A Novel Approach for Software Property Validation

Salamah Salamah
Department of Computer and Software Engineering, Embry-Riddle Aeronautical University, salamahs@erau.edu.
Irbis Gallegos, Omar Ochoa
Computer Science Department, University of Texas at El Paso, irbis, omar@miners.utep.edu.

Abstract

Formal approaches to software assurance such as runtime monitoring, model checking, and theorem proving have been shown to improve system dependability. All of these approaches require formal specifications of behavioral properties to verify a software system. Writing, reading, and validating formal specifications is difficult, and previously, there has been inadequate tool support for this task. The Property Specification tool, is a tool that was developed to assist users in the creation of formal specifications. Currently, Prospec assists in the generation of formal specifications in multiple languages. This paper describes a new approach/idea that will be integrated into Prospec to enhance the ability of users to validate the generated formal specifications, against the original intent.

1. Introduction

Formal methods to support software assurance require the identification of behavioral properties of the software system, generation of formal specifications for the properties, validation of the specifications, and verification of the correctness of the system. The effectiveness of the assurance approach depends on the quality of the formal specifications, and a significant hurdle to the use of formal approaches is the development of correct formal specifications [6, 7].

Typically, the person creating the formal specification is required to have a strong mathematical background and be aware of the subtleties of the specification language. For example, model checkers [2], such as SPIN [8] and NuSMV [1] use formal specifications written in Linear Temporal Logic (LTL) [11], which can be difficult to read, write, 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.

The specifier must be aware of the differences in expressiveness of each of the target languages.

The Property Specification (Prospec) tool [14] was developed to address some of these challenges. Prospec uses the Specification Pattern System (SPS) [3] and Composite Propositions (CP) [12, 13] to assist developers in the elicitation and specification of system properties. Usability studies of Prospec have shown that it facilitates the elicitation, understanding, and specification of formal properties [12].

1.1 Motivation

Although, the soon to be released, Prospec 2.0 supports the generation of formal specifications in multiple languages such as LTL, Computational Tree Logic (CTL) [10], and Meta Event Definition Language (MEDL) [9], it currently does not provide sufficient support for validation of the generated properties. While Prospec and similar tools and approaches [3, 18] 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 [16].

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 trans-
action 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 LTL as follows: “G(user_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}}))))”\textsuperscript{4}. It is obvious that such a description is hard to validate by those stakeholders who are not immersed in LTL.

This work describes a basic idea that can be used for simple validation of formal specifications in LTL, CTL and MEDL. We intend in incorporating this idea in future versions of Prospec.

The paper is organized as follows: Section 2 provides a brief description of the three formal languages LTL, CTL, and MEDL, and their semantics. Section 3 describes the validation approach introduced by the paper. The section includes descriptions of the models and codes for the verification tools used in the validation. The section also provides a scenario of using the proposed approach. The paper concludes with summery and future work, followed by the References.


text continues here...
accepted by the verification tool. We provide a way for validating specifications written in LTL, CTL, and MEDL. We use the NuSMV model checker [1] for LTL and CTL validation, and the Java-MaC runtime monitor [9] to validate MEDL specifications. In this approach, we use the very same verification techniques (and their respective tools) that these formal specifications are used for.

3.1 General Approach

The general idea in our approach is based on the work of Salamah et.al., [16], and it consists of the following steps:

1. Create a simple model in the language of the verification tool (NuSMV uses SMV language, and Java-MaC use the Java language),

2. map propositions in the formal specification to the variable(s) in the simple model, and

3. run the verification tool with the model, formal specification, and proposition values as input, and

4. check for consistency.

The model created for each type of formal language remains the same in validating all specifications in that language. The model is simple enough that it is easy to compare the expected result of running the model against the formal specification and the actual result as returned by the tool. In validating a specification we need only to change the mapping of the propositions in that specification to the variables in the model. More details on the approach are included in the upcoming paragraphs.

3.1.1 LTL and CTL Validation Using NuSMV

For validating LTL formulas, we used the NuSMV model checker. Although multiple numbers of model checkers, including the famous model checker spin [8] could have been used instead of NuSMV, the choice of NuSMV was basically because it can be used for both LTL and CTL validation.

The model created for LTL validation, consists 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 “F P” (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.

CTL validation using NuSMV can be done in a similar
fashion to LTL. However, we use a different model for validating CTL than the one used with LTL. The new model used for CTL validation is designed to allow for exploring and testing the branching capabilities of CTL. Figures 4 and 5 provide the model and SMV code, respectively, used for CTL validation.

3.1.2 MEDL Validation Using Java-MaC

Java-Monitoring and Checking (Java-MaC) [9] is a runtime verification tool developed at the University of Pennsylvania targeted towards Java Programs.

In Java-MaC, the user formally specifies the requirements in terms of high-level events and conditions. An event is something that occurs instantaneously during the program execution. A condition represents information that holds for a duration of time. A monitoring script is used to relate the events and conditions with the runtime low-level data of the system. Based on the monitoring script, the system is instrumented to send the monitored data to an event recognizer.

The event recognizer transforms the low-level data into abstract events and sends them to a run-time checker generated from the requirement specifications. The runtime checker verifies the sequence of abstract events with respect to the requirement specifications and detects violations of requirements.

Java-MaC uses two languages to formally specify properties, the Meta Event Definition Language (MEDL) and the Primitive Event Definition Language (PEDL). MEDL is used to express requirements, is based on an extension of LTL, expresses a large subset of safety properties of system such as real-time properties, and is independent of the monitored system. PEDL describes primitive high-level events and conditions in terms of system objects, and is tied to the implementation language of the monitored system.

For MEDL validation, we use the same model used for LTL validation with NuSMV, with the exception that the model is written in Java instead of SMV. The Java code representing the model is shown in Figure 6.

To validate a property in MEDL, we have to specify MEDL and PEDL files as shown in Figures 7 and 8 respectively. In the PEDL file, we specify the conditions or events that we build the MEDL specification on. This is similar to mapping the propositions in an LTL or CTL formula to the value of the variable Q.State in the SMV model. The MEDL file, contains the specification to be monitored by Java-Mac at runtime. The PEDL and MEDL files in Figures 7 and 8 are those defined for monitoring the specification that i will
always be less than a 100.

**Scenario:** Sue is interested in generating a formal specification for the following property “a request is always followed by an acknowledgment”. Assume further that $R$ signifies Request, and $A$ signifies Acknowledgment. Using Prospec\(^2\), Sue generates the following LTL specification 

$$G (R \rightarrow F A)$$

Sue’s original intent was that the acknowledgment must strictly follow the request, i.e., Acknowledgment has to hold in a state that comes after the state where Request holds. To validate whether such a behavior is allowed or not by the generated LTL specification, Sue runs the specification in NuSMV against the SMV model in Figure 3, setting both $R$ and $A$ to be true when Q.State equals 4 and observes NuSMV’s output. The output in this case is valid, which means that such behavior is allowed by the generated LTL formula. As a result, Sue uses Prospec to refine her property and generates the LTL property 

$$G (R \rightarrow X F A)$$

which does match her original intent.

### 4 Conclusions and Future Work

Although the use of formal verification techniques improve the dependability of programs, they are not widely adapted in standard software development practices. One reason for the hesitance in using formal verification is the difficulty of reading, writing, and validating formal specifications required for the use of these techniques. While there exists multiple tools and approaches that assist in the generation of formal specifications, these tools do not provide adequate support to validate the generated specifications.

In this work, we described an idea for validating formal specifications using the same formal verification tools that these specifications are intended to be used with. We provided simple models written in the native languages of these verification techniques, which can be used in the validation process. The approach defined in this paper can be used to validate specifications in different formalisms as long as they are supported by a verification tool.

Currently, as a project for sophomore level course in Software Engineering, a tool is being developed that allows for LTL and CTL validation using the model checker NuSMV based on the proposed approach. The tool will relieve the users of the burden of changing the models specified in Section 2 to test different specifications and different scenarios. The tool will be used as an interface to the model checker.

We plan on expanding that tool to include an interface for Java-MaC and allow for MEDL validation. Eventually, the developed will be incorporated into the Property Specification tool Prospec.

### References


---

\(^2\)Refer to [14] more details on property generation in Prospec


