The Journal of Systems and Software 85 (2012) 1915–1929

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Validated templates for specification of complex LTL formulas

Salamah Salamah[a,*], Ann Gates[b], Vladik Kreinovich[b]

[a] Department of Electrical, computer, Software, and Systems Engineering, Embry Riddle Aeronautical University, 600 S. Clyde Morris Blvd., Daytona Beach, FL 32114, United States
[b] Department of Computer Science, University of Texas at El Paso, 500 W. University Blvd., El Paso, TX 79968, United States

ARTICLE INFO

Article history:
Received 3 January 2011
Received in revised form 7 February 2012
Accepted 19 February 2012
Available online 7 March 2012

Keywords:
Formal specifications
LTL
Pattern
Scope
Composite propositions
Model checking

ABSTRACT

Formal verification approaches that check software correctness against formal specifications have been shown to improve program dependability. Tools such as Specification Pattern System (SPS) and Property Specification (Prospec) support the generation of formal specifications. SPS has defined a set of patterns (common recurring properties) and scopes (system states over which a pattern must hold) that allows a user to generate formal specifications by using direct substitution of propositions into parameters of selected patterns and scopes. Prospec extended SPS to support the definition of patterns and scopes that include the ability to specify parameters with multiple propositions (referred to as composite propositions or CPs), allowing the specification of sequential and concurrent behavior. Prospec generates formal specifications in Future Interval Logic (FIL) using direct substitution of CPs into pattern and scope parameters. While substitution works trivially for FIL, it does not work for Linear Temporal Logic (LTL), a highly expressive language that supports specification of software properties such as safety and liveness. LTL is important because of its use in the model checker Spin, the ACM 2001 system Software Award winning tool, and NuSMV. This paper introduces abstract LTL templates to support automated generation of LTL formulas for complex properties in Prospec. In addition, it presents formal proofs and testing to demonstrate that the templates indeed generate the intended LTL formulas.

1. Introduction

Today more than ever, society depends on complex software systems to fulfill personal needs and to conduct business. Software is an integral part of many mission and safety critical systems. In transit systems, for example, software is used in railway signaling, train control, fault detection, and notification systems among other things (Lin, 2004). Implanted drug delivery pumps, pacemakers and defibrillators, and automated cancer cell and DNA-based diagnostics systems are examples of medical equipment built around embedded software systems (MacKenzie, 1999). The National Aeronautics and Space Administration (NASA) Space Shuttle program, a multi-billion dollar program built on computer software and hardware, is an example of a safety critical system that could lead to the loss of life and huge finances if it fails.

Because of society's dependence on computers, it is vital to assure that software systems behave as intended. The estimated cost due to software errors in the aerospace industry alone was \$6 billion in 1999 (The Economic Impacts, 2002). The numbers are even more alarming when considering that software errors cost U.S. economy \$59.5 billion annually (NIST, 2002). It is imperative that the software industry continue to invest in software assurance approaches, techniques, and tools.

Although the use of formal verification methods such as model checking (Holzmann, 2004), theorem proving (Rushby, 2000), and runtime monitoring (Stolz and Bodden, 2005) has been shown to improve the dependability of programs, software development professionals have yet to adopt them. The reasons for this hesitance include the high level of mathematical sophistication required for reading and/or writing formal specifications needed for the use of these approaches (Hall, 1990).

Linear Temporal Logic (LTL) (Manna and Pnueli, 1991) is a prominent formal specification language that is highly expressive and widely used in formal verification tools such as the model checkers SPIN (Holzmann, 2004) and NuSMV (Cimatti et al., 1999). LTL is also used in the runtime verification of Java programs (Havelund and Pressburger, 2000).

Formulas in LTL are constructed from elementary propositions and the usual Boolean operators for *not*, *and*, *or*, *imply* ($\neg$, $\wedge$, $\vee$, $\rightarrow$, respectively). In addition, LTL provides the temporal operators *next* (X), *eventually* ($\Diamond$), *always* ($\Box$), *until*, (U), *weak until* (W), and *release* (R). These formulas assume discrete time, i.e., states $s = 0, 1, 2, \ldots$ The meaning of the temporal operators is straightforward[1]:

* Corresponding author. Tel.: +1 3862264919; fax: +1 3862266678.
E-mail address: salamahs@erau.edu (S. Salamah).

[1] In this work we only consider the first four of these operators.

0164-1212/\$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2012.02.041

- The formula Xp holds at state s if p holds at the next state $s+1$,
- the formula $p\,U\,q$ holds at state s, if there is a state $s' \geq s$ at which q is true and, if s' is such a state, then p is true at all states s_i for which $s \leq s_i < s'$,
- the formula $\Diamond p$ holds at state s if p is true at some state $s' \geq s$, and
- the formula $\Box p$ holds at state s if p is true at all states $s' \geq s$.

One problem with LTL is that, when specifying software properties, the resulting LTL expressions can become difficult to write and understand. For example, consider the following LTL specification: $\Box(a \rightarrow \Diamond(p \wedge \Diamond((\neg a) \wedge \neg p)))$, where a denotes "Train approaches the station." and p denotes "Train passes the station." It is not immediately obvious that the specification describes the following: "If a train approaches the station, then the train will pass the station and, after it passes, the train does not approach or pass the station."

To assist users in the generation of LTL specifications, Dwyer et al. (1999) (http://patterns.projects.cis.ksu.edu/) defined a set of patterns to represent the most commonly used formal properties. The work also defined a set of scopes of system execution where the pattern of interest must hold. Each pattern and scope combination can be mapped to specifications in multiple formal languages including LTL. Using the notions of patterns and scopes a user can define system properties in LTL without being an expert in the language. Section 2 provides more details on SPS' patterns and scopes.

In SPS, patterns and scopes parameters are defined using atomic propositions (i.e., each pattern and scope parameter is defined using a single proposition with a single truth value). To extend the expressiveness of SPS, Mondragon et al. (2004) and Mondragon and Gates (2004) developed the Property Specification (Prospec) tool. Prospec attempts to extended SPS through the definition of a set of composite propositions (CP) classes with the intent of using these to define pattern and scope parameters. A complete description of Mondragon's composite proposition classes can be found in Section 2.

Although SPS provides LTL formulas for basic patterns and scopes (ones that use single, "atomic", propositions to define parameters) and Mondragon and Gates (2004) provided LTL semantics for the CP classes as described in Table 1. below, in most cases it is not adequate to simply substitute the LTL description of the CP class into the basic LTL formula for the pattern and scope combination. We delay the introduction of a formal example of this inadequacy after Section 2 where we describe the notions of pattern, scope, and CP in more details.

This work aims at creating high-level LTL templates that can be used to define LTL formulas for complex system properties. These LTL templates take as an input a combination of pattern, scope, and CP classes that describe the desired property. The output of the templates is an LTL formula that can be used by formal verification tools such as model checkers. However, in order to be able to combine patterns, scopes, and CP classes to generate LTL formulas, we first need to provide a precise definition of the semantics of each pattern and scope when used in conjunction with CP classes and vice versa. Providing these formal definitions is a secondary goal of this paper.

The rest of the paper is organized as follows; Section 2 provides the background of the work including SPS' patterns and scopes, as well as a more detailed description of the CP classes introduced by Mondragon. Section 2 also includes an example to show the problems that can arise when using direct substitution within LTL. In Section 3 we provide a formal definition of the meaning of patterns and scopes when defined using CP classes. Section 4 introduced a new LTL operators that will be used to simplify the abstract LTL templates. Those LTL templates are described in Section 5 along with an example of their use. In Section 6 we show the methods we used to validate that the LTL templates generate LTL that meet the original meaning of the selected pattern, scope, and CP combination. The paper concludes with summary and future work followed by the references.

2. Background

This section provides the background information needed for the rest of the paper. We describe the notions of patterns, scopes as defined by Dwyer et al. (1999). We also describe Mondragon's CP classes as well as provide a more formal description of these classes. These formal descriptions of CP classes are necessary for describing the semantics of patterns and scopes that use CP classes, which we introduce in Section 3.

2.1. Specification pattern system

Writing formal specifications, particularly those involving time, is difficult. The Specification Pattern System (SPS) (Dwyer et al., 1999) provides patterns and scopes to assist the practitioner in formally specifying software properties. *Patterns* capture the expertise of developers by describing solutions to recurrent problems. Each pattern describes the structure of specific behavior and defines the pattern's relationship with other patterns. Patterns are associated with scopes that define the portion of program execution over which the property holds.

The main patterns defined by SPS are: *Universality*, *Absence*, *Existence*, *Precedence*, and *Response*. The descriptions given below are taken verbatim from the SPS website (http://patterns.projects.cis.ksu.edu/).

- *Absence*(P): To describe a portion of a system's execution that is free of certain event or state (P).
- *Universality*(P): To describe a portion of a system's execution which contains only states that have the desired property (P). Also known as Henceforth and Always.
- *Existence*(P): To describe a portion of a system's execution that contains an instance of certain events or states (P). Also known as Eventually.
- *Precedence*(P, Q): To describe relationships between a pair of events/states where the occurrence of the first (Q) is a necessary pre-condition for an occurrence of the second (P). We say that an occurrence of the second is enabled by an occurrence of the first.
- *Response*(P, Q): To describe cause-effect relationships between a pair of events/states. An occurrence of the first (P), the cause, must be followed by an occurrence of the second (Q), the effect. Also known as Follows and Leads-to.

In SPS, each pattern is associated with a *scope* that defines the extent of program execution over which a property pattern is considered. There are five types of scopes defined in SPS: *Global*, *Before R*, *After L*, *Between L And R*, and *After L Until R*. *Global* denotes the entire program execution; *Before R* denotes the execution before the first time the condition R holds; *After L* denotes execution after the first time L holds; *Between L And R* denotes the execution between intervals defined by L and R; and *After L Until* denotes the execution between intervals defined by L and R and, in the case when R does not occur, until the end of execution.

The SPS website provides patterns and scopes for formal specification languages such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL), and Graphical Interval Logic (GIL). These formulas are provided for patterns and scopes involving *single (atomic) propositions*, i.e., patterns and scopes in which P, Q, L, and R each of which occur at a single state of execution. The website also provides examples of properties that can be defined using these patterns and scopes. For example, the property "When a connection is made to the SMTP server, all queued messages in