Consistency Checks of System Properties Using LTL and Büchi Automata

Salamah Salamah, Matthew Engskow
Department of Electrical, Computer, Software, and Systems Engineering
Embry Riddle Aeronautical University, Daytona Beach, Florida, USA
{salamahs, engskowm}@erau.edu

Omar Ochoa
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, USA
omar@miners.utep.edu

Abstract

Although formal approaches to software assurance such as model checking and theorem proving improve system dependability, software development professionals have yet to adopt these approaches. A major reason for the hesitance is the lack of maturity of tools that can support use of formal specification approaches. These techniques verify system correctness against formal specifications. Tools such as the Specification Pattern System (SPS) and the Property Specification tool (Prospec) assist users in translating system properties (requirements) into formal specifications in multiple languages such as Linear Temporal Logic (LTL). The goal of such tools is to aid in the generation of a large set of formal specifications of systems or subsystems. A major advantage of formal specifications is their ability to discover inconsistencies among the generated properties. This paper provides an approach to extend the work of the aforementioned Prospec tool to allow for consistency checks of automatically generated system properties. The work will allow for the discovery of discrepancies among system requirements at early stages of system development, providing a return on investment for specifying formal properties using the Prospec tool.

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 numerous mission and safety critical systems. Because of society’s dependence on computers, it is vital to assure that software systems behave as intended. It is alarming to consider that software errors cost U.S. economy $59.5 billion annually [13]. The same studies also show that a significant amount of resources can be saved if software defects are discovered at the early stages of development such as requirements and design, rather than the later stages such as implementation and testing. Because of that, it is imperative that the software industry continue to invest in software assurance approaches, techniques, and tools, especially ones that ensure early detection of defects.

The use of formal methods in software engineering can be of great value to increase the quality of developed systems. Formal approaches to software assurance require the 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 adoption 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) [6, 7]. It is also important that the generated formal properties are consistent. A major advantage of using formalism in describing software properties, is that it becomes easier to discover conflicts between properties as early as these properties are generated, which is typically during the requirement and design phases.

Currently, there exist multiple formal specification languages that can be used in a variety of verification techniques and tools. Linear Temporal Logic (LTL) [10], and Computational Tree Logic (CTL) [9] are two of these languages. The aforementioned languages can be used in a variety of verification techniques and tools. For example,
the model checkers SPIN [8] 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.

In this work we propose an approach and a tool to check the consistency between system properties specified in LTL which is a prominent formal specification language in software engineering. The importance of the work stems from the fact that formal properties can be generated early in the development cycle, and detecting discrepancies among properties at that early stage can lead to significant savings in time and resources. The approach to detect inconsistencies makes use of translating LTL specifications into a special type of state machines called a Büchi automata and then checking for the intersection of the state machines for these specifications. This is the same approach used by model checkers to check the correctness of models against formal specifications of properties. The developed tool allows users to examine LTL specifications against other specifications and report any conflicts. The tool is intended to be part of the Property Specification (Prospec) tool [11, 12].

The rest of the paper is organized as follows: Section 2 provides the necessary background for the rest of the work including a description of LTL and Büchi Automata. Section 3 provides the motivation for the work. Sections 4 and 5 introduce the new approach for consistency check and a special tool developed for that purpose respectively. The paper concludes with a summary and the references.

2 Background

This section discusses the background work necessary for the rest of the paper. Specifically we describe the LTL language, the Prospec tool, the notion of Büchi automata, and the model checking approach for consistency checks.

2.1 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[8] and NuSMV [1]. LTL is also used in the runtime verification of Java programs [16].

Formulas in LTL are constructed from elementary propositions and the usual Boolean operators not, and, or, imply (neg, ∧, ∨, →, 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 \( X P \) holds at state \( s \) if \( P \) holds at the next state \( s + 1 \). \( P U Q \) 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 \( G(a → F(p ∧ F(¬p ∧ ¬a))) \) 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 proposed tool aims at providing the means by which developers and users can validate the meaning of such specifications as the one above.

2.2 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. [11] introduced Composite Propositions (CPs) to handle pattern and scope parameters that represent multiple conditions or events. This was done as part of the Property Specification (Prospec) tool [12, 5]. 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].

Prospec 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 guided 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 completed 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. Figures 1 and 2 provide screen shots of the Prospec tool. More detailed description of Prospec 2.0 can be found in [5].

2.3 Model Checking and Büchi Automata

Classical LTL model checking is based on a variation of the classic theory of finite automata [8]. While a finite automaton accepts only terminating executions, model checking requires a different type of machines that can handle executions that might not terminate. Such machines are necessary to model nonterminating systems such as operating systems, traffic lights, or ATMs. One such machine is a Büchi automaton. A Büchi automaton is a tuple \((Q, \Sigma, \delta, Q_0, F)\) where \(Q\) is a finite set of states, \(Q_0 \subseteq Q\) is a set of initial states, \(\Sigma\) is an alphabet, \(\delta: Q \times \Sigma \rightarrow 2^Q\) is a transition function, and \(F \subseteq Q\) is set of accepting states.

Languages of Büchi Automata represent a superset of those of LTL; every LTL formula can be represented by a Büchi Automata. When a Büchi Automata is generated from an LTL formula, the language of the Büchi Automata represents only the traces accepted by the LTL formula. For example, the Büchi Automata in Figure 3 represents the language accepted by the LTL formula \((a \ U \ b)\). This formula specifies that \(b\) holds in the initial state of the computation, or \(a\) holds until \(b\) holds. The language of the Büchi Automata in Figure 3 accepts the set of traces \(\{b, ab, aab, \ldots, aaab\}\). Notice that each of these traces passes through the accepting state Final. This state is both reachable from the initial state and is visited infinitely often (by virtue of the self-transition marked 1).

In Automata-Based model checking, the system model is written in the model checker modeling language (in case of SPIN the language is Promela [8], and in NuSMV it is the SMV language [1]. The languages of these model checkers
allow for the presentation of system models as Büchi automata. On the other hand, system properties of interest are provided as temporal logic formulas (SPIN accepts properties in LTL, and NuSMV accepts properties in both LTL and CTL). In the case of LTL formulas, the model checker translates the negation of LTL specification into a Büchi automaton and checks the intersection of both Büchi automata for the system model and that of the negated specification. If the intersection is empty, then the model and the original (non-negated specification) are deemed consistent. Otherwise, if the intersection is non-empty (it contains accepted execution traces) then the model accepts some behavior of the negated specification which implies an inconsistency between the model and the specification [2]. In this work, we use the same approach to check consistency between LTL specifications. Specifically, we check for emptiness of the generated Büchi automata that results from “anding” all LTL formulas for the properties in question.

3 Motivation

The goal of automated formal property generation tools like Prospec, is to generate formal specification for multiple system properties. However, Propsec only generates these properties and does not support any consistency checks among these properties. Considering that in a typical software development environment, system properties are elicited by multiple developers for different parts of the system under development, it is essential that these properties are consistent. Lack of overall consistency among system properties will result in problems that will surface during system integration. Discovering defects this late in development will result in significant rework and delays in production.

While formal specifications, by their nature, are better suited for formal analysis and consistency checks, they are hard to manually read or validate and, as such, are difficult to analyze for consistency by manual means. We provide a tool that takes as an input, multiple formal specifications (in LTL) generated by tools such as Prospec, and returns a verdict on the consistency between the set of formal specifications under questions. Moreover, the LTL Consistency checker tool, reports on the group of LTL specifications that cause this inconsistency.

4. Consistency of Formal Properties

Once systems properties are available as LTL formulas it is desirable to check the consistency among the generated properties. In this section we provide an approach to test the consistency of multiple properties written as LTL formulas. In our approach to demonstrate consistency between two LTL formulas we check for emptiness of the Büchi Automata of the conjunction of both formulas. Given two LTL formulas $LTL_1$ and $LTL_2$ the two formulas are consistent if the Büchi Automata for $(LTL_1 \land LTL_2)$ is not empty (i.e., the combined LTL formula “$LTL_1 \land LTL_2$” accepts some behaviors or execution traces).

By definition a Büchi Automata is empty if it does not contain a reachable accepting state that is visited infinitely often [2]. The work described here consists of using the LTL2BA[4] tool to generate the Büchi Automata for the formula $(LTL_1 \land LTL_2)$ and to check for an absence of acceptance state(s). If acceptance state(s) is/are available then we check that none of them is within a cycle (i.e., visited infinitely often).

An example of a consistent formula is shown in Figure 3 above. The figure shows the Büchi Automata for the formula “$p \lor b$”. This is clearly not an empty Büchi Automata as there is a final state (the double circle state called Final), and this state is within a cycle by virtue of the self loop (i.e., it is visited infinitely often).

Two examples of empty Büchi Automata are shown in Figures 4 and 5. Figure 4 shows the BA for the formula “$\neg (((G(l \land \neg r) \rightarrow ((Gp) \lor (pU r)))) \land (G((l \land \neg r) \rightarrow (\neg((p \land \neg r)U((\neg p) \land \neg r)))))$”. This Büchi Automata is empty because it contains no acceptance states (no double circles). The Büchi Automata in Figure 5 is for the LTL formula “$\neg (((G(l \land \neg r) \rightarrow ((Gp) \lor (pU r)))) \land (G((l \land \neg r) \rightarrow (\neg((p \land \neg r)U((\neg p) \land \neg r)))))$”. Although the Büchi Automata in Figure 5 contains an accepting state, this state
is not contained within a cycle and (it cannot be visited infinitely often), as such, this too is an empty automaton.

5 LTL Consistency Check Tool

The LTL Consistency Check Tool\(^1\) is designed to ensure that a given LTL formula is valid and is not contradictory (accepts some behaviors and is not empty). It may also be used to check that two separate LTL formulas do not contradict by using the AND operator. In addition, the tool can be used to prove the equivalence of two formulas. Provided we have formula’s A and B, checking to see that \((A \land \neg B)\) is invalid and that \((B \land \neg A)\) is invalid will prove that the formulas are equivalent \([14]\).

While the above examples in Figures 3-5 can be checked for consistency simply by examining the diagrams (these are manually drawn diagrams based on the textual output by LTL2BA) for the generated Büchi Automata, other more complex Büchi Automata can be hard to visually check for consistency. For this purpose, we have developed the LTL Consistency Check tool. The tool allows the user to input any number of LTL formulas to check for their consistency. The tool makes use of LTL2BA \([4]\) to construct the resulting Büchi Automata for the conjunction of these LTL formulas. The tool interfaces with LTL2BA to pass the “anded” list of LTL formulas (as one formula) and get back the resulting Büchi Automata (in text formats). From here, the tool tests the consistency of the formulas by looking for a reachable acceptance state that falls within a cycle. This is performed by first finding all states reachable from every given state, then by linking these together to form cycles. Should the tool find a single acceptance cycle, that is, an acceptance state which can reach itself, the formulas are declared consistent. Otherwise there is an inconsistency among the formulas. This is the same approach used in automata model checking \([2]\).

In case of inconsistency, the tool will loop through all the permutations of the input formulas and perform the same consistency checks on each permutation group. This procedure will ensure that the user is informed not only of the presence of an inconsistency, but also which set of specifications are inconsistent.

Example 1 Figure 6 shows a screen shot of the LTL consistency check tool showing no inconsistencies between the three LTL formulas\(^2\):

- \(G(p \rightarrow Fq)\)
- \(Fp\)

These three formulas are obviously consistent. The first describes a response formula, where a cause \(p\) must be followed by a effect \(q\). The second and third formula describe the behavior that at some future point of execution \(q\) will hold and \(p\) will hold respectively.

Example 2 Figure 7 shows the screen shot of the LTL consistency tool showing an inconsistency between the following three LTL formulas:

- \(G(p \rightarrow Fq)\)
- \(Fp\)
- \(G\neg p\)

These three formulas are obviously inconsistent. The first formula specifies that every occurrence of \(p\) must be followed by a \(q\), and the second formula specifies that there will be an occurrence of \(p\). Combining the two formulas one can conclude that \(q\) must happen at some future point. However the third formula states that \(q\) will never hold. As a result, the three formulas, combined, are inconsistent.

\(^{1}\)The current tool can be requested from salamahs@erau.edu. Also a demo of the tool will be performed at SEKE 12.

\(^{2}\)Note that the consistency check tool makes use of the SPIN syntax for (Global (G): \([\])\), (Future (F): \(<>\)\), and (Logical OR: \(||\)\)
Example 3  Figure 8 shows the screen shot of the LTL consistency tool showing an inconsistency between the following four LTL formulas:

- $G(p \rightarrow Fq)$
- $Fp$
- $G\neg p$
- $(G\neg r) \lor ((\neg r) U q)$

This latest example shows that the tool will distinguish between the inconsistent formulas and others that do not pose any inconsistency with other formulas. As such, the tool reports that the first three formulas are inconsistent (as described in Example 2) while the last formulas has no consistency issues with other formulas and itself is a valid formula (does accept certain execution traces).

6  Summary and Future Work

Formal verification techniques such as model checking and theorem proving have been shown to increase system dependability. These techniques verify system models against formal specifications. As such, the success of these techniques depends on the quality of developed systems properties. Tools that assist in the generation of formal specifications in LTL are important to the formal verification community as they relieve the user from the burden of writing specifications in a language that is hard to read and write. Without the help of tools such as Prospec, the user might create faulty specifications. These tools must generate specifications that correspond to the intent of the user. Prospec has demonstrated to provide such support [15]. However, it is also important that tools like Prospec generate specifications that are consistent with each other. Discovering inconsistencies among specifications early in the requirement or design phases should reduce integration and testing times significantly.

This work introduced a new approach, complemented with a tool, to automatically check for consistency among multiple formal specifications in LTL. The tool will help the users discover inconsistencies among their specifications at early stages. While the current tool is a stand-alone tool that has been developed separately from Prospec, it is our goal to integrate this tool with the latest release of Prospec in order to encourage the use of Prospec.

References