View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Supporting requirements formulation in software formal verification

José Creissac Campos
Departamento de Informática/CCTC
Universidade do Minho
jose.campos@di.uminho.pt

José Machado
Departamento de Eng[a] Mecânica/CT2M
Universidade do Minho
jmachado@dem.uminho.pt

Abstract

Formal verification tools, such as model checkers, have reached a stage were their applicability in the design and development of dependable and safety critical systems has become viable. While the formal verification step in tools such as model checkers is fully automated, writing appropriate models and properties is a skillful process. In particular, a correct understanding of the logics used to express properties is needed to guarantee that properties correctly encode the original requirements. In this paper we present a properties editor tool which can help in simplifying the process of encoding requirements as logical formulae for verification purposes.

1. Introduction

Formal verification is becoming established as a useful and powerful technique for guaranteeing the correctness of software. Tool support for formal verification must now go beyond the actual verificatio step and address issues from the editing of models and properties, to helping the interpretation of verification results. One specific aspect that deserves attention is the writing of properties to be verified.

Properties must encode relevant requirements of the system. The formalization of requirements is still pretty much an open issue. Going from a written account of system requirements to a formal description of those requirements, is no easy task. In this paper we do not address that problem directly. Instead, we take a pragmatics approach, and look at how to facilitate the writing of the formal properties. To that end, we are developing a tool that enables engineers to obtain correct properties formalization, even with limited knowledge of the logic in which properties are expressed.

2. Property specification patterns

Expressing properties in a formal language can be a complex task. As discussed, in order to address this, a tool to help property creation is being developed. The tool is based on the notion of property patterns.

A pattern is a means to capture proven solutions to known problems, and demonstrate how they can be used in practice to solve the same or similar problems in new situations [1]. Dwyer et al. [2] carried out an extensive review of published property specifications, and proposed a system of property specification patterns. Each pattern features a description, including the pattern's intent, usage examples, relationships to other patterns, and mappings to different logics (e.g., LTL and CTL [3]). Campos and Machado [4] carried out a similar study for the area of Discrete Event Systems (DES). They found that many of Dwyer et al.'s patterns can be applied, but also found a number of new patterns occurring in the DES literature. In particular, they identified the need to restrict the formulation of the properties to a subset of all the states in the model (e.g. to consider stable states in the model only [5]).

An example of a pattern is the "Precedence" pattern which implies the requirement that some event or state P must occur before some other event or state Q. Its formulation in CTL is:

$$A[\neg Q \ W \ P]$$

which is read as: it is always the case (A) that Q cannot happen ($\neg$Q), until (W) P happens.

Typically, however, model checkers (c.f., NuSMV), do not support the week until operator (W). Hence, the property above must in practice be expressed as:

$$\neg E[\neg P \ U \ (Q \wedge \neg P)]$$

which states the same but in a rather more convoluted manner: it is never the case that P will not happen before Q.

Figure 1 shows an extract of the Precedence pattern. As can be seen, properties get more complex when scoping and stable states are considered.

3. The Patterns Editor tool

Using the patterns mentioned above implies first selecting the most appropriate patterns, and then instantiating it with appropriate values. This can be an error

prone process. The Properties Editor tool (see Figure 2) supports both aspects of the problem.

Property Pattern: *Precedence*

Intent: To express that some event or state P must occur before some other event or state Q. Conceptually this pattern is the opposite of the response pattern. Notice that the pattern is defined for the current state only. If needed it can be combined with the universality pattern.

Basic Formulation:

CTL: $A[\neg Q \; W \; P]$

LTL: $\neg Q \; W \; P$

Whatever the system behavior, P will always happen before Q happens.

Scoping (CTL)

After S_t: $A[\neg S_t \; W \; (S_t \wedge A[\neg Q \; W \; P])]$

After S_t until S_p: $AG((S_t \wedge \neg S_p) \rightarrow A[\neg Q \; W \; (P \vee S_p)])$

Scoping (LTL)

After S_t: $G(\neg S_t) \vee F(S_t \wedge (\neg Q \; W \; P))$

After S_t until S_p: $G((S_t \wedge \neg S_p) \rightarrow [\neg Q \; W \; (P \vee S_p)])$

Stable Formulation

CTL: $A[\neg (stable \wedge Q) \; W \; (stable \wedge P)]$

LTL: $\neg(stable \wedge Q) \; W \; (stable \wedge P)$

P always precedes Q in stable states.

Examples: (…)

Figure 1. An example pattern

On the left hand side, the tool presents the available patterns organized hierarchically. Users are able to browse the available patterns, and investigate the intent and known usages of each of them in order to select the appropriate pattern for their needs. Additionally, users can select the scope and logic to use when generating the property. On the right side, information about the pattern is provided, and users can define instantiations for the variables in the pattern. The resulting property is generated at the bottom right.

In the example in the figure, the goal was to express that the system should reach some predefined temperature before a specific valve could be open. After inspecting the pattern collection, the Precedence pattern mentioned above was chosen. In this case, its stable formulation was chosen. The "reaching desired temperature" and "opening valve" events are represented by the *s_temp_ok* and *s_open_v* variables, respectively. Hence, these variables were used to instantiate parameters P and Q. The resulting CTL formula is:

$$\neg E[\neg(stable \wedge s_temp_ok) \; U \; (stable \wedge s_open_v \wedge \neg s_temp_ok)]$$

Notice that no CTL was actually written by the user. He had only to select the appropriate pattern and indicate appropriate instantiations for its parameters.

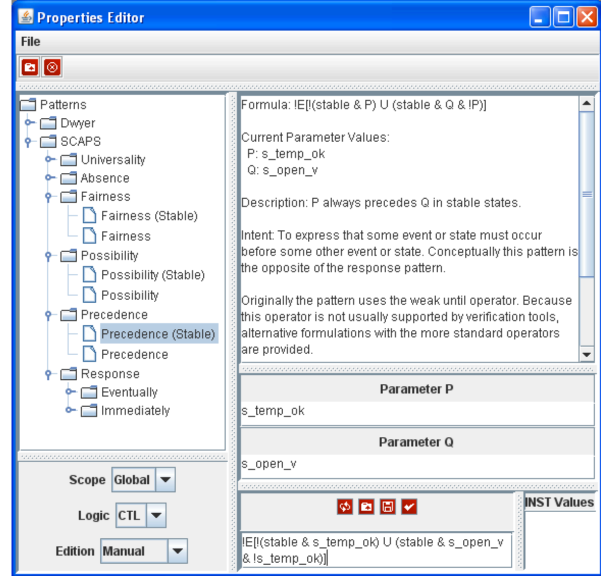

Figure 2. The Property Editor tool

4. Conclusions

In this paper we have looked at the issue of supporting the expression of property specifications. Tool support is being developed to address this issue. Together with its pattern collection, the Property Editor tool allows the expression of complex properties using basic knowledge of propositional logic only.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[2] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. "Patterns in property specification for finite-state verification". In *Proc. ICSE'98*, pp 411–420. IEEE Computer Society Press, 1998.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] J. C. Campos, J. Machado, and E. Seabra. "Property patterns for the formal verification of automated production systems". In *Proc. 17th IFAC World Congress*, pp 5107–5112. IFAC, 2008.

[5] J.Machado, B.Denis, and J.-J. Lesage. A generic approach to build plant models for DES verification purposes. In *Proc. WODES'06*, pp 407–412, July 2006.