Towards a Pattern-based Automatic Generation of Logical Specifications for Software Models

Radosław Klimek
AGH University of Science and Technology,
al. A. Mickiewicza 30, 30-059 Krakow, Poland

2013

Abstract

The work relates to the automatic generation of logical specifications, considered as sets of temporal logic formulas, extracted directly from developed software models. The extraction process is based on the assumption that the whole developed model is structured using only predefined workflow patterns. A method of automatic transformation of workflow patterns to logical specifications is proposed. Applying the presented concepts enables bridging the gap between the benefits of deductive reasoning for the correctness verification process and the difficulties in obtaining complete logical specifications for this process.
Keywords: formal methods; temporal logic; generating logical specifications; workflow patterns; deductive reasoning

1 Introduction

The deductive approach is always an essential part of the human thought process and might provide the natural support for reasoning about system correctness and guarantee a rigorous approach in software constructions. The approach also dominates scientific works and is used intuitively in everyday life. It enables the analysis of infinite computation sequences. The need to manually build a systems's logical specification seems to be a significant obstacle to the practical use of deduction-based verification tools. Hence, the challenge of automating the process of obtaining logical specifications seems to be understandable, justified and particularly important. A large class of software models can be organized and developed using workflow patterns, e.g. business models expressed in BPMN [1], or the UML activity diagrams [2]. However, the presented approach is more general and, for example, business models should not be related directly to the patterns of the work despite any literal similarity.

The main purpose of the work is to propose a kind of language for expressing software models and modeling logical specifications. The main idea is the generation of logical specifications, a process based on both workflow patterns

and their predefined temporal logic formulas which describe every pattern's behavior. The work's contribution is a method that automates the generation of logical specifications. The generation algorithm for workflow patterns is presented. The proposed method is characterized by the following advantages: introducing patterns as primitives to logical modeling, scaling up to real-world problems, 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 less skills in logic. All these factors are discussed in the work and summarized in the last section.

The considerations in the work are focused on the *propositional linear time logic* PLTL, for which syntax and semantics are defined in [3]. It should be pointed that atomic propositions could be identical to atomic activities, or tasks, of which particular patterns can be constructed. The work's considerations are justifiably limited to the minimal temporal logic, e.g. [4]. Less expressiveness of this logic, compared to more complex ones, can be compensated or counterbalanced for the fact that it is much easier to build a deduction engine, or use the existing valid provers, for this logic. For example, the deduction engine for the semantic tableaux method for the minimal temporal logic is relatively easy to build. Thus, the proposed approach can be verified more quickly than in the case of more complex logical systems. Last but not least, the work is focused on automating the process of generating logical specifications for temporal logic and minimal temporal logic seems to be the first step in the research, with more complex logics to follow.

2 Pattern-oriented logical modeling

A *pattern* is a generic description of structure of some computations, and does not limit the possibility of modeling arbitrary complex models. Every workflow pattern can be linked with a set of temporal logic formulas describing its properties (liveness and safety), i.e. expressing the behavior of tasks or activities included in a pattern.

Let us assume that syntactically correct temporal logic formulas are already defined [3].

Definition 1. *The* elementary set of formulas $pat(a_1, \ldots, a_n)$*, or* $pat()$*, build over atomic formulas* $a_1, \ldots, a_n$ *is a set of temporal logic formulas* $f_1, \ldots, f_m$ *such that all formulas are syntactically correct, i.e.* $pat() = \{f_1, \ldots, f_m\}$.

A general example for the definition is given as $pat(a, b, c) = \{a \Rightarrow \Diamond b, \Box\neg(b \wedge \neg c)\}$ which is a two-element set of LTL formulas created over three atomic formulas.

The key notion for the work is a logical pattern considered as a structure.

Definition 2. *The* logical pattern *is a structure* $Pattern = \langle ini, fin, pat() \rangle$*, where* ini *and* fin *are logical statements of classical propositional logic describing the logical circumstances of, respectively, the start and the termination of*

the whole workflow pattern execution, and pat() is an elementary set of temporal logic formulas describing the behavior of a workflow pattern.

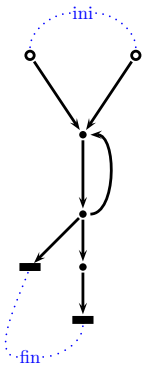

Figure 1: The illustration for *ini*- and fin-conditions for a pattern

The meaning of $pat()$ does not raise any questions. *ini* and fin describe logic circumstances associated with, respectively, the initiation/opening and termination/closing of a pattern. *ini* is satisfied when some initial activity or activities are active. For example, $a \wedge b$ for *ini* means that when the execution of a pattern is initiated then both activities a and b are satisfied. $a \vee b$ for fin means that when execution of a pattern is to be terminated, then activity a or activity b is active. In other words, *ini*- and fin-expressions (conditions) describe, respectively, the first and the last activity, or activities, of the pattern which are active. (Thus, *ini* and fin should not be confused with the well-known, pre- and post-conditions, respectively.) The idea of *ini* and fin conditions for a pattern is shown in Fig. 1. Transitions illustrate flows of control inside a pattern. *ini* and fin are expressed in classical propositional logic since they are valuated at a certain time point as point expressions.

Pattern.pat is a general notation for a set of formulas for a pattern, and in the case when a particular pattern is considered it can be written as *Seq.pat* (for the Sequence pattern), or, if it does not lead to ambiguity then *Seq*. Similarly, instead of *Seq.ini* and $Seq.fin$ can be written *ini* and fin.

Logical consistency is one of the most important properties of logical systems. A set of logical formulas is *consistent* if it does not contain contradiction. In other words, logical theory is consistent if and only if (iff) there is no situation that both Φ and $\neg\Phi$ are provable from the axioms and formulas of the theory. When building logical patterns for workflow patterns, it is assumed that logical patterns are consistent, i.e. prepared *basic formulas* $pat() = \{p_1, ..., p_n\}$ are consistent, and also all temporal formulas describing behavior/valuation over the *ini*- and fin-conditions, let us call them *transition formulas*, are consistent, e.g. $\Diamond ini$, $ini \Rightarrow \Diamond fin$, etc.

Definition 3. *The logical pattern is* consistent *iff all formulas associated with this pattern, including formulas for ini- and fin-conditions, i.e. pat()*$\cup\{\Diamond ini, ini \Rightarrow \Diamond fin, \Diamond fin, \Box\neg(ini \wedge fin)\}$ *are consistent.*

The discussed transition formulas are important and describe both safety and liveness aspect of a pattern which is considered as a whole, i.e. in terms of the *ini-* or *fin*-conditions. In certain situations, if desired, it enables abstracting from details of a pattern internal behavior and examining it as a whole. Both *ini-* and *fin*-expressions have equal rights to the pattern representation.

Corollary 1. *Every pattern contains, and its logical pattern describes, the structure consisting of at least two activities (or tasks).*

Instead of a formal proof, note that, in the general case, e.g. for a one-activity pattern, if the above statement is not satisfied then the formula $\Box\neg(ini \wedge fin)$ is also not satisfied.

Algorithm 1 Labeling logical expressions

Input: logical expression W_L (non-empty)
Output: labeled logical expression W_L'

▷ Lexical tokens are processed/scanned one by one from left to right

```
 1: l:= 0; pre:= ϵ;                          ▷ l=current label, pre=previous label
 2: for every token scanned in the whole expression do
 3:     if token="(" then
 4:         l++; replace "(" by "(l]" for the token
 5:     else if token=")" then
 6:         if pre=")" then
 7:             l−−
 8:         end if
 9:         replace ")" by "[l)" for the token
10:     end if
11:     pre:= token
12: end for
```

A symbolic notation allows for literal representation of a structure of an arbitrary complexity and including nested patterns.

Definition 4. *The* logical expression W_L *is a structure created using the following rules:*

1. *every elementary set pat*(a_i)*, where* $i > 0$ *and every* a_i *is an atomic formula, is a logical expression,*
2. *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.

Definition 5. *The* labeled logical expression W'_L *is received from a logical expression* W_L *introducing (numerical) labels into the logical expression using the Algorithm 1.*

The labeled logical expression shows directly the nested structure of patterns. The function of the *maximum label max* for an expression returns the maximum (numerical) value of the label placed in the expression.
$Seq(1]a, Seq(2]ParalSplit(3]b, c, d[3), Synchron(3]e, f, g[3)[2)[1) = w$ is an example of the labeled expression which is intuitive in that it shows the sequence that leads to a parallel split and then synchronization of some activities and the maximum label is $max(w) = 3$.

```
Seq(f1,f2):
ini= f1 / fin= f2
f1 => <>f2 / ~f1=>~<>f2
[]~(f1 & f2)
Concur(f1,f2,f3):
ini= f1 / fin= f2 | f3
f1 => <>f2 & <>f3  /  ~f1 => ~<>f2 & ~<>f3
[]~(f1 & (f2 | f3))
Branch(f1,f2,f3):
ini= f1 / fin= (f2 & ~f3) | (~f2 & f3)
f1 => (<>f2 & ~<>f3) | (~<>f2 & <>f3)
~f1 => ~<>(f1 | f2)
[]~(f2 & f3) / []~((f1 & f2) | (f1 & f3))
```

Figure 2: A sample predefined set P

Logical properties of any pattern from *predefined temporal patterns* Π are expressed in temporal logic and stored in the predefined set P, c.f. example in Fig. 2. The set is a plain ASCII text/file to indicate that it can be easy modified using a simple text editor. Most elements of the P set, i.e. two temporal logic operators, classical logic operators, are not in doubt. The slash allows to separate formulas placed in a single line. f_1, f_2 etc. are atomic formulas and constitute a kind of formal arguments for a pattern. Although this sample set P contains only three predefined temporal patterns $\Pi = \{Seq, Concur, Branch\}$, i.e. sequential order of activities, and fork of control to enable concurrent execution of activities, and conditional execution of activities, there is no difficulty with defining a set of elementary formulas for other workflow patterns. The considerations of this work are limited to three patterns to present the main idea of the work which is the pattern-based generation of logical specifications.

Let us supplement Definition 2, which provides information about the partial order ($\preceq$) for the set of atomic formulas as arguments of a pattern. It consists of three subsets which are pairwise disjoint: the *ini* arguments (at least one element), ordinary arguments (may be empty), and the fin arguments (at least one element). $f_i \preceq f_j$ iff $f_i \prec f_j$ or $f_i = f_j$. $f_i = f_j$ iff both f_i and f_j belong exclusively to only one of the three subsets. $f_i \prec f_j$ iff f_i belongs to *ini* arguments and f_j belongs to ordinary arguments, or f_i belongs to ordinary arguments and f_j belongs to fin arguments. All the *ini* arguments (and no others) form the *ini*-expression. All the fin arguments (and no others) form

the fin-expression. None of the ordinary arguments are included either in the ini- or fin-expression.

One may need to calculate the consolidated ini- and fin-expressions to obtain expressions for complex and nested patterns.

Definition 6. *Let w^c for a logical expression w with the upper index $c = i$ (or f, respectively) be the* consolidated ini-expression *(or the* consolidated fin-expression*, respectively). The consolidated expression is calculated using the following (recursive) rules:*

1. *if there is no pattern itself in the place of any atomic argument which syntactically belongs to the ini-expression (or the fin-expression, respectively) w, then w^i is equal to $w.ini$ (w^f is equal to $w.fin$, respectively),*

2. *if there is a pattern, say $t()$, in a place of any atomic argument, say r, which syntactically belongs to the ini-expression (or the fin-expression, respectively) of w, then r is replaced by t^i (or t^f, respectively) for every such case.*

Examples of consolidated expressions with regard to the patterns from Fig. 2 are given as follows. For $w = Seq(a, b)$ conditions are $w^i = a$ and $w^f = b$ (step 1). For $w = Concur(a, Seq(b, c), d)$ conditions are $w^i = a$ (step 1) and $w^f = c \vee d$ (step 2 gives $Seq^f =$"c" and step 1 gives "$\vee d$", which is consolidated to "$c \vee d$"). For $w = Concur(a, Seq(b, Concur(c, d, e)), f)$ conditions are $w^i = a$ and $w^f = (d \vee e) \vee f$ (step 2 gives Seq^f and step 1 gives "$\vee f$", the next step 2 for Seq^f leads to $Concur^f =$"$(d \vee e)$", and after consolidation "$(d \vee e) \vee f$" is obtained).

3 Generation of logical specifications

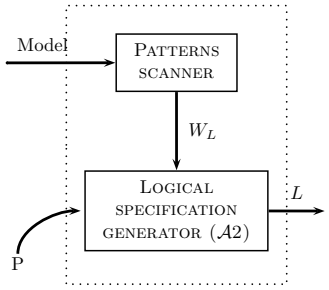

Figure 3: System for generating logical specifications

Generating logical specifications requires an algorithm that converts a logical expression to the target logical specification.

Definition 7. Logical specification L *consists of all formulas derived from a logical expression* W_L *using the Algorithm 2 (shortly* $\mathcal{A}2$*), i.e.* $L(W_L) = \{f_i : i \geq 0 \wedge f_i \in \mathcal{A}2(W_L, P)\}$*, where* f_i *is a PLTL formula.*

Generating a logical specification is not a simple summation of formula collections resulting from the components of a logical expression. The architecture of the whole proposed system is shown in Fig. 3. The generation algorithm $\mathcal{A}2$ has two inputs. The first is a logical expression W_L (Def. 4) which is a kind of variable in that it varies for every (workflow) model. The second is a predefined set P (Fig. 2) which is a kind of constant in that it is predefined for a certain class of software models. The output of the generation algorithm $\mathcal{A}2$ is a logical specification L (Def. 7) understood as a set of temporal logic formulas.

Algorithm 2 Generating logical specifications

Input: Logical expression W_L (non-empty), predefined set P (non-empty)
Output: Logical specification L

1: $L := \emptyset$ ▷ initiating specification
2: **for** l:= $max(W'_L)$ **to** 1 **do**
3: p:= getPat(W'_L, l); ▷ get the first (lefmost) pattern with the label l
4: **repeat**
5: **if** pattern p consists only atomic formulas **then**
6: $L := L \cup p.pat()$
7: **end if**
8: **if** any argument of the p is a pattern itself **then**
9: Specification L' for every combination $C_{i=1,\ldots,n}$, i.e. $L'(C_i)$, are
10: calculated considering *ini*- and fin-expressions for every non-atomic
11: arguments and substituting consolidated expressions in places
12: of these patterns as arguments, i.e. $L := L \cup L'(C_i)$
13: **end if**
14: p:= getPat(W'_L, l) ▷ get the next pattern with the label l
15: **until** p is empty
16: **end for**

When the Algorithm 2 is analyzed, it seems that lines 8 to 13 need some clarification. For example, the consideration of the logical expression $p(a, q(), r(), d)$, where only a and d are atomic formulas, leads to the following combinations $C_{i=1,\ldots,4}$: $p(a, q^i(), r^i(), d)$, $p(a, q^i(), r^f(), d)$, $p(a, q^f(), r^i(), d)$, and $p(a, q^f(), r^f(), d)$. Let us supplement the $\mathcal{A}2$ by some other examples. For lines 5–7: $Branch(a, b, c)$ gives $L = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \neg a \Rightarrow \neg\Diamond(b \vee c), \Box\neg(b \wedge c), \Box\neg((a \wedge b) \vee (a \wedge c))\}$. The example for lines 8–13: $Concur(Seq(a, b), c, d)$ leads to $L = \{a \Rightarrow \Diamond b, \neg a \Rightarrow \neg\Diamond b, \Box\neg(a \wedge b)\} \cup \{a \Rightarrow \Diamond c \wedge \Diamond d, \neg a \Rightarrow \neg\Diamond c \wedge \neg\Diamond d, \Box\neg(a \wedge (c \vee d))\} \cup \{b \Rightarrow \Diamond c \wedge \Diamond d, \neg b \Rightarrow \neg\Diamond c \wedge \neg\Diamond d, \Box\neg(b \wedge (c \vee d))\}$, where the first set results from the nested pattern, the second results from the use of the *ini*-expression considered as an argument instead of the nested pattern, and the third results from the use of the fin-expression also considered as an argument instead of the nested pattern. Thus, the final specification is $L = \{a \Rightarrow \Diamond b, \neg a \Rightarrow \neg\Diamond b, \Box\neg(a \wedge b), a \Rightarrow \Diamond c \wedge \Diamond d, \neg a \Rightarrow \neg\Diamond c \wedge \neg\Diamond d, \Box\neg(a \wedge (c \vee d)), b \Rightarrow \Diamond c \wedge \Diamond d, \neg b \Rightarrow \neg\Diamond c \wedge \neg\Diamond d, \Box\neg(b \wedge (c \vee d))\}$.

The basic question is always consistency of logic specifications.

Theorem 1. *Supposing that every pattern of the P set is non-empty and consistent and every pair of patterns has disjointed set of atomic formulas then the logical specification obtained for the $\mathcal{A}2$ algorithm is consistent.*

Proof. Let us consider four cases each of which refers to the important points of the Algorithm 2.
Line 1. Immediately due to the initiation of L.
Line 2. Patterns processing order does not affect the consistency of a logical specification since it itself does not add new formulas to the logical specification.
Lines 5–7. Due to the assumptions of the disjointedness of atomic formulas for patterns and the disjointedness of patterns in a logical expression (note that any two patterns are either completely disjointed or completely contained one in the other), and also the assumption of the predefined patterns consistency, c.f. Definition 3, it is not possible to introduce contradictions when adding a new elementary set of formulas.
Lines 8–13. Due to the consistency of transition and basic formulas, c.f. Definition 3, as well as syntax, and properties, and consistency of transition formulas, the newly-generated logical specification is consistent, it follows from the fact that new temporal formulas for ini- and fin-conditions refer to the consistent transition formulas of a pattern. □

4 Conclusions

The proposed method of generating enables scaling up, that is migration from small problems, e.g. comprising a single UML activity diagram and its workflow, or a single pool for business processes, to real-world problems with more and more diagrams and pools with nesting patterns. This gives hope for practical use in the case of problems of any size. Another advantage for the idea of logical patterns is the fact that they are once well-defined and could be widely and commonly used by an inexperienced user. Then, it does not necessarily require in-depth knowledge of the complex aspects of temporal logic by an ordinary user. Introducing patterns as logical primitives foreshadows progress in building logical specifications that breaks some barriers and obstacles mentioned in the work for practical use of deduction-based formal methods.

Further works may include: other logical properties of the approach, more enriched logics, different workflow patterns, and different classes of applications.

References

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

[2] R. Klimek, From extraction of logical specifications to deduction-based formal verification of requirements models, in: R. Hierons, M. Merayo,

M. Bravetti (Eds.), Proceedings of 11th International Conference on Software Engineering and Formal Methods (SEFM 2013), 25–27 September 2013, Madrid, Spain, Vol. 8137 of Lecture Notes in Computer Science, Springer Verlag, 2013, pp. 61–75.

[3] E. Emerson, Handbook of Theoretical Computer Science, Vol. B, Elsevier, MIT Press, 1990, Ch. Temporal and Modal Logic, pp. 995–1072.

[4] J. van Benthem, Handbook of Logic in Artificial Intelligence and Logic Programming, 4, Clarendon Press, 1993–95, Ch. Temporal Logic, pp. 241–350.