AUTOMATYKA/ AUTOMATICS • 2013 • Vol. 17 • No. 1

http://dx.doi.org/10.7494/automat.2013.17.1.59

Radosław Klimek*

From Workflow Design Patterns to Logical Specifications

1. Introduction

Formal methods in software development might provide the natural and intuitive support for reasoning about system correctness and guarantee a rigorous approach in software constructions. Formal specification and formal reasoning are two important and closely related parts of formal approach. Formal specification establish fundamental system properties and invariants.

Formal inference enables reliable verification of desired properties. States exploration and deductive inference are two basic approaches to formal reasoning about information systems. They both are well-established but during recent years spectacular and significant progress in the field of states exploration, i.e. model checking, was made and it wins on points with the deductive approach. However, model checking is rather operational than analytic approach and is a kind of simulation for all reachable paths of computation. Deductive inference is always the most natural for human beings and is used intuitively in everyday life.

It seems that there are two reasons why formal inference is far behind states exploration and one of these is a key issue. The important question is the choice of a deductive system but the key question is a lack of a method for obtaining a logical specification of a system as a set of temporal logic formulas and above all the automation of this process. Necessity to specify a large collection of temporal logic formulas as a system specification is difficult and monotonous process especially for an inexperienced user. It can be a significant obstacle to the practical use of deduction-based verification tools. Hence, the need to automate the process of obtaining a logical specification seems to be understandable, justified and particularly important. The so-obtained logical specification can be used in the process of formal reasoning in temporal logic, for example using resolution methods or semantic tableaux methods.

* AGH University of Science and Technology, Krakow, Poland

When considering semantic tableaux method, the general form of an analyzed formula can be $P \Rightarrow Q$, where Q is a verified system property, or more precisely $p_1 \wedge ... \wedge p_n \Rightarrow Q$, where every p_i is a formula extracted from a model. All formulas p_1, ..., p_n constitute a logical specification. When n is large, it is not possible in practice to build a logical specification manually and therefore this process should be subjected to automation. A large class of models with plain control flows is considered here. They relatively easy lead to the generation process since they represent a kind of logical network of tasks and activities and informally speaking their control flows are not disturbed by data. Good examples are BPMN models of business processes or UML activity diagrams. The main ideas of this work and in fact the generation process of specification are based on the assumption that corresponding software models are developed using only design patterns. *Design patterns* are abstraction forms of reusable solutions and there are already defined 23 patterns for business models [3] and is not difficult to define patterns for the UML activity diagrams.

Motivation and contribution

The motivation for this work is a lack of tools for the automatic extraction of logical specifications from software models, i.e. obtaining a set of temporal logic formulas. The contribution of this work is a method for automating the generation process of logical specifications. Theoretical possibilities of such an automation are discussed and the generation algorithm for some design patterns is presented.

2. Generation of specifications

Temporal logic TL is a well established and broadly used formalism for specification and verification. It exists in many varieties, however, considerations in this paper are limited to the *linear time temporal logic* LTL, and to the *smallest*, or *minimal*, *temporal logic* ([1]), also known as temporal logic of the class K. The following formulas may be considered as examples of this logic: $act \Rightarrow \Diamond rec$, $\Box(sen \Rightarrow \Diamond ack)$, $\Diamond liv$, $\Box\neg(evn)$, etc. Although the logic K is sufficient to define most of system properties, the time structure may be enriched until, for example, the logic KDT4 is received ($\Box p \Rightarrow \Diamond p$, $\Box p \Rightarrow p$, $\Box p \Rightarrow \Box\Box p$, $\Diamond\Diamond p \Rightarrow \Diamond p$, etc.) which seems to have properties very close to our intuition of time. However, this logic is not considered here and could be the next step in the research.

Suppose well-formed and syntactically correct temporal logic formulas are defined [2]. On the other hand design patterns which are associated with temporal logic formulas describing their liveliness and safety properties are considered.

Definition 1. *An* elementary set *of formulas over atomic formulas* $a_{i,\, i = 1, ..., n}$, *what is denoted* $pat(a_i)$, *or simply* $pat()$, *is a set of temporal logic formulas* f_1, ..., f_m *such that all formulas are well-formed.*

(It is recalled that all formulas are restricted to the logic K.) The proposed temporal logic formulas should describe both safety and liveness properties of each pattern. In this

way, as an example, $Seq(a, b) = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$ describes property of the Sequence pattern. Set $Concur(a, b, c) = \{a \Rightarrow \Diamond b \wedge \Diamond c, \Box\neg(a \wedge (b \vee c))\}$ describes the Concurrency pattern and $Branch(a, b, c) = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$ the Branching pattern. The meaning of the above patterns is intuitive.

Every developed model is designed using only the predefined design patterns of workflows. The whole workflow model can be quite complex including nesting patterns and this is why there is a need to define a symbolic notation which enables to represent any potentially complex structure.

Definition 2. *The* logical expression*WL 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* a_j*, where* $j > 0$*, or*
  - *a set* $pat(a_j)$*, where* $j > 0$ *and* a_j *is an atomic formula, or*
  - *a logical expression* $pat(A_j)$*, where* $j > 0$

  is also a logical expression.

Any logical expression may represents an arbitrary structure of design patterns and an example of this is $Seq(a, Seq(ParalSplit(b, c, d), Synchron(e, f, g)))$ meaning of which is intuitive, i.e. it shows the sequence that leads to a parallel split and then synchronization of some activities.

Properties of all potentially used design patterns are expressed in temporal logic and stored in the set *P*, which is predefined and fixed. Below is an example of such a set *P*.

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

Most elements of the *P* set, i.e. two temporal logic operators, classical logic operators, are not in doubt. f_1, f_2 etc. are atomic formulas and constitute a kind of formal arguments for a pattern. Although the above set contains only three patterns and their formulas, i.e. {*Seq*, *Concur*, *Branch*}, there is no difficulty with defining a set of elementary formulas for any design pattern, for example, for the 23 patterns in work [3], and also for the UML activity diagram design patterns.

The last step is to define a logical specification which is generated from a logical expression.

Definition 3. *The* logical specification *L consists of all formulas derived from a logical expression using the algorithm* Π*, i.e.* $L(W_L) = \{f_i : i > 0 \wedge f_i \in \Pi(W_L, P)\}$*, where* f_i *is a TL formula.*

Generating a logical specification is not a simple summation of formulas collections resulting from a logical expression. The sketch of the generation algorithm is presented below. The generation process 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. The second one is a predefined set *P* which is a kind of constant.

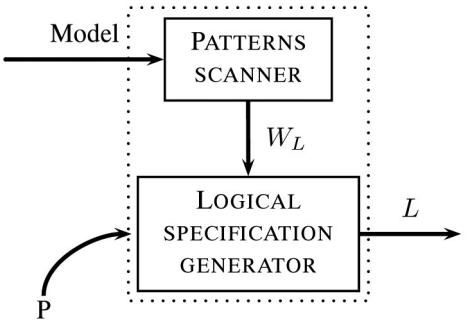

Fig. 1. System for generating logical specifications

The output of the generation algorithm π is a logical specification understood as a set of temporal logic formulas. The architecture of the whole system is shown in Figure 1 and the sketch of the generation algorithm is given below.

Algorithm 1 (Sketch)

1. *At the beginning, the logical specification is empty, i.e.* $L := \emptyset$;
2. *patterns are processed from the most nested pattern to the located more towards the outside and from left to right*;
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, 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 a place of the pattern*.

All patterns of the logical expression are processed one by one and the algorithm always halts. All parentheses are paired. Let us supplement the algorithm by some examples. The example for step 3: *Seq*(*a*, *b*), gives $L = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\}$ and *Branch*(*a*, *b*, *c*) gives $L = \{a \Rightarrow (\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond b \wedge \Diamond c), \Box\neg(b \wedge c)\}$. The example for step 4: *Concur*(*Seq*(*a*, *b*), *c*, *d*) leads to $L = \{a \Rightarrow \Diamond b, \Box\neg(a \wedge b)\} \cup \{(a \vee b) \Rightarrow \Diamond c \wedge \Diamond d, \Box\neg((a \vee b) \wedge (c \vee d))\}$.

3. Conclusion

The method for an automatic generation of logical specifications from software models based on design patterns is proposed. This specification is a set of temporal logic formulas and obtaining it is crucial in the case a formal verification based on a deductive approach. The minimal temporal logic is considered. Further works may involve both enriched logics and defining other design patterns.

References

[1] Chellas B.F., *Modal Logic*. Cambridge University Press, 1980.
[2] Emerson E.A., *Handbook of Theoretical Computer Science*, vol. B, chapter Temporal and Modal Logic, pp. 995–1072. Elsevier, MIT Press, 1990.
[3] Aalst W.M.P. van der, ter Hofstede A.H.M., Kiepusewski B., Barros A.P. *Workflow patterns*. Distributed and Parallel Databases, 4(1), 2003, pp. 5–51.