Enhancing Property Specification Tools With Validation Techniques

Salamah Salamah, Matthew Del Buono, Eric Baily, Sarah Printy, Derek Ferris, and Laurel Christian
Embry-Riddle Aeronautical University, Daytona Beach, Florida
salamahs, delbu9c1, baile6d4, print4a4 , ferri91f, chris819@erau.edu

Abstract

Although formal approaches to software assurance such as runtime monitoring and model checking have been shown to improve system dependability, software development professionals have yet to adopt them. Among the reasons for this hesitance are the difficulty for clients and developers to write and validate formal specifications required by these approaches, and the lack of maturity of tools that can support the use of formal methods in a software development environment. This paper describes the Temporal Validator (TV) tool that assists in analyzing formulas in Linear Temporal Logic (LTL), which is one of the most widely used specification languages. The tool enhances the ability of users to validate generated formal specifications against their original intent in order to discover subtle errors. The paper also describes how the TV tool can be incorporated into the Property Specification (Prospec) tool to enhance users’ ability to generate formal specifications that match his/her interpretations.

1 Introduction

Formal approaches to software assurance require description of behavioral properties of the software system, generation of formal specifications for the identified properties, validation of the generated specifications, and verification of system’s adherence to the formal specification. The effectiveness of the assurance approach depends on the quality of the formal specifications. A major impediment to the use of formal approaches in software development remains the difficulty associated with the development of correct formal specifications (i.e., ones that match the specifier’s original intent) [5, 6].

Currently, there exists multiple formal specification languages that can be used in a variety of verification techniques and tools. Linear Temporal Logic (LTL) [10], Computational Tree Logic (CTL) [9], and Meta Event Definition Language (MEDL) [8] are some of these languages. The aforementioned languages can be used in a variety of verification techniques and tools. For example, the model checkers SPIN [7] and NuSMV [1] use LTL to specify properties of software and hardware systems. On the other hand, the SMV [2] model checker verifies system behaviors against formal properties in CTL. MEDL is used by JavaMac in runtime monitoring of java programs [8].

Formal languages have varying expressive powers, and as such, require a strong foundation in logic and mathematics for one to identify what can and cannot be expressed in a specific language. In addition, formal languages, by their nature, are hard to write, read, and validate. This problem is compounded if requirements must be specified in more than one formal language, which frequently is the case if more than one verification tool is used. Some research efforts continue to evolve to address the issue of automatic generation of formal specification by developers or customers who are not immersed in logic or the specification language of preference.

This paper introduces the Temporal Validator (TV) tool that can be used for validating formal specifications written in LTL. The tool was developed by students in the introductory course in Software Engineering at Embry-Riddle Aeronautical University in Daytona Beach. TV allows users to examine formal specifications (written in LTL) against traces of computations that represent different possible behaviors of the system being verified. The tool is intended to be part of the Property Specification (Prospec) tool [4, 12].

The paper is organized as follows; Section 2 provides a brief description of LTL and the Prospec tool. Section 3 provides the motivation of the work and the TV tool, while Section 4 introduces the tool and the ways it can be used. Finally, a summary of the work and future goals are presented.

2 Background: Linear Temporal Logic

Linear Temporal Logic (LTL) is a prominent formal specification language that is highly expressive and widely used in formal verification tools such as the model checkers SPIN [7] and NuSMV [1]. LTL is also used in the runtime verification of Java programs [17].
Formulas in LTL are constructed from elementary propositions and the usual Boolean operators not, and, or, imply (neg, \( \wedge \), \( \vee \), \( \rightarrow \), respectively). In addition, LTL allows for the use of the temporal operators next (\( X \)), eventually (\( F \)), always (\( G \)), until (\( U \)), weak until (\( W \)), and release (\( R \)).

Formulas in LTL assume discrete time, i.e., states \( s = 0, 1, 2, \ldots \). The meanings of the temporal operators are straightforward. The formula \( XP \) holds at state \( s \) if \( P \) holds at the next state \( s + 1 \). \( PUQ \) is true 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 \( FP \) is true at state \( s \) if \( P \) is true at some state \( s' \geq s \). Finally, the formula \( GP \) holds at state \( s \) if \( P \) is true at all moments of time \( s' \geq s \). Detailed description of LTL is provided by Manna et al. [10].

A major factor for the hesitance in using temporal logics in general is that they are hard to write. In addition, once specifications are written in LTL or CTL, it is hard to read and validate the meaning of the generated statement. For example, it is not immediately obvious that the LTL specification \( G(a \rightarrow F(p \wedge F(\neg p \wedge \neg a))) \) represents the English requirement “If a train is approaching(a), then it will be passing(p), and later it will be done passing with no train approaching”. The TV tool aims at providing the means by which developers and users can validate the meaning of such specifications as the one above.

2.1 Specification Pattern System and Prospec

Because of the difficulties associated with developing formal specifications, the Specification Pattern System (SPS) [3] developed a set of patterns to assist users in writing formal specifications in multiple formal languages. Patterns are high-level abstractions that provide descriptions of common properties that hold on a sequence of conditions or events in a finite state model. SPS patterns are grouped occurrence and order. Occurrence patterns are universality, absence, existence, and bounded existence. Order patterns are precedence, response, chain of precedence and chain of response. Chain patterns define a sequencing of events or conditions. Chain-precedence and chain-response patterns permit specifying a sequence of events or conditions as a parameter of precedence or response patterns, respectively. SPS allows the specification of sequences only to precedence and response patterns.

In SPS, a pattern is bounded by the scope of computation over which the pattern applies. The beginning and end of the scope are specified by the conditions or events that define the left (L) and right (R) boundaries, respectively. A study by Dwyer et. al. [3] identified the response pattern as the most commonly used pattern, followed by the universality and absence patterns. These three patterns accounted for 80% of the 580 properties sampled in the study.

In many system properties multiple propositions may be needed to specify pattern or scope parameters. Mondragon et al. introduced Composite Propositions (CPs) [11] to handle pattern and scope parameters that represent multiple conditions or events. This was done as part of the Property Specification (Prospec) tool [4, 12]. The introduction of CPs supports the specification of concurrency, sequences, and non-consecutive sequential behavior on patterns and scopes. Mondragon proposes a taxonomy with twelve classes of CPs. In this taxonomy, each class defines a detailed structure for either concurrent or sequential behavior based on the types of relations that exist among a set of propositions. The complete list of CP classes and their LTL descriptions is available in Mondragon et. al. [11].

The original version of Prospec [12] is an automated tool that guides a user in the development of formal specifications. It includes patterns and scopes, and it uses decision trees to assist users in the selection of appropriate patterns and scopes for a given property. Prospec extends the capability of SPS by supporting the specification of CP classes for each parameter of a pattern or scope that is comprised of multiple conditions or events. By using CPs, a practitioner is directed to clarify requirements, which leads to reduced ambiguity and incompleteness of property specifications.

Prospec uses guided questions to distinguish the types of scope or relations among multiple conditions or events. By answering a series of questions, the practitioner is lead to consider different aspects of the property. A type of scope or CP class is identified at the end of guidance. The soon to be released Prospec 2.0 generates formal specifications in Future Interval Logic (FIL), Meta-Event Definition Language (MEDL), and LTL. The automatic generation of CTL specification is left as future work. More detailed description of Prospec 2.0 can be found in [4].

3 Motivation

Although, the soon to be released, Prospec 2.0 supports the generation of formal specifications in multiple languages including LTL, it currently does not provide sufficient support for validation of the generated properties. While Prospec and similar tools and approaches [3, 16] provide significant support for property specification, there is a real need to ensure that the generated formal specifications do, indeed, match the original intent of the specifier. Additionally, it has been shown that the specifications generated by these tools, do not always match the natural language description provided by these tools [15]. Such discrepancies are easier to find once it is possible to validate the specifications generated by these tools against a user’s defined
expected behavior of the system. Section 4.3 provides examples of how the TV can assist in this validation effort.

Providing the means to validate the generated specifications is extremely significant, as effective use of these formal specifications (whether in formal verification, design and code automation, or test cases development) is not possible if the generated specifications are faulty (i.e., do not match the developer’s original intent). Indeed, incorrect specifications could lead to the very mishaps their use is designed to prevent.

By their nature, formal specifications are hard to read and validate, and as such, support for validation and understanding of these specifications is required. For example, consider an ATM system with the following property: “The response to user approval of a withdrawal transaction includes: the user’s account is updated, money is dispensed, the receipt is printed, and the user’s ATM card is returned”. This property can be specified in LTL as follows: “\(G(\text{user}\_\text{approval} \rightarrow F (\text{account}\_\text{updated} \land X (F \text{money}\_\text{dispensed} \land X (F \text{receipt}\_\text{printed} \land X (F \text{card}\_\text{returned}))))))\)”. It is obvious that such a description is hard to validate by those stakeholders who are not immersed in LTL.

4 Temporal Validator (TV) Tool

The Temporal Validator (TV) tool\(^1\) allows for simple validation of formal specifications written in LTL. The tool allows users to test traces of computations that represent system behaviors against LTL specifications. A trace of computation is a sequence of states that depicts the propositions that hold in each state. The user models a state of computation by assigning truth values to the propositions of the LTL formula for a particular state. For example, a user may examine one or more combinations of the following: a proposition holds in the first state of execution, a proposition holds in the last state, a proposition holds in multiple states, a proposition holds in one state and not the next.

To clarify the above idea, consider the following LTL formula: “\((F R) \rightarrow ((\neg R) U (P \land \neg R))\)”. This formula specifies that “If \(R\) holds, then \(P\) must hold before \(R\)”. Possible traces of computation to validate this formula are:

1. \(- - - - P - - R - - -\)
2. \(- - - - R - - - P - - -\)

According to the English specification above, the first trace is consistent with the LTL formula while the second should fail. Note that in the traces above, the symbol “-” indicates that none of the propositions of interest is true in that particular state.

---

\(^1\)The current TV tool can be requested from salamahs@erau.edu. Also a demo of the tool will be performed at the SEKE 09.
every execution of the model [2]. If there is an inconsistency between the model and the property being verified, a counter example, in form of execution trace, is provided to assist in identifying the source of the error. Figure 1 shows the process of model checking.

The general idea in TV is based on the work of Salamah et al., [15, 13, 14], and it consists of the following steps:

1. Create a simple model in SMV
2. map propositions in the formal specification to the variable(s) in the simple model,
3. run NuSMV with the model, formal specification, and proposition values as input, and
4. check for consistency.

The SMV model created for LTL validation, consists of a loop that starts with the value of the variable state equals 1 and continues to increment the value of states until it reaches 20, at which point it remains at 20. Figure 2 provides a graphical representation of the model, while Figure 3 provides the actual SMV code for the model.

While the details of the SMV code are irrelevant here, it is important to note that in each state of the model, the value of the variable Q.State changes to the value of the state. For example, in the first state, the value of Q.State is 1, it is 5 in the fifth state, and it is 20 in the last state. The importance of the value of the variable Q.State is that it is the value that propositions in the LTL formula are mapped to. For example, if one wants to validate the LTL formula “FP” (as in the fourth line in the SMV code), then the value of P has to be specified in terms of the variable Q.State. For example P can be specified as the truth value of the statement “Q.State = 5” (as in the third line in the SMV code), which is only true in the fifth state in the model.

4.2 TV Interface

TV allows users to input the LTL formulas to be validated either by reading from an input file, or by manually entering the LTL formula and the trace of interest. Figure 4 shows the user input as the LTL formula $G(p \rightarrow Fq)$. The formula specifies the Response property, i.e., “Anytime $p$ holds, it must be followed by $q$”. In addition, the figure shows the user’s trace of computation as: $pq$.

Figure 5 shows the result of the verification (“Evaluates to TRUE”) as returned by NuSMV, as well as a visual description of the trace.

Figures 6 and 7 show the validation of the same LTL formula for Response, but with the trace: $p$.

The result of the verification is FALSE, since $p$ holds in the ninth state and $q$ never holds after that. This is shown in Figure 7.

4.3 Scenarios

The following scenarios illustrate the use of the TV tool to validate formal specifications.
Scenario 1. Sarah is a software engineer working on security issues in web services. She recognizes that the system must support the following requirement: “A message recipient shall reject messages containing invalid signatures, messages missing necessary claims, or messages whose claims have unacceptable values.” Since the project team will use a model checker to verify the algorithms, she needs an LTL specification. Sarah uses the Prospec tool and generates the following LTL formula $G((invalid\_sign \lor miss\_claim \lor unacceptable\_value) \rightarrow F reject)$.

Although Prospec allowed Sarah to generate the above LTL formula, being new to LTL, Sarah is most likely not sure if this formula actually depicts her original intent. So, she inputs the generated LTL formula into the TV tool and then tries to validate her understanding by testing multiple traces of computations against the formula. Sarah runs the following traces (symbol $i$ stands for proposition invalid\_sign, symbol $m$ for proposition miss\_claim, symbol $u$ for proposition unacceptable\_value, and symbol $r$ for proposition reject):

- $--i---r----$
- $--r-u------$
- $u---m------$
- $(im) --------r$

Note that the last trace indicates that the propositions $i$ and $m$ hold in the same state (first state) of computation. Sarah observes the results from the TV tool and sees that they match her expected results for each trace (first and last traces are accepted while the others are not).

Scenario 2. Eric is a student in Software Engineering. He was just introduced to Temporal Logics and was asked to write some specifications in LTL. One of the properties Eric is trying to specify is the following “Every request must be followed by an acknowledgment.” Eric starts by assigning the symbols “r” and “a” to the propositions “request” and “acknowledgment” respectively. Eric comes up with the following LTL formula “$G(r \rightarrow F a)$”.

To make sure that his specification matches his understanding, Eric runs test the specification against the following traces (along with his expected results):

- $--r---a---- (valid)$
- $--a-a------ (valid)$
- $--a-r-a-r- (invalid)$
- $--(ra)-------- (invalid)$

Except for the last trace, TV returns the same result as Eric’s expected result. From this, Eric learns that the “Eventually” operator ($F$) in LTL holds if the operand (in this case $a$) holds in the current state. This is the case since the current state is part of the future in LTL. As a result of this testing, Eric now has to reconsider his understanding of the original property. He talks to his project teammates and instructor and comes to the conclusion that the desired property indicates that “acknowledgment” must “strictly” follow the “request” (i.e., it has to hold in a future state other
than the current state. Eric changes his original LTL formula to \(G (r \rightarrow XF a)\). He runs all the previous traces using the TV tool. The tool returns validation results that match Eric’s expected results.

5 Summary and Future Work

The TV tool was developed by students in the introductory course in Software Engineering at Embry-Riddle Aeronautical University. The tool allows users to examine formal specifications against traces of computations that represent the different behaviors of the software system being verified. Providing the means to validate formal specifications is extremely significant, as effective use of these formal specifications (whether in formal verification, design and code automation, or test cases development) is not possible if the generated specifications are faulty (i.e., do not match the developer’s original intent). Indeed, incorrect specifications could lead to the very mishaps their use is designed to prevent.

TV can also be used as part of property elucidation as a way to distinguish accepted and unaccepted behaviors of the system under consideration. The tool can also be used in academic settings in efforts relating to formal methods and formal specification [13].

Future work includes extending the tool to allow for validation of specifications written in CTL. The foundations of that work have already been implemented [14] and we plan to have it included in the future release of TV. Ultimately, we intend to integrate the TV tool into the Prospec tool and to link both property specification and property validation efforts. Finally, an important goal of this work is allow for the automatic generation of traces of computations and the expected results (as in Section 5.3) for LTL and CTL specifications. This will relieve the user of the burden of defining those traces, as well as provide another way of validating specifications.

References