SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements

Aaron W. Fifarek[1], Lucas G. Wagner[2], Erika R. Hoffman[3], Benjamin D. Rodes[4], M. Anthony Aiello[4], and Jennifer A. Davis[2]

[1] LinQuest Corporation, Dayton, USA
aaron.fifarek@linquest.com
[2] Rockwell Collins, Cedar Rapids, USA
{lucas.wagner,jen.davis}@rockwellcollins.com
[3] Air Force Research Laboratory, Wright-Patterson AFB, USA
erika.hoffman@us.af.mil *
[4] Dependable Computing, Charlottesville, USA
{ben.rodes,tony.aiello}@dependablecomputing.com

Abstract. This paper describes current progress on SpeAR, a novel tool for capturing and analyzing requirements in a domain specific language designed to read like natural language. Using SpeAR, systems engineers capture requirements, environmental assumptions, and critical system properties using the formal semantics of Past LTL. SpeAR analyzes requirements for logical consistency and uses model checking to prove that assumptions and requirements entail stated properties. These analyses build confidence in the correctness of the formally captured requirements.

1 Introduction

This paper presents SpeAR (Specification and Analysis of Requirements) v2.0 [1], an open-source tool for capturing and analyzing requirements stated in a language that is formal, yet designed to read like natural language.

Requirements capture and analysis is a challenging problem for complex systems and yet is fundamental to ensuring development success. Traditionally, requirements suffer from unavoidable ambiguity that arises from reliance on natural language. Formal methods mitigates this ambiguity through mathematical representation of desired behaviors and enables analysis and proofs of properties.

SpeAR allows systems engineers to capture requirements in a language with the formal semantics of Past Linear Temporal Logic (Past LTL) [3] and supports proofs of critical properties about requirements using model checking [2]. Moreover, the SpeAR user interface performs validations, including type-checking, that provide systems engineers with real-time feedback on the well-formedness of requirements. Initial feedback from systems engineers has been positive, emphasizing the readability of the language. Additionally, our use of SpeAR on early case studies has identified errors and omissions in captured requirements.

* Approved for Public Release; Distribution Unlimited
(Case Number: 88ABW-2016-6046)

2 Related Work

Previous work has investigated the role of formal methods in requirements engineering. Parnas laid the foundation for constraint based requirements with the four variable model: monitored inputs, controlled outputs, and their software representation as inputs and outputs [11]. The Software Cost Reduction (SCR) method builds upon the four variable model using a tabular representation of requirements and constraints [10]. SCR provides tool support for formal analysis, including a consistency checker and model checker. SpeAR also builds upon the four-variable model but expresses requirements in a language that is designed to read like natural language instead of a tabular representation. In contrast to tools like ARSENAL [8] that provide formal analysis of natural language requirements, engineers use SpeAR to capture requirements directly in a formal language, avoiding the introduction of potential ambiguity.

Previous versions of SpeAR [5] used pre-defined specification patterns [4] that were found to be too rigid in practice. SpeAR v2.0 introduces a language providing the formal semantics of Past LTL that is more flexible, allowing users to capture requirements directly, rather than choosing from pre-defined patterns.

3 Formal Requirements Capture

SpeAR captures requirements in a formal language that not only provides the semantics of Past LTL, but is also designed to read like natural language. Previous versions of SpeAR required explicit scoping for temporal operators, using an awkward syntax, for example:

```
while signal > threshold :: always output == ON;
```

SpeAR v2.0 eliminates this syntax and provides English alternatives for most operators, such as `equal to`, `greater than`, `less than or equal to`, `implies`, and `not`. Additionally, SpeAR provides aliases for many operators so that systems engineers can more naturally express their requirements. With these English alternatives, the previous example can be written as:

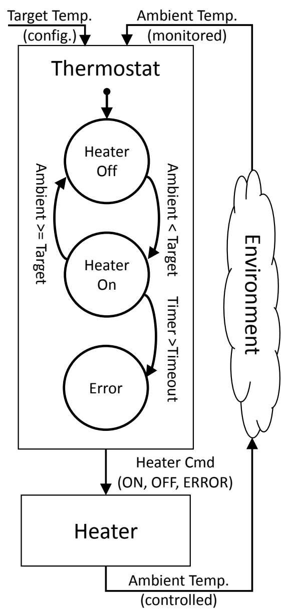
```
if signal greater than threshold then output equal to ON
```

This syntax is much closer to natural language.

We motivate further discussion of the SpeAR language by describing partial requirements for the thermostat of a simple heating system. As seen in Fig. 1a, the thermostat is represented by a three-state automaton describing reactions to changes in the ambient temperature.

3.1 SpeAR File Stucture

SpeAR promotes grouping requirements according to system components enabling modularity and reuse. Requirements are captured in files laid out in a common structure. Partial requirements for the thermostat, a component of the heating system, are shown in Fig. 1b.



(a)

```
Inputs:
  ambient is a temperature
  target  is a temperature
Outputs:
  heater_command is a heater_operating_state
State:
  timer is an int
Assumptions:
  a0: previous heater_command equal to ON implies
        ambient greater than or equal to previous ambient
Requirements:
  r0: if ambient less than target and
           timer less than or equal to TIMEOUT then
        heater_command equal to ON
  r1: if ambient greater than or equal to target and
           timer less than or equal to TIMEOUT then
        heater_command equal to OFF
  r2: if timer greater than TIMEOUT then
        heater_command equal to ERROR
  r3: timer equal to previous(timer) + 1
Properties:
  p_heat:  if heater_command equal to ON then
             ambient less than target
  p_off:   if heater_command equal to OFF then
             ambient greater than or equal to target
  p_error: if heater_command equal to ERROR then
             timer greater than TIMEOUT
  p_elatch: once heater_command equal to ERROR
             implies heater_command equal to ERROR
```

(b)

Fig. 1. (a) Simple Heating System with associated (b) Partial Thermostat SpeAR File

Inputs, Outputs, State: Inputs represent monitored or observed data from the environment, as well as inputs from other components. Outputs represent data to the environment, as well as outputs to other components. State represents data that is not visible to the environment or to other components. For example, the thermostat monitors the ambient and target temperatures for a room (*inputs*), controls the heater by sending a signal that turns it on or off (*outputs*), and has a counter that tracks heating duration (*state*).

Assumptions: Assumptions identify necessary constraints on inputs from the environment and from other components. For example, the thermostat assumes that the ambient temperature rises when the heater is on (`a0`). This constraint is an assumption: the thermostat cannot directly control the ambient temperature.

Requirements: Requirements identify constraints that the component must guarantee through its implementation. For example, the thermostat will send a signal to turn the heater on when the ambient temperature is lower than the target temperature (`r0`).

Properties: Properties represent constraints that the system should satisfy when operating in its intended environment. Properties can be used to validate that the requirements define the correct component behavior or to prove that certain undesirable conditions never arise. For example, the heater is only on when the ambient temperature is below the target temperature (`p_heat`).

Table 1. Past Time Temporal Expressions with SpeAR Equivalences

SpeAR	Past LTL
previous ϕ with initial value *false*	Yϕ
previous ϕ with initial value *true*	Zϕ
historically ϕ	Hϕ
once ϕ	Oϕ
ϕ since ψ	ϕ S ψ
ϕ triggers ψ	ϕ T ψ

3.2 SpeAR Formal Semantics

The formal semantics of SpeAR is as expressive as Lustre [9] and is based upon Past LTL [3] but omits future looking operators. We define this subset as Past-Only LTL, which allows users to express temporal behaviors that begin in the past, with arbitrarily long but finite history, and end at the current step (i.e., transition). Supported temporal operators in SpeAR are shown in Table 1, where φ and ψ are propositions — unlike Past LTL, SpeAR provides support for a general `previous` operator that can be used on all legal types in the model, not just boolean types. In addition to temporal operators, SpeAR provides basic arithmetic, logical, and relational operators.

4 Analysis

In addition to capturing requirements formally, SpeAR provides an analysis platform. SpeAR performs type checking, dimensional analysis of unit computations, and other well-formedness checks on the requirements in real-time. Once requirements have passed these checks, the user can analyze the requirements for logical entailment and logical consistency.

4.1 Logical Entailment

SpeAR enables systems engineers to prove that stated properties are consequences of captured assumptions and requirements. This capability provides early insight into the correctness and completeness of captured requirements.

Formally, SpeAR proves that the conjunction of the Assumptions (A) and Requirements (R) entails each Property (P) as shown in Eq. (1).

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge R_1 \wedge R_2 \wedge \cdots \wedge R_m \vdash P_i \qquad (1)$$

SpeAR proves entailment by (1) translating SpeAR files to an equivalent Lustre model and (2) analyzing the Lustre model using infinite-state model checking. SpeAR presents a counterexample if the requirements do not satisfy a property.

In the thermostat example seen in Fig. 1b, there are four properties: `p_heat`, `p_off`, `p_error`, and `p_elatch`. Two properties describe the nominal behavior of the system: (1) `p_heat` asserts the heater is on if the ambient temperature is less than the target temperature, (2) `p_off` asserts the heater is off if the ambient temperature is greater than or equal to the target temperature. Two properties describe the error behavior of the system: (1) `p_error` asserts the system is in the error state if a timeout occurs, (2) `p_elatch` asserts that after the system enters the error state it remains in that state.

Logical entailment allows systems engineers to prove the captured requirements and assumptions satisfy all of the stated properties.

4.2 Logical Consistency

Logical entailment is only valid if the captured requirements and assumptions are not conflicting. When there is a conflict among requirements or assumptions, the logical conjunction of the constraints is false, and thus the logical implication described in Eq. (1) is a vacuous proof (i.e., $false \implies true$).

Currently, SpeAR provides partial analysis to detect logical inconsistency. Logical inconsistency can exist for all steps and inputs, for example when two constraints are always in conflict. Logical inconsistency may also occur only during certain steps or as a result of certain inputs.

SpeAR analyzes requirements for logical inconsistency that is provable within the first N steps, for some user-selected N. This is accomplished by (1) translating SpeAR files to an equivalent Lustre model and (2) searching for a counterexample to the assertion that the conjunction of the assumptions and requirements cannot be true for N consecutive steps, beginning at the initial state, as shown in Eq. (2). Since we use counterexample generation to check consistency, we need a minimum step count to prevent the model checker from merely confirming that the requirements are consistent on the first timestep (a 1-step counterexample).

$$\neg((A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge R_1 \wedge R_2 \wedge \cdots \wedge R_m) \wedge (Step_{Count} >= N)) \qquad (2)$$

If the requirements are proven inconsistent for the first N steps, SpeAR alerts the user to the inconsistency and identifies the set of constraints in conflict. If, however, a counterexample is found to Eq. (2), SpeAR declares the requirements to be consistent even if the constraints are inconsistent at step $N+1$ or for some other set of inputs. This result may mislead the systems engineer to conclude that the requirements are consistent when in fact they are inconsistent. Future versions of SpeAR will address this issue by implementing the stronger concept of realizability [6] — a proof that all requirements and assumptions are consistent for all steps and combinations of inputs that satisfy the assumptions.

5 Conclusion and Future Work

SpeAR is a tool for capturing and analyzing formal requirements in a language that provides the formal semantics of Past LTL and is also designed to read like natural language. In addition to type checking and real-time validation of well-formedness, SpeAR provides two analyses that depend upon model checking: logical entailment and logical consistency. Logical entailment proves that specified properties, which define desired behaviors of the system, are consequences of the set of captured assumptions and requirements. Logical consistency aims to identify conflicting assumptions and requirements.

Systems engineers familiar with, but not experts at, formal methods provided positive initial feedback: SpeAR is more readable than typical formal languages and is worth the effort of learning. Additionally, applying SpeAR to requirements for a stateful protocol revealed a set of unreachable states; a decision was based on a variable whose value was overwritten on the current step. This error represented an incomplete understanding of the requirement that would have been difficult to identify through testing or inspection. After all contributing errors were found and fixed, SpeAR was used to prove that all states were reachable.

While this paper presents current progress on SpeAR v2.0, development and improvement is ongoing. We will expand logical consistency analysis to include realizability, allowing users to prove that the requirements are consistent for all steps and inputs. We will incorporate recent work in inductive validity cores [7] to provide logical traceability analysis, allowing users to identify which requirements and assumptions are used to prove each property — unused requirements and assumptions should be deleted as they overconstrain the system.

We are continuing to refine SpeAR and assess its utility by applying it to the development of unmanned autonomous systems and other research efforts. These results will be presented in future publications.

References

1. `https://github.com/lgwagner/SpeAR`
2. Baier, C., Katoen, J.P., Larsen, K.G.: Principles of Model Checking. MIT press (2008)
3. Cimatti, A., Roveri, M., Sheridan, D.: Bounded Verification of Past LTL, pp. 245–259. Springer Berlin Heidelberg (2004)
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-State Verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice. pp. 7–15. FMSP '98, ACM, New York, NY, USA (1998)
5. Fifarek, A.W., Wagner, L.G.: Formal Requirements of a Simple Turbofan Using the SpeAR Framework. In: 22nd International Symposium on Air Breathing Engines. International Society on Air Breathing Engines, University of Cincinnati (2015)
6. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards Realizability Checking of Contracts using Theories. In: NASA Formal Methods Symposium. pp. 173–187. Springer (2015)
7. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient Generation of Inductive Validity Cores for Safety Properties. ArXiv e-prints (Mar 2016)
8. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: Automatic Requirements Specification Extraction from Natural Language, pp. 41–46. Springer International Publishing, Cham (2016)
9. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Data Flow Programming Language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)
10. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for Constructing Requirements Specification: The SCR Toolset at the Age of Ten. International Journal of Computer Systems Science and Engineering 20(1), 19–53 (2005)
11. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming 25, 41–61 (1995)