several reasons, one of which is the choice of a deductive system. However, a more important problem is a lack of methods for obtaining system specifications which are considered as sets of temporal logic formulas and the automation of this procedure. It is not so obvious how to build a logical specification of a system, which in practice is a large collection of temporal logical formulas. The need to build logical specifications can

be recognized as a major obstacle to untrained users. Thus, the automation of this process seems particularly important. Application of the formal approach to the requirements engineering phase may increase the maturity of the developed software.

Motivations, contributions and related works

The main motivation for this work is the lack of satisfactory and documented results of the practical application of deductive methods for the formal verification of requirement models. It is especially important due to the above mentioned model checking approach that wins points through its practical applications and tools (free or commercial). Another motivation that follows is the lack of tools for the automatic extraction of logical specifications, i.e. a set of temporal logic formulas, which is an especially important issue because of the general difficulty of obtaining such a specification. However, the requirements model built using the use case and activity diagrams seems to be suitable for such an extraction. All of the above mentioned aspects of the formal approach seem to be an intellectual challenge in software engineering. On the other hand, deductive inference is very natural and it is used intuitively in everyday life.

The main contribution of this work is a complete deduction-based system, including its architecture, which enables the automated and formal verification of software models which covers requirements models based on some UML diagrams. Another contribution is the use of a non-standard method of deduction for software models. Deduction is performed using the semantic tableaux method for temporal logic. This reasoning method seems to be intuitive. The automation of the logical specification generation process is also an important and key contribution. Theoretical possibilities of such an automation are discussed. The generation algorithm for selected design patterns is presented. Prototype versions of inference engines, i.e. provers for the minimal temporal logic, are implemented.

The approach proposed in this work, i.e. how to build a requirements model through a well-defined sequence of steps, may constitute a kind of quasi methodology and is shown in Fig. 1. The loop between the last two steps refers to a process of both identifying and verifying new and new properties of a model. The work shows that formal methods can integrate requirements engineering with the expectation of obtaining reliable software. It encourages an abstract view of a system as well as reliable and deductive-based formal verification.

Let us discuss some related works. In work by Kazhamiakin [14], a method based on formal verification of requirements using temporal logic and model checking approach is proposed, and a case study is discussed. Work by Eshuis and Wieringa [10] addresses the issues of activity diagram workflows but the goal is to translate diagrams into a format that allows model checking. Work by Hurlbut [13] provides a very detailed survey of selected issues concerning use cases. The dualism of representations and informal character of scenario documentation implies several difficulties in reasoning about system behavior and validating the consistency between diagrams and scenarios description. Work by Barrett et al. [1] presents the transition of use cases to finite state machines.

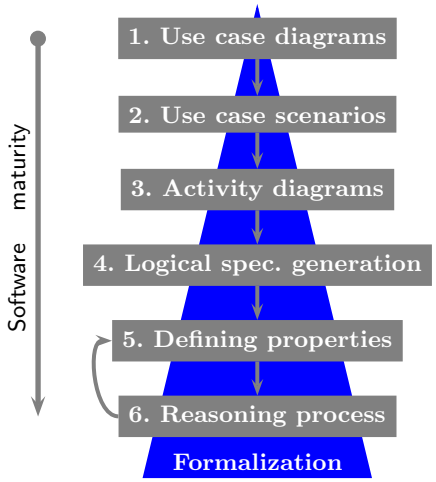


Fig. 1. Software requirements modeling and verification

Work by Zhao and Duan [22] shows the formal analysis of use cases, however the Petri Nets formalism is used. In work by Cabral and Sampaio [3] a method of automatic generation of use cases and a proposed subset of natural languages to the algebraic specification is introduced. There are some works in the context of a formal approach and UML for requirements engineering but there is a lack of works in the area of UML models and deductive verification using temporal logic and the semantic tableaux method.

2 Use case and activity diagrams

This work relates to use case and activity diagrams which are well known, e.g. [20, 11, 19, 7, 17]. They are presented from a point of view of the approach adopted in this work, which clearly leads to the formal verification of a requirements model using the deductive method, c.f. Fig. 1.

The *use case diagram* consists of actors and use cases. *Actors* are objects which interact with a system. Actors create the system's environment and they provide interaction with the system. *Use cases* are services and functionalities which are used by actors. Use cases must meet actors' requirements. A use case captures some user visible functions. The main purpose of the use case diagram is to model a system's functionality. However, the diagram does not refer to any details of the system implementation since it is a rather descriptive technique compared with the other UML diagrams. On the other hand, each use case has its associated *scenario* which is a brief narrative that describes the expected use of a system. The scenario is made up of a set of a possible sequence of steps which enables the achievement of a particular goal resulting from use case functionality.

Thus, the use case and the goal may be sometimes considered synonymous. The scenario describes a basic flow of events, and possible alternative flows of events.

From the point of view of the approach presented here it is important to draw attention to the requirement that every scenario should contain activities and actions of which individual scenario steps are built. An *activity* is a computation with its internal structure and it may be interrupted by an external event, while an *action* is an executable atomic computation which cannot be interrupted before its completion, c.f. [20]. Defining both the activity and the action results from efforts to achieve greater generality, i.e. to obtain both computations which can be interrupted and computations which cannot be interrupted.

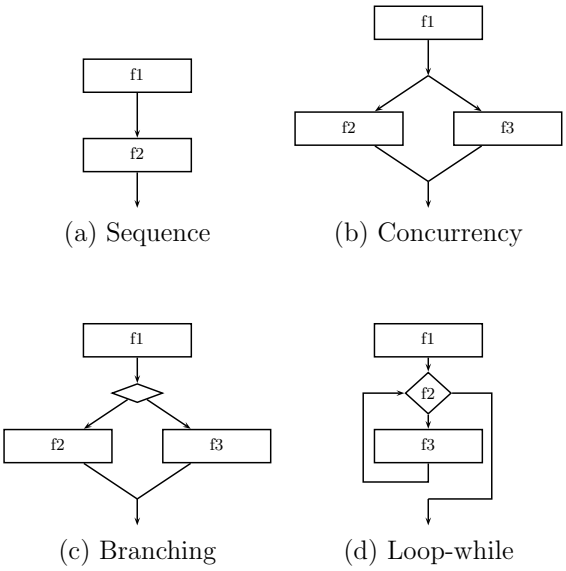
Let us summarize these considerations more formally. In the initial phase of system modeling, in the course of obtaining requirements information, use case diagrams UCD are built and they contain many use cases UC which describe the desired functionality of a system, i.e. $UC_1, ..., UC_i, ..., UC_l$, where $l > 0$ is a total number of use cases created during the modeling phase. Each use case UC_i has its own scenario. Activities or actions can and should be identified in every step of this scenario. The most valuable situation is when the whole use case scenario is not expressed in a narrative form but contains only the identified activities or actions. Thus, every scenario contains some activities $v_1, ..., v_i, ..., v_m$ and/or actions $o_1, ..., o_j, ..., o_n$, where $m \geq 0$ and $n \geq 0$. Broadly speaking every scenario contains $a_1, ..., a_k, ..., a_p$, where $p > 0$ and every a_k is an activity or an action. 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 engineering. However, one of the most important things in this approach is to identify activities and actions when creating scenarios since these objects will be used when modeling activity diagrams.

The *activity diagram* enables model activities (and actions) and workflows. It is a graphical representation of workflow showing flow of control from one activity to another one. It supports choice, concurrency and iteration. The activity diagram AD consists of *initial* and *final states* (activities) which show the starting and end point for the whole diagram, *decision points*, activities, actions, e.g. input receiving action or output sending action, and swimlanes. The *swimlane* is useful for partitioning the activity diagram and is a way to group activities in a single thread. The activity diagram shows how an activity depends on others.

The nesting activities is permitted. Every activity is:

1. either an elementary activity, i.e. syntactically indivisible unit, or
2. one of the acceptable design patterns for activity workflows/diagrams.

From the viewpoint of the approach presented in this work it is important to introduce some restrictions on activity workflows. This relies on the introduction of a number of design patterns for activities that give all the workflows a structural form. *Pattern* is a generic description of structure of some computations, and does not limit the possibility of modeling arbitrary complex sets of activities. The following design patterns for activities, c.f. Fig. 2, are introduced: *sequence*, *concurrent fork/join*, *branching* and *loop-while* for iteration.



(a) Sequence (b) Concurrency

(c) Branching (d) Loop-while

Fig. 2. Design patterns for workflow of activities

Let us summarize this section. For every use case UC_i and its scenario, which belongs to any use case diagram UCD, at least one activity diagram AD is created. Workflow for the activity diagram is modeled only using activities v_k and actions o_l which are identified when building a use case scenario. Workflows are composed only using the predefined design patterns shown in Fig. 2. When using these patterns one can build arbitrarily complex activity workflows.

3 Logical background and deduction system

Temporal logic and the semantic tableaux reasoning method are two basic instruments used in the presented approach. *Temporal logic* TL is a well established formalism for specification and verification, i.e. representing and reasoning about computations and software models, e.g. [9, 15, 21]. Temporal logic is broadly used and it can easily express liveliness and safety properties which play a key role in proving system properties. Temporal logic exists in many varieties, however, considerations in this paper are limited to axiomatic and deductive systems for the *smallest temporal logic*, which is also known as temporal logic of the class K defined, e.g. [2], as a classical propositional calculus extension of axiom $\Box(P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q)$ and inference rule $\frac{\vdash P}{\vdash \Box P}$. This logic can be developed and expanded through the introduction of more complex time structure properties [4, 18]. Examples of such enriched logics are: logic/formula D: (sample

formula) $\Box p \Rightarrow \Diamond p$; logic/formula T: $\Box p \Rightarrow p$; logic/formula G: $\Diamond\Box p \Rightarrow \Box\Diamond p$; logic/formula 4: $\Box p \Rightarrow \Box\Box p$; logic/formula 5: $\Diamond p \Rightarrow \Box\Diamond p$; logic/formula B: $p \Rightarrow \Box\Diamond p$; etc. It is also possible to combine these properties and logics to establish new logics and relationships among them, e.g. KB4 $\Leftrightarrow$ KB5, KDB4 $\Leftrightarrow$ KTB4 $\Leftrightarrow$ KT45 $\Leftrightarrow$ KT5 $\Leftrightarrow$ KTB. However, it should be clear that considerations in this work are limited to the logic K and we focus our attention on the linear time temporal logic as it is sufficient to define most system properties. The following formulas may be considered as examples of this logic's formulas: $act \Rightarrow \Diamond rec$, $\Box(sen \Rightarrow \Diamond ack)$, $\Diamond liv$, $\Box\neg(evn)$, etc.

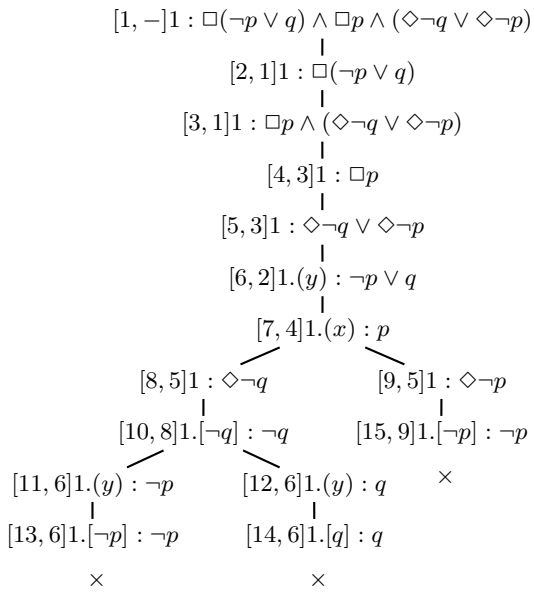


Fig. 3. The inference tree of the semantic tableaux method

Semantic tableaux, or *truth tree*, is a decision procedure for checking formula satisfiability. The truth tree method is well known in classical logic but it can also be applied in the field of temporal logics [8]. It seems that it has some advantages in comparison with the traditional axiomatic approach. The method is based on formula decompositions. At each step of a well-defined procedure, formulas have fewer components since logical connectives are removed. At the end of the decomposition procedure, all branches of the received tree are searched for contradictions. Finding a contradiction in all branches of the tree means no valuation satisfies a formula placed in the tree root. When all branches of the tree have contradictions, it means that the inference tree is *closed*. If the negation of the initial formula is placed in the root, this leads to the statement that the initial formula is true. 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 the reverse strategy. The inference structure is represented by a tree and not by a sequence of formulas. The expansion of any tree branch may be halted after finding the appropriate sub-structure. In addition, 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. The example of an inference tree for a smallest temporal logic formula is shown in Fig. 3. The purpose of this example is to demonstrate the reasoning and its specificity. The relatively short formula gives a small inference tree, but shows how the method works. Each node contains a (sub-)formula which is either already decomposed, or is subjected to decomposition in the process of building the tree. Each formula is preceded by a label referring to both a double number and a current world reference. 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 consequent 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, as well as labeling, refers to the first-order predicate calculus and can be found for example in the work of [12]. All branches of the analyzed trees are closed ($\times$). There is no valuation that satisfies the root formula. This consequently means that the formula[1] before the negation, i.e. $\neg(\Box(\neg p \vee q) \wedge \Box p \wedge (\Diamond\neg q \vee \Diamond\neg p))$, is always satisfied.

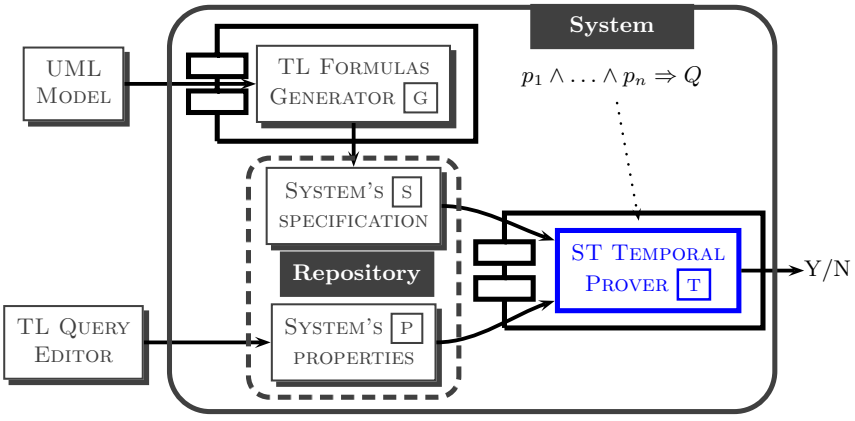


Fig. 4. Deduction-based verification system

[1] This formula and this inference tree originate from the master thesis by Łukasz Rola from AGH University of Science and Technology, Kraków, Poland. The thesis was prepared under the supervision of the author of this work.

The architecture of the proposed inference system using the semantic tableaux method for the UML activity diagram workflows is presented in Fig. 4. The system works automatically and consists of some important elements. Some of them can be treated as a software components/plugins. The first component [G] generates a logical specification which is a set of a usually large number of temporal logic formulas (of class K). Formulas generation is performed automatically by extracting directly from the design patterns contained in a workflow model. The extraction process is discussed in section 4. Formulas are collected in the [S] module (repository, i.e. file or database) that stores the system specifications. It is treated as a conjunction of formulas $p_1 \wedge \ldots \wedge p_n = P$, where $n > 0$ and p_i is a specification formula generated during the extraction. The [P] module provides the desired system properties which are expressed in temporal logic. The easiest way to obtain such formulas is to use an editor and manually introduce the temporal logic formula Q. Such a formula(s) is/are identified by an analyst and describe(s) the expected properties of the investigated system. Both the system specification and the examined properties are input to the [T] component, i.e. *Semantic Tableaux Temporal Prover*, or shortly *ST Temporal Prover*, which enables the automated reasoning in temporal logic using the semantic tableaux method. The input for this component is the formula $P \Rightarrow Q$, or, more precisely:

$$p_1 \wedge \ldots \wedge p_n \Rightarrow Q \tag{1}$$

After the negation of 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. Broadly speaking, the output of the [T] component, and therefore also the output of the whole deductive system, is the answer Yes/No in response to any introduction of a new tested property for a system modeled using UML. This output also realizes the final step of the procedure shown in Fig. 1.

The whole verification procedure can be summarized as follows:

1. automatic generation of system specification (the [G] component), and then stored (the [S] module) as a conjunction of all extracted formulas;
2. introduction of a property of the system (the [P] module) as a temporal logic formula or formulas;
3. automatic inference using semantic tableaux (the [T] component) for the whole complex 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 there is a need for a new inference due to the revised system specification (steps 2 or 3).

4 Generation of specification

Automated support for building any logical specification of a modeled system is essential for requirements acquisition and formal verification of properties. Such

a specification usually consists of a large number of temporal logic formulas and its manual development is practically impossible since this process can be monotonous and error-prone. Last but not least, the creation of such a logical specification can be difficult for inexperienced analysts. Therefore, all efforts towards automation of this design phase are very desirable.

The proposed method and algorithm for an automatic extraction of a logical specification is based on the assumption that all workflow models for the UML activity diagrams are built using only well-known design patterns, c.f. Fig. 2. It should be noted that constructed activity workflow models are based on activities and actions identified when writing use case scenarios. The whole process of building a logical specification involves the following steps:

1. analysis of a workflow model of activities to extract all design patterns,
2. translation of the extracted patterns to a logical expression W_L which is similar to a well-known regular expression,
3. generation of a logical specification L from the logical expression, i.e. receiving a set of formulas of linear time temporal logic of class K.

Let us introduce all formal concepts necessary for the application of the above steps to illustrate the entire procedure in a more formal way.

The temporal logic *alphabet* consists of the following symbols: a countable set of atomic formulas p, q, r, etc., classical logic symbols like **true**, **false**, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$ and two linear temporal logic operators $\Box$ and $\Diamond$. (One can also introduce other symbols, e.g. parenthesis, which are omitted here to simplify the definition.) For this alphabet, syntax rules for building well-formed logic formulas can be defined. These rules are based on BNF notation. The definition of *formula* of linear time temporal logic LTL includes the following steps:

- every atomic formula p, q, r, etc. is a formula,
- if p and q are formulas, then $\neg p$, $p \vee q$, $p \wedge q$, $p \Rightarrow q$, $p \Leftrightarrow q$ are formulas, too,
- if p and q are formulas, then $\Box p$, $\Diamond p$, are formulas, too.

Examples of valid, well-formed and typical formulas, restricted to the logic K, are the following formulas: $p \Rightarrow \Diamond q$ and $\Box\neg(p \wedge (q \vee r))$.

A set of LTL formulas describe temporal properties of individual design patterns of every UML activity diagram. This aspect is important as the approach presented here is based on predefined design patterns. An *elementary set* of formulas over atomic formulas a_i, where $i = 1, \ldots, n$, which is denoted $pat(a_i)$, is a set of temporal logic formulas $f_1, ..., f_m$ such that all formulas are well-formed (and restricted to the logic K). For example, an elementary set $pat(a, b, c) = \{a \Rightarrow \Diamond b, \Box\neg(b \wedge \neg c)\}$ is a two-element set of LTL formulas, created over three atomic formulas.

Suppose that there are predefined sets of formulas for every design pattern of the UML activity workflow from Fig. 2. The proposed temporal logic formulas should describe both safety and liveness properties of each pattern. In this way, $Sequence(a, b) = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$ describes properties of the Sequence pattern. Set $Concurrency(a, b, c) = \{a \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$ describes

the Concurrency pattern and $Branching(a, b, c) = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$ the Branching pattern.

Let us introduce some aliases for all patterns (Fig. 2): *Seq* as *Sequence*, *Concur* as *Concurrency*, *Branch* as *Branching* and *Loop* as *Loop−while*.

Every activity diagram 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. This is one important reason why it is needed to define a symbolic notation which represents any potentially complex structure of the activity workflow. *Logical expression* W_L is a structure created using the following rules:

- every elementary set $pat(a_i)$, where $i = 1, \ldots, n$ and a_i is an atomic formula, is a logical expression,
- every $pat(A_i)$, where $i = 1, \ldots, n$ and where A_i is either
  - a sequence of atomic formulas a_j, where $j = 1, \ldots, m$, or
  - a set $pat(a_j)$, where $j = 1, \ldots, m$ and a_j is an atomic formula, or
  - a logical expression $pat(A_j)$, where $j = 1, \ldots, m$

  is also a logical expression.

The above defined symbolic notation is equivalent to a graphical one which is usually the most convenient for users when modeling a system. However, another rule describing a special case of a sequence of sequences is introduced:

- if $pat_1()$ and $pat_2()$ are logical expressions, where the empty parentheses means any arguments, then their concatenation $pat_1() \cdot pat_2()$, also noted $pat_1()pat_2()$, is also a logical expression.

This rule is redundant but the concatenation of sequences seems more convenient and in addition it provides concatenation of sequences as a sequence of three arguments. Thus, another predefined set $SeqSeq(a, b, c) = \{a \Rightarrow \Diamond b, b \Rightarrow \Diamond c, \Box\neg((a \wedge b) \vee (b \wedge c) \vee (a \wedge c))\}$ is introduced. It describes the concatenation properties of sequences of two patterns.

Any logical expression represents an arbitrary complex and nested workflow model for an activity diagram which was modeled using predefined design patterns. The logical expression allows representation of sets of temporal logic formulas for every design pattern. Thus, the last step is to define a logical specification which is generated from a logical expression. *Logical specification* L consists of all formulas derived from a logical expression, i.e. $L(W_L) = \{f_i : i > 0\}$, where f_i is a temporal logic formula. Generating logical specifications, which constitutes a set of formulas, is not a simple summation of formula collections resulting from a logical expression. Thus, the sketch of the generation algorithm is presented below.

The generation process of a logical specification L has two inputs. The first one is a logical expression W_L which is built for the activity workflow model. The second one is a predefined set P of temporal logic formulas for every design pattern. The example of such a set is shown in Fig 5. Most elements of the predefined P set, i.e. comments, two temporal logic operators, classical logic operators,

```
                              /* ver. 15.10.2012
/* Activity Design Patterns
Sequence(f1,f2):
f1 => <>f2
[]~(f1 & f2)
SeqSeq(f1,f2,f3):
f1 => <>f2
f2 => <>f3
[]~ ((f1 & f2) | (f2 & f3) | (f1 & f3))
Concurrency(f1,f2,f3):
f1 => <>f2 & <>f3
[]~(f1 & (f2 | f3))
Branching(f1,f2,f3):
f1 => (<>f2 & ~<>f3) | (~<>f2 & <>f3)
[]~(f2 & f3)
Loop-while(f1,f2,f3):
f1 => <>f2
f2 & c(f2) => <>f3
f2 & ~c(f2) => ~<>f3
f3 => <>f2
[]~((f1 & f2) | (f2 & f3) | (f1 & f3))
```

Fig. 5. Predefined set of patterns and their temporal properties

are not in doubt. 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 completed, i.e. the token left 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.

The output of the generation algorithm is the logical specification understood as a set of temporal logic formulas. The sketch of the algorithm 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 type of the analyzed pattern $pat()$, i.e. $L = L \cup pat()$;
4. if any argument is a pattern itself, then the logical disjunction of all its arguments, including nested arguments, is substituted in place of the pattern.

The above algorithm refers to similar ideas in work [16]. Let us supplement the algorithm by some examples. The example for the step 3: $SeqSeq(p, q, r)$, gives $L = \{a \Rightarrow \Diamond b, b \Rightarrow \Diamond c, \Box\neg((a \wedge b) \vee (b \wedge c) \vee (a \wedge c))\}$ 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 the step 4: $Concur(Seq(a, b), c, d)$ leads to $L = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\} \cup \{(a \vee b) \Rightarrow \Diamond c \wedge \Diamond d, \Box\neg((a \vee b) \wedge (c \vee d))\}$.

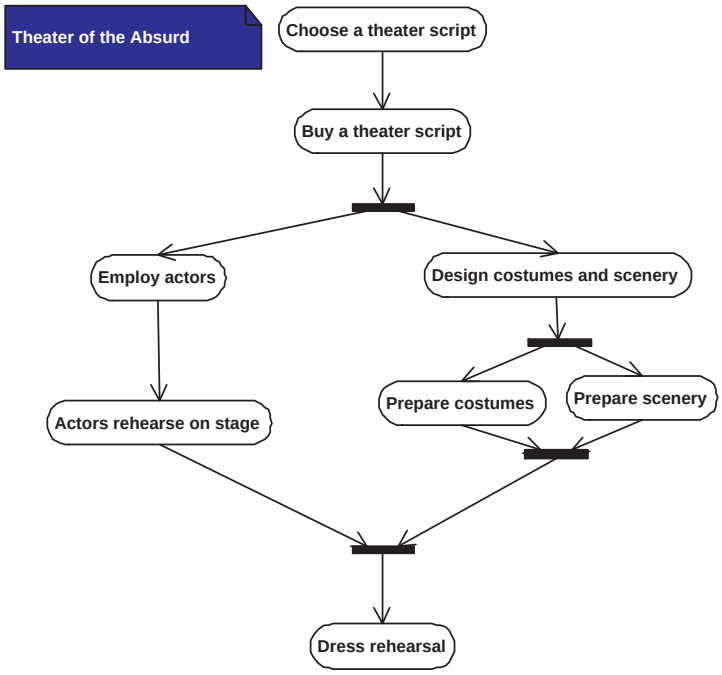


Fig. 6. A sample activity diagram and its workflow

Let us consider a simple example to illustrate the approach, c.f. Fig. 6. This example illustrates the proposed verification method only for a single activity diagram (workflow). This is due to size constraints on the work. A model consisting of several activity diagrams obtained from multiple use case scenarios could also be considered. A single liveness property (3) is verified but it is possible to verify other properties, including the safety property. When any formula describing activity is considered, e.g. $\Diamond p$, this means that the activity p is sometime (or eventually in the future) completed.

Suppose[2] that an owner of the "Theater of the Absurd", which has recently declined in popularity, wants to organize a new and great performance to begin a new and splendid period for the institution. Analyzing the scenario of the UML use case "Preparation of a theatrical performance" the following activities are identified: "Choose a theater script", "Buy a theater script", "Employ actors", "Actors rehearse on stage", "Design costumes and scenery" "Prepare costumes", "Prepare scenery" and "Dress rehearsal". The way of creating the use case scenario is not in doubt and therefore is omitted here. The scenario is usually given in a tabular form and its steps should include all the above identified activities. When building a workflow for the activity diagram, a combination of Sequence

[2] This example is an adaptation of another example from lectures by Prof. Tomasz Szmuc (AGH University of Science and Technology, Kraków, Poland).

and Concurrency patterns is used. The logical expression W_L for this model is

$$SeqSeq(ChooseScript, Concur(BuyScript, Seq(EmployActors, ActorsRehearse), Concur(Design, PrepareCostumes, PrepareScenery)), DressRehearsal)$$

or after the substitution of propositions as letters of the Latin alphabet: a – Choose a theater script, b – Buy a theater script, c – Employ actors, d – Actors rehearse on stage, e – Design costumes and scenery, f – Prepare costumes, g – Prepare scenery, and h – Dress rehearsal, then logical expression W_L is

$$SeqSeq(a, Concur(b, Seq(c, d), Concur(e, f, g)), h)$$

From the obtained expression a logical specification L will be built in the following steps. At the beginning, the specification of a model is $L = \emptyset$. Most nested patterns are Seq and $Concur$. Sequence gives $L = L \cup \{c \Rightarrow \Diamond d, \Box\neg(c \wedge d)\}$, and then Concurrency gives $L = L \cup \{e \Rightarrow \Diamond f \wedge \Diamond g, \Box\neg(e \wedge (f \vee g))\}$. The next considered pattern is Concurrency for which the disjunction of arguments is considered $L = L \cup \{b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g), \Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g)))\}$. The most outside situated pattern gives $L = L \cup \{a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g), (b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h, \Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee ((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h))\}$.

Thus, the resulting specification contains formulas

$$\begin{aligned} L = \{&c \Rightarrow \Diamond d, \Box\neg(c \wedge d), e \Rightarrow \Diamond f \wedge \Diamond g, \\ &\Box\neg(e \wedge (f \vee g)), b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g), \\ &\Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g))), \\ &a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g), \\ &(b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h, \\ &\Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee \\ &((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h))\} \end{aligned} \tag{2}$$

The examined property can be

$$a \Rightarrow \Diamond h \tag{3}$$

which means that if the theater script is chosen then sometime in the future the dress rehearsal is completed. The whole formula to be analyzed using the semantic tableaux method is

$$\begin{aligned} &((c \Rightarrow \Diamond d) \wedge (\Box\neg(c \wedge d)) \wedge (e \Rightarrow \Diamond f \wedge \Diamond g) \wedge \\ &(\Box\neg(e \wedge (f \vee g))) \wedge \\ &(b \Rightarrow \Diamond(c \vee d) \wedge \Diamond(e \vee f \vee g)) \wedge \\ &(\Box\neg(b \wedge ((c \vee d) \vee (e \vee f \vee g)))) \wedge \\ &(a \Rightarrow \Diamond(b \vee c \vee d \vee e \vee f \vee g)) \wedge \end{aligned}$$

$$\begin{aligned}((b \vee c \vee d \vee e \vee f \vee g) \Rightarrow \Diamond h) \wedge \\ (\Box\neg((a \wedge (b \vee c \vee d \vee e \vee f \vee g)) \vee \\ ((b \vee c \vee d \vee e \vee f \vee g) \wedge h) \vee (a \wedge h)))) \\ \Rightarrow (a \Rightarrow \Diamond h) \end{aligned} \tag{4}$$

Formula 2 represents the output of the G component in Fig. 4. Formula 4 provides a combined input for the T component in Fig. 4. Presentation of a full reasoning tree for formula 4 exceeds the size of the work since the tree contains many hundreds of nodes. The formula is true and the examined property 3 is satisfied in the considered activity model.

5 Conclusions

The work presents a new approach to the formal verification of requirements models using temporal logic and the semantic tableaux method. The paper presents the method for transformation of activity diagram design patterns to logical expressions which represent a model of requirements. The algorithm for generating logical specifications from a logical expression is proposed. The advantage of the whole methodology is that it provides an innovative concept for formal verification which might be done for any requirements model created using the UML use case and activity diagrams. It enables both receiving high-quality requirements and stable software systems as well as the transformation from verified goal-oriented requirements to reliable systems.

Future work may include the implementation of the logical specification generation module and the temporal logic prover. Another possibility is an extension of the deduction engine in order to support more complex logics to compare with the minimal temporal logic. Important issues are questions how to apply the approach in the real software development process and in the industry practice. The approach should results in a CASE software providing modeling requirements and software design. The purpose of the CASE software should include both the identification of activities and actions in use case scenarios, workflow modeling in activity diagrams as well as detecting and resolving possible inconsistencies of models being designed by different designers. All the above mentioned efforts will lead to user friendly deduction based formal verification of requirements engineering and user centered software development.

References

1. 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'09). pp. 182–189 (2009)
2. van Benthem, J.: Handbook of Logic in Artificial Intelligence and Logic Programming, chap. Temporal Logic, pp. 241–350. 4, Clarendon Press (1993–95)

3. Cabral, G., Sampaio, A.: Automated formal specification generation and refinement from requirement documents. Journal of the Brazilian Computer Society 14 (1), 87–106 (2008)
4. Chellas, B.F.: Modal Logic. Cambridge University Press (1980)
5. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
6. Clarke, E., Wing, J., et al.: Formal methods: State of the art and future directions. ACM Computing Surveys 28 (4), 626–643 (1996)
7. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2001)
8. d'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J.: Handbook of Tableau Methods. Kluwer Academic Publishers (1999)
9. Emerson, E.: Handbook of Theoretical Computer Science, vol. B, chap. Temporal and Modal Logic, pp. 995–1072. Elsevier, MIT Press (1990)
10. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. IEEE Transactions on Software Engineering 30 (7), 437–447 (2004)
11. Fowler, M.: UML Distilled. Third Edition. Addison-Wesley (2004)
12. Hähnle, R.: Tableau-based Theorem Proving. ESSLLI Course (1998)
13. Hurlbut, R.R.: A survey of approaches for describing and formalizing use cases. Tech. Rep. XPT-TR-97-03, Expertech, Ltd. (1997)
14. Kazhamiakin, R., Pistore, M., Roveri, M.: Formal verification of requirements using spin: A case study on web services. In: (SEFM 2004) Proceedings of the Second International Conference on Software Engineering and Formal Methods, 28–30 September 2004, Beijing, China. pp. 406–415 (2004)
15. Klimek, R.: Introduction to temporal logic [in Polish]. AGH University of Science and Technology Press (1999)
16. Klimek, R.: Towards formal and deduction-based analysis of business models for soa processes. In: Filipe, J., Fred, A. (eds.) Proceedings of 4th International Conference on Agents and Artificial Intelligence (ICAART 2012), 6-8 February, 2012, Vilamoura, Algarve, Portugal. vol. 2, pp. 325–330. SciTePress (2012)
17. Klimek, R., Szwed, P.: Formal analysis of use case diagrams. Computer Science 11, 115–131 (2010)
18. Pelletier, F.: Semantic tableau methods for modal logics that include the b and g axioms. Tech. Rep. Technical Report FS-93-01, AAAI (Association for the Advancement of Artificial Intelligence) (1993)
19. Pender, T.: UML Bible. John Wiley & Sons (2003)
20. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley (1999)
21. Wolter, F., Wooldridge, M.: Temporal and dynamic logic. Journal of Indian Council of Philosophical Research XXVII(1), 249–276 (2011)
22. Zhao, J., Duan, Z.: Verification of use case with petri nets in requirement analysis. In: Proceedings of the International Conference on Computational Science and Its Applications: Part II. pp. 29–42. ICCSA '09, Springer-Verlag, Berlin, Heidelberg (2009)