Specification: The Biggest Bottleneck in Formal Methods and Autonomy *

Kristin Yvonne Rozier[1]
Iowa State University, Ames, Iowa, USA,
kyrozier@iastate.edu

Abstract. Advancement of AI-enhanced control in autonomous systems stands on the shoulders of formal methods, which make possible the rigorous safety analysis autonomous systems require. An aircraft cannot operate autonomously unless it has design-time reasoning to ensure correct operation of the autopilot and runtime reasoning to ensure system health management, or the ability to detect and respond to off-nominal situations. Formal methods are highly dependent on the specifications over which they reason; there is no escaping the "garbage in, garbage out" reality. Specification is difficult, unglamorous, and arguably the biggest bottleneck facing verification and validation of aerospace, and other, autonomous systems.
This VSTTE invited talk and paper examines the outlook for the practice of formal specification, and highlights the on-going challenges of specification, from design-time to runtime system health management. We exemplify these challenges for specifications in Linear Temporal Logic (LTL) though the focus is not limited to that specification language. We pose challenge questions for specification that will shape both the future of formal methods, and our ability to more automatically verify and validate autonomous systems of greater variety and scale. We call for further research into LTL Genesis.

1 Introduction

Formal methods have now scaled to the point of enabling rigorous safety analysis of full-scale, real-life systems, and none too soon, as such capabilities are required for developing the autonomous systems of the future. This is because autonomy requires systems to be reactive and concurrent [36], operating in real-time and in an open environment. Formal methods have been recognized as a critical, and often expected, design-time component for autonomous and life-critical systems, such as aircraft and spacecraft. FAA standards including DO-178-B [46] DO-178-C [48], and DO-254 [47] incorporate formal specification, validation, and verification. For one example, NASA's Lunar Atmosphere Dust Environment Explorer (LADEE) mission was a resounding success. LADEE used model-based development starting with specification of the requirements; refinement of these specifications via analysis against system models; automatic generation of software from verified models; and a variety of verification techniques including formal methods, static analysis, formal inspection, and code coverage applied early and often throughout the system design lifecycle [22]. We have influenced the design of an automated air traffic control system via model checking analysis

* Thanks to NASA's Autonomy Operating System (AOS) Project and NSF CAREER Award CNS-1552934 for supporting this work.

[55,56,57]. We have also used formal methods to help NASA assess the Functional Allocation question: in the early design stage, when there are thousands of options for allocating essential system functions, how can we formally analyze the space of many possible deigns to determine which are the most safe [37,16]?

In addition to design-time analysis, autonomous systems in particular critically depend on formal runtime reasoning, for runtime verification that unanticipated events do not violate their specifications, and to ensure system health management, or the ability to detect and respond to off-nominal situations that could not be verified at design time. NASA's Copilot language and compiler generates runtime monitors for distributed, hard real-time systems, including pitot tube subsystems and MAVLink (Micro Air Vehicle Link); these verified systems have flown in the Edge 540 aircraft [38]. Our own Realizable, Responsive, Unobtrusive Unit (R2U2) [41,18,51,49,50] utilizes formal specifications to generate runtime observers integrated with Bayesian reasoning to provide runtime system health management for Unmanned Aerial Systems (UAS) such as NASA's Swift and DragonEye UAS.

All of these formal methods, from design time to runtime, require formal specifications. A *formal methodology*, as defined by Manna and Pnueli in their seminal text on reactive and concurrent systems [36], consists of a specification language and a repertoire of proof methods by which the correctness of a proposed system, relative to the specification, can be formally verified. By this definition, a formal methodology provides two components central to autonomy: (1) the ability to make early, precise decisions, e.g. between multiple possible designs, about major system functions; (2) the ability to remove ambiguities from the system's expected behavior, from design-time behavioral descriptions to runtime behavioral monitors. For clarity through the remainder of the paper, we will distinguish the formal specification, or the description of the behavioral requirement that most often appears in the form of a formula (which we will call φ), from the system model that instead specifies how the system works (M). The verification question is then the question of whether (or not) these two things match; both are necessary inputs to a proof method.

Figure 1 shows one such example of a formal methodology. In this case, the formal specification is given as a set of Linear Temporal Logic (LTL) formulas; the system model is a description of system operation in a formal semantics we call M. A set of *validation specifications* is written simultaneously with the system model M; specification debugging increases confidence in the correctness of this set, and model checking against M serves to validate M. A set of *verification specifications*, which first pass specification debugging, are model-checked against M to verify that the early design satisfies its requirements. These specifications can then be carried throughout the system development process, e.g., used for test-case generation or simulation, and all the way to runtime verification of the final system implementation. This goal system design process, using Linear Temporal Logic (LTL) as the specification language and model checking as a proof method appeared in [56], where it was used successfully during the design time of a coordination protocol for an automated air traffic control system. Formal methods, including model checking, are highly dependent on the specifications over which they reason; not only are specifications required for analysis, but there is no escaping the "garbage in, garbage out" reality.

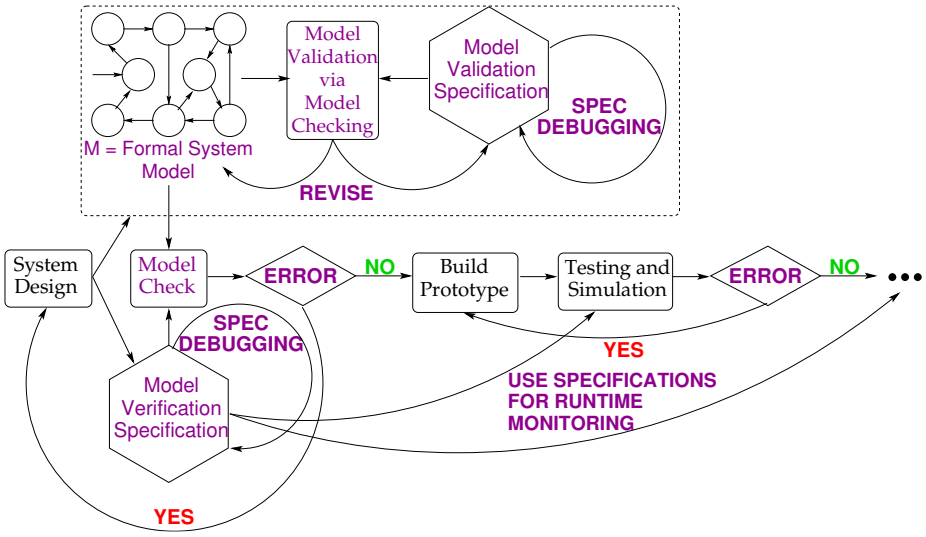

Fig. 1. A goal system design process (based on LTL model checking) where specifications are first debugged, then utilized for early system design validation, used in design verification, and carried through the system development process to runtime [56].

Figure 2 zooms in on the inputs to this process. The bottom line for formal methods is that the inputs to formal analysis are the biggest challenge. In [56], over 100 person-hours were required to create the inputs, which dwarfs the less than 10 hours of total runtime required to complete model checking analysis. In the follow-on study of a more complex version of the system with a large space of possible designs, over 1000 person-hours were required to generate the inputs that resulted in the 1620 model-checking instances (model-specification-set pairs) whose automated verification then averaged approximately 5 minutes per pair [16]. (Validation and further analysis, e.g., using fault trees, took several hours per pair but still far less time than specification; total time for input generation of all automated analysis including validation, verification, and fault tree analysis totaled over 2000 person-hours [17].)

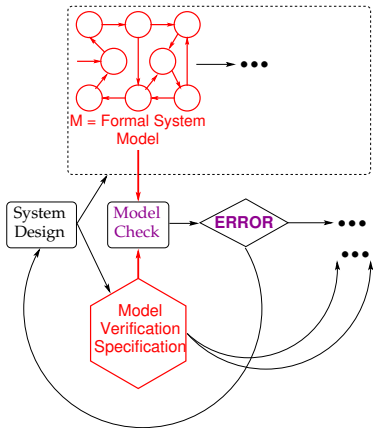

Fig. 2. Bottom line: **inputs** to formal analysis are the **biggest** challenge.

When it comes to formal system modeling, there is some hope in the form of *synthesis*. Recall that in model checking, we check whether $M \models \varphi$, e.g., does the system model satisfy its specification? *LTL Synthesis* is predicated on the fact that designing M is hard and expensive; re-designing M when $M \not\models \varphi$ is also hard and expensive [52]. Starting from LTL formula φ synthesis designs M such that $M \models \varphi$, which simplifies verification, eliminates the problem of re-designing M, and, for algorithmic derivations,

eliminates the burden of design entirely [52]. While synthesis as a technique does not yet enjoy the same level of tool support or scalability as verification techniques such as model checking, the field is well on the way to being able to greatly improve the bottleneck of the system model as input to the formal verification process. However, synthesis shares with model checking the requirement of a formal specification: the input formula φ. So, while synthesis is a worthwhile goal with the potential to eventually solve half of the inputs bottleneck, what we really need is **LTL Genesis**!

The remainder of this paper is organized as follows. Section 2 asks where we will get specifications from, while Section 3 examines how we will examine their quality. Section 4 asks how do we best use specifications, including introducing new ideas for specification patterns. Section 5 asks how should we organize specifications to enable these uses and examines the merits of strategies for accomplishing this. Section 6 concludes and gives an outlook for a future of well-specified autonomous systems.

2 Specification Origins

Specifications are required for all applications of formal methods, yet extracting specifications for real-life safety critical systems often proves to be a huge bottleneck or even an insurmountable hurdle to the application of formal methods in practice. This is the state for safety-critical systems today and as these systems grow more complex, more pervasive, and more powerful in the future, there is not a clear path even for maintaining the bleak status quo [3,4].

At NASA in particular, extracting specifications needed for any formal analysis is a huge challenge [55,4,56,5,37,16]. Some critical systems are designed without ever having what this community would consider to be a formal set of requirements. Some design processes don't formally define requirements until the testing phase, far too late to use them for design or design-time analysis, or other key periods in the system development life-cycle where formal methods are applicable. Even for critical systems where specifications are defined early in the system development life-cycle, they often mix many different objectives, mixing many different levels of detail and describing things like how the system is defined, how the system should behave, legal-speak on why the system satisfies rules, and more – sometimes all in the same sentence! As safety-critical systems become increasingly complex and the budgetary and other constraints tighten, where can we look in the future to hope to extract the specifications we need for formal analysis?

Even outside of the formal methods community, systems engineering processes are adapting to the fact that the old standard V model of systems engineering (shown in Figure 3) is outdated and does not capture the steps necessary for the design of today's complex, possibly autonomous, systems [27]. This realization comes from the need to define, and debug, requirements first, modify them throughout the system design lifecycle with each new phase of development, and perform verification at every stage of system design, not just at the end. AFRL has documented the unreasonable cost associated with the V model [25,26,21]. While an estimated 70% of faults are introduced in the early design phases on the left of the V, all but 3.5% are found in the later stages of system integration and testing (on the right of the V), where they are increasingly costly

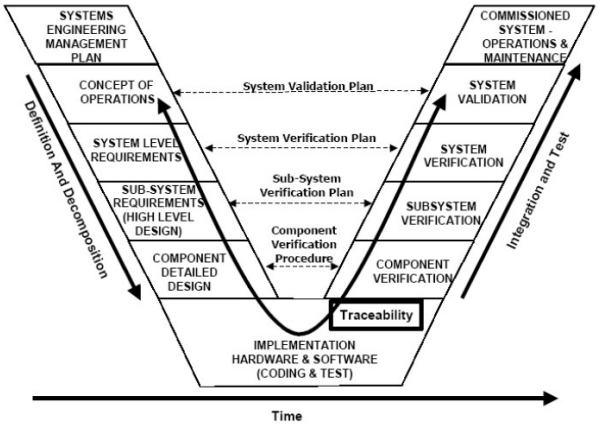

Fig. 3. An illustration of the outdated V Systems Engineering Model from [27].

to fix. The estimated nominal cost for fault removal is 300-1000x for faults found in the final "Acceptance" or "Operation" phase versus the early design phases [25,26]. The emerging realization that we need to define precise specifications that can be automatically analyzed from the earliest stages of system design has given rise to many different methods for deriving specifications, e.g., in LTL.

Though none of these have emerged as industrial standards, several specification extraction strategies remain under study as active areas for further research.

Human Authorship: Train system designers to write formal specifications first and have them author their own LTL, or pair designers with formal methods team to write specifications.

- Advantages include potential for accuracy and improved design-level reasoning; disadvantages include high learning curves and lack of automation.

Natural Language Processing (NLP): Extract formal specifications from English Operational Concepts, encoding them in LTL for automated analysis. Notable tools include ARSENAL [20] and VARED [5]. NLP is highly input-dependent: it is difficult to handle unstated assumptions, implied/arbitrary functions, slang, mixed abstraction levels, and other inconsistencies. There is a question whether structured English is advantageous over natural language.

- Advantages include the high level of automation and low learning curve required; disadvantages include that is it hard to measure correctness, completeness, and closeness to the designers intentions.

Specification Mining: Extract behaviors from existing systems. Can combine with test-case generation to explore system behavior [13].

- Advantages include automation; disadvantages include the need for a code specification as input.

Static Analysis: Map all paths of a program.

- Advantages include automation; disadvantages include that it is hard to differentiate normal usage from exceptions; also some essential specifications, like function postconditions, can be difficult to extract [54].

Learning/Dynamic Invariants: Analyze actual executions; observe use-cases.

- Advantages include that checking observed variable values against a library of fixed invariant patterns can automatically generate valuable specifications. Disadvantages include that specifications might refer to internal details or be irrelevant; observations are too limited and are heavily dependent on the set of observed executions [54].

Specification Wizard: Semi-automated exploration of system facets, guided by human input.

- Disadvantages include that similar ideas similar were tried previously and failed to catch on widely; advantages include that today's complex autonomous systems demand a more standardized system design process that may provide a better platform to build upon. With the widespread use of COTS components that could be added to an online database and the recent advances in specification extraction from LTL patterns and component parameters, there is a new opportunity for a wizard.

Notably, Zeller asked: can we have specifications for free [54]? Can we combine specification mining, test-case generation, static analysis, and dynamic invariants to extract specifications automatically? The specifications would be automatically mined from code, so that specification validation would equate to software defect detection. While this is a promising strategy for software runtime verification, fundamentally this process still requires code as an input. (In a sense, the code is now the specification; so we have not solved the specification genesis problem.) This strategy does not solve the specification problem for early design time, where code has not yet been written, or for cyber-physical systems that combine code with other components. The problem of requiring input code can be mitigated by using specifications extracted from the last version of a system for creating new designs. However, there remain challenges with specialization of the previous code, levels of abstraction, and relevance to the new system. Other challenges include scalability, efficiency, and expressiveness of extracting specifications for free. Still, Zeller's idea is highly intriguing!

3 Specification Quality

How can we know when we're "done" extracting specifications or have some idea of how well we've done? As critical systems continue to grow in complexity, how will we measure the completeness, coverage, or general quality of a specification or a set of specifications? We asked these questions in a panel at NFM2014 [4], yet in large degree they remain open areas for future research.

The emerging area of *specification debugging* [24,30], also called *sanity checking*, has made notable progress in automated analysis of specification quality, chiefly in four areas. We briefly discuss each, with respect to LTL specifications specifically.

Satisfiability For LTL, satisfiability checking reduces to model checking against a *universal model*, or a model that accepts all possible valuations of the variables at all states [43]. Formally, if we let φ be a specification over the set $Prop$ of propositions then a system model M is *universal* if it contains all possible traces over $Prop$:

$L_\omega(M) = (2^{Prop})^\omega$. A model checker negates φ and checks for emptiness of the combined model for $\neg\varphi$ and M. Then φ is satisfiable by any counterexample returned by model checking against M: $M \not\models \neg\varphi$ iff φ is satisfiable. If there is no counterexample, then φ is not satisfiable. In [43,44] we advocate for a sanity check of checking φ, $\neg\varphi$, and the conjunction of all specifications describing the same system for satisfiability before using them in system design and verification.

Stated another way, let φ describe a "good" requirement that the system must uphold. Then $\neg\varphi$ describes a "bad" behavior that the system must never display. The model checker takes as input φ, then negates it, combines it with the input model, and checks if the resulting combined automaton is empty, outputting a counterexample if not. Model checking φ against a universal model will show whether or not $\neg\varphi$ is satisfiable. A counterexample returned by the model checker in this case is a satisfying assignment to the formula. If $\neg\varphi$ is not satisfiable, then the model checking search of its combination with the universal model will not return a counterexample because no satisfying assignment exists. The reverse situation is also a problem. If φ is not satisfiable, then $\neg\varphi$ is a tautology. So, in a normal model checking run, we would model check $\neg\varphi$ against a system model, the model checker would negate φ to get $\neg\varphi$, and return a counterexample, which we are expecting to indicate that there is something wrong with the system model. However, since $\neg\varphi$ is a tautology, no matter how we change the system model, we will always get some counterexample.

In [44], we conducted an extensive experimental evaluation of LTL satisfiability checking via model checking, concluding that using symbolic model checking for this task is vastly superior to explicit-state model checking in terms of both correctness and performance. (Symbolic tools always returned the correct SAT/UNSAT result; this was not true for any of the explicit tools available at the time, perhaps due to the difficulty of implementing their algorithms.) In [45] we designed a portfolio approach consisting of 30 new encodings for LTL satisfiability via symbolic model checking that performed up to exponentially faster than was previously possible. In [33,34], the explicit approach was improved, circumventing explicit-state model checking and solving the LTL satisfiability problem directly using techniques borrowed from propositional SAT solving. Today, the (freely available) tools PANDA [45] and Aalta [34], represent the state of the art in symbolic (via the nuXmv model checker) and explicit LTL satisfiability checking, respectively.

Vacuity Sanity checks in industry include many types of simple, often ad hoc, tests such as checking for duplicate conflicting variable assignments or enabling conditions that are never enabled [32]. *Vacuity checking* can help detect errors in specifications by checking whether a subformula of a specification does not affect the satisfaction of the specification in the system model [31]. A common example is checking for implications like $\Box(p \rightarrow \Diamond q)$ where p can never be enabled. *Inherent vacuity checking* is a set of sanity checks that can be applied to a set of temporal properties, even before a model of the system has been developed, but many possible errors cannot be detected by inherent vacuity checking [15]. This capability is available in some proprietary industrial tools [7], and VaqUoT provides a front-end checker for nuXmv, but it only handles the subset of LTL that can be encoded as CTL [19]. VARED [5] integrates an updated algorithm for vacuity checking [23] into an end-to-end toolchain for requirements analysis.

Realizability Realizability checking provides another, stronger sanity check for a set of temporal properties in LTL by testing whether there is an open system that satisfies all the properties in the set [40], but such a test is very computationally expensive: 2ExpTime-Complete. However, notable progress on the problem is underway. RATSY [8] checks realizability of the class of Generalized Reactivity(1) (GR(1) [39]) specifications via an interactive game with the specifier. Acacia+ [9] also solves LTL realizability problems encoded as safety games. Another approach to realizability checking [35] builds upon RATSY using a template-based specification mining approach to identify situations of an under-constrained environment or an over-constrained system. This approach is complimented by work on detecting unrealizability due to overly-strong system guarantees or overly restricted signals [29]. An algorithm for finding minimal cores of unrealizability of GR(1) specifications is implemented in nuXmv [12]. All of these address the tricky space of checking specifications that are satisfiable but unrealizable because there is no implementation that can produce outputs that satisfy the specification given all possible inputs that can be generated by the environment. Realizability is inherently tied to synthesis: the LTL synthesis problem seeks to produce a model such that φ is realizable.

Coverage Coverage is a complicated sanity check because significant research has been contributed just to a set of definitions; measuring coverage for each such definition is a separate research question. Informally, coverage asks whether a set of LTL specifications considers all of the behaviors of the system; behaviors may be defined in various ways with respect to states or paths through an execution graph/automaton required for a specification to pass, the set of system variables, model checking analysis, checks for incomplete or redundant sub-models, etc. In a sense, coverage is complimentary to vacuity checking in that it asks whether there are parts of the system that are not relevant for the verification process to proceed. Coverage checking for LTL can be integrated into model checking [11]. Algorithms for automatically checking LTL coverage and completeness have been successfully used in industry for sanity checking, e.g., the requirements for an airplane control system [6].

4 Specification Usage

How should formal specifications (both those we are given and those we must extract) fit into the design life-cycle for different kinds of critical systems? How can we indoctrinate formal specifications into diverse teams of system designers without hitting barriers to adoption such as huge costs in terms of time and learning curves? What should our roadmap look like for a future full of well-specified (formally analyzable) critical systems?

Figure 4 shows the updated waterfall model for system design that has supplanted the former V model of Figure 3. The need to define specifications early and carry them through all stages of system design has given rise to many different specification use strategies. All present interesting challenges for future research.

Property-Based Design: system design centers around specifications

- Challenge: defining a foundation of specifications early

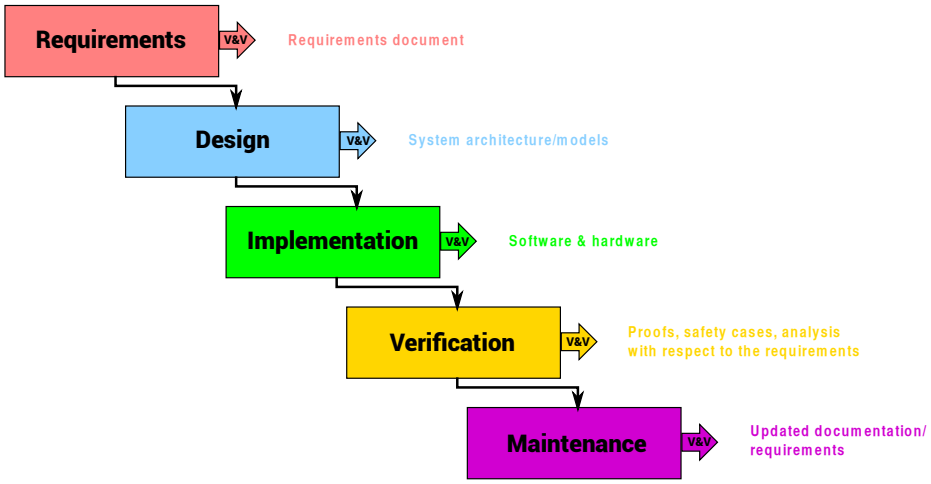

Fig. 4. The current waterfall model for system engineering incorporates the specifications (aka system requirements) throughout all phases of system design.

Synthesis: generate M such that $M \models \varphi$

– Challenge: how can we synthesize a *cyber-physical* system M?

Specification-Based Testing: use specifications in test-case generation

– Challenge: how can we carry specifications through different levels of abstraction?

From Design- to Run-Time: carry specifications through the design cycle

– Challenge: how do we define a specification design lifecycle?

Maintenance: using specifications in system up-keep

– Challenge: what do best practices for maintenance of specifications look like?

4.1 Specification Patterns

Since the early days of temporal logic specifications, we have been concerned with dividing them into classes like Safety/Liveness/Guarantee/Obligation, Fairness/Justice/-Compassion, or Safety/Response/Reactivity [36]. While these classes have proven useful in specializing algorithms for automated analysis, they are still too coarse and tied to syntax for practical use; there is a need for more functional and hierarchical specification.

Dwyer et al. [14] answered with definitions of specification formula patterns that have many practically useful properties. Formula patterns are organized in a hierarchy based on semantics and leverage experience with design and coding patterns to enable system designers to more efficiently generate specifications. This *specification pattern system* captures recurring solutions and allows specifiers to generalize across domain-specific problems. It encourages re-use by better enabling practitioners to identify the

same patterns across systems and makes transparent the means by which requirements are satisfied.

Formula patterns each have a name, intent, logic (language), scope (time interval), and relationship to other patterns. Each pattern is characterized by the following traits:

- Solves a Specific Problem, e.g. not too abstract
- Proven Concept effective in practice
- Not Obvious or direct application of basic principles
- Describes Relationships, not single components
- Generative, describes how to construct a solution

However, challenges remain with the translational semantics of these formula patterns: they are not compositional and are often inconsistent with the semantics of informal definitions. Therefore, [10] introduced automata-based patterns. These are:

- Compositional: based on compositions of patterns (logic executions) and scopes (time)
- Homogeneous: don't flatten key patterns/scopes separation
- Extensible: compositional semantics allow adding patterns & scopes
- Generic: can combine any pattern and any scope
- Faithful: formal guarantee that the translated temporal formula is faithful to the intended natural semantics

While automata-based patterns correct some inconsistencies in the previous formula patterns, they present other challenges: it is often more natural for practitioners to think of specifications in terms of time lines (temporal logic) than automata, and automata patterns pose a challenge for many of the sanity checks from Section 3. Design-time formula patterns and automata patterns still do not answer the pressing question: what about *runtime specifications* for *autonomous systems*?

4.2 R2U2: Runtime Specification Patterns in the Field

Work on specification patterns focuses mostly on *design time*, which is impactful for applications such as model checking. But in today's complex, cyber-physical, and/or autonomous systems, exhaustive verification is not achievable for all subsystems; in practice, more specifications are used for applications such as runtime verification. Formula patterns are not compositional, which can be a challenge for runtime evaluation. Automata patterns are not decomposable and are more complex to sanity check, e.g., because it is easier to check satisfiability and realizability on a formula than an automaton. Yet it is vital to sanity check runtime specifications.

Therefore, we ask the question: what about *functional patterns*[1]? Are there different patterns for specification functions, e.g., between design time and runtime? In our experience with runtime verification in the field [41,18,51,49,50], we have observed the following five functional patterns.

[1] Note that the term *functional patterns* has been used in a different context: describing Requirements Specification Language (RSL) patterns for system state changes in response to external stimuli [2].

Ranges Sensors have well-defined operating ranges: both ranges of the values they can report and ranges of operation. For example, a laser altimeter has a ceiling; above this altitude its readings should not be trusted. For each sensor, we check its operating ranges and the bounds on correct values it can return.

Rates For each sensor stream on a system, there are rate constraints. We must check that value changes fall within realistic bounds, both for the sensitivity and tolerances of the individual sensor and for the physics of the system. For example, if a sensor indicates that an aircraft is falling faster than gravity, clearly there is something wrong with that sensor!

Relationships There are predictable relationships between multiple sensors; we need to compare temporal outputs from related or redundant sensors for correctness. For example, the readings from all three altimeters should be consistent, modulo sensor noise. Pitching up and increasing power to the engines should result in a rise in altitude shortly afterward.

Control Sequences A sequence of events will predictably happen following a command to take off, land, or carry out a procedure like a waypoint visit, with check-able milestones along the way A command to take off requires an ordered set of actions such as turning on the engines, taxiing, increasing altitude above ground level, and reaching a prescribed altitude. A command to land involves a series of actions in a precise order, such as an initial decrease in altitude, deploying of landing gear, and approaching the appropriate runway.

Consistency Checks Do all components have the same view of system state/environment? We consider both intra- and inter-component properties. For example, the rate of noise from a sensor should not suddenly increase. The flight computer and autopilot should always agree on which waypoint the UAS is currently visiting.

R2U2 specification format:

1. **TL Observers**: Efficient temporal reasoning
   (a) **Asynchronous**: output $\langle t, \{0, 1\}\rangle$
   (b) **Synchronous**: output $\langle t, \{0, 1, ?\}\rangle$
   - **Logics**: MTL, pt-MTL, Mission-time LTL
   - **Variables**: Booleans (from system bus), sensor filter outputs
2. **Bayes Nets**: Efficient decision making
   - **Variables**: outputs of TL observers, sensor filters, Booleans
   - **Output**: most-likely status + probability

Fig. 5. R2U2 system health management framework in a nutshell [41,50].

In industrial systems, LTL is often not the exclusive specification language. While languages and constructs for specification vary widely and are often tailored to specific applications, one general trend is the propensity for expanding upon LTL or combining

it with other specification constructs. An example of this is the specification format we use for R2U2, the Realizable, Responsive, Unobtrusive Unit for runtime system health management. Figure 5 summarizes R2U2 specifications, which combine two encodings for each linear-time temporal logic formula, which may be in one of several variants of LTL, with efficient (non-dynamic) Bayes Nets to provide diagnostic decision-making capabilities. Specifications analyzed via R2U2 are exclusively checked during runtime and do not follow previously defined patterns for formulas or automata because those describe design-time specifications consisting exclusively of temporal logic formulas.

We need to expand specification patterns to runtime! How do we expand patterns to reason about specifications in the field?

Health Nodes / Failure Modes	
H_FG	**Magnetometer sensor**
H_FC_RxUR	Receiver underrun
H_FC_RxOVR	Receiver overrun
H_FG_TxOVR	**Transmitter overrun in sensor**
H_FG_TxErr	Transmitter error in in sensor

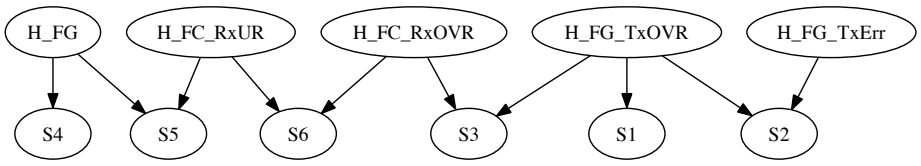

Fig. 6. The possible failures a Fluxgate Magnetometer can suffer can be diagnosed by a Bayes Net with a health node corresponding to each type of failure. These nodes take as input the valuations from six temporal logic runtime observers; many failures require inputs from multiple temporal observers in order to make an accurate diagnosis [18].

As an example, Figure 6 displays a pictorial representation of a set of specifications for determining if a fault has occurred in the fluxgate magnetometer during runtime. From the manual, we know that there are five possible faults that can occur. We can write six temporal logic specifications that we encode as runtime observers outputting statuses S1, ..., S6. The outputs from these runtime observers are inputs to five Bayesian health nodes, one for determining whether it is probabilistically likely that each possible fault has occurred. A health node may hierarchically depend on the output from more than one runtime sensor node and the runtime observers may supply temporal information to multiple health nodes.

Cyber-physical, autonomous systems often utilize hierarchical, multi-formalism specifications; see, e.g., [53]. In R2U2, we combine specifications in a way that is hierarchically structured, compositional, and cross-language. *How do we organize R2U2 specifications?*

5 Specification Organization

How should we organize specifications? How do we store specifications in an accessible way that allows for automated analysis including verification? How do we best enable re-use from design time to runtime to the design of future systems? How do we pair English and formal specifications in an understandable way? How do we preserve the hierarchical structure, compositionality, and relationships between specifications in our practical, organizational structure? Can we do all of this in a performable way?

Scenario definition languages such as the Aviation Scenario Definition Language (ASDL) [28] establish structured specification standards over domain-specific vocabulary for verification, execution, simulation, sharing, comparing, and re-using scenario specifications. This approach provides transparency to system designers via model-to-text translation, and graphical modeling environments. ASDL is an Eclipse modeling framework suited to defining scenario models for simulation, but we still need an efficient way to store and codify specifications. Most significantly, there is the question of M vs φ: how do we distinguish functions of the system model from design- and runtime specifications so that we can analyze specifications automatically and use them throughout the system lifecycle?

One can turn to an *all-in-one tool suite such as Matlab/Simulink*, but since these tools were not designed for specification organization, this solution tends to be kludgy and not scalable. Considering the often long life of specifications, which follow a system throughout its entire lifecycle, the lack of backwards-compatibility in successive tool versions presents a significant negative.

SQL databases are routinely used for longterm, scalable information storage. However, the relationships between specifications are inherently non-tabular; fitting them into this schema requires flattening the database, and accessing them requires extensive `JOIN`s, making this solution non-performable.

None of these strategies solve the organization problem. We have hit an era of *Big Data of Specifications*. If we follow recommended practices for system design, then specifications are everywhere! So, how do we organize specifications for each subsystem, subcomponent, and level of abstraction? How do we mine specifications for data, patterns, statistical analysis, and coverage? How do we search specifications? How do we sort specifications? How do we integrate specification languages for different purposes? How do we make specifications available for reuse?

5.1 A Property Graph Database Approach to Specification Organization with Neo4j

We can represent R2U2 specifications using a *property graph*.

Definition 1. *[42] A property graph $G = (V, E, \lambda, \mu)$ is a directed, edge-labeled, attributed multi-graph where V is a set of nodes, E is a multiset of directed edges, $\lambda : E \to \Sigma$ is an edge labeling function assigning a label from the alphabet Σ to each edge, and $\mu : (V \cup E) \times K \to S$ is a property assignment function over the sets K of property keys and S of property values.*

Organizing big data requires a database that can store and enable efficient access to large specification sets, so we use a *property graph database*. Neo4j[2] is a publicly available, performable, NoSQL graph database implemented in Java and Scala that efficiently implements the property graph model to allow, e.g., constant-time traversals for relationships in the graph. A property graph database stores Nodes (graph data records), and Relationships (directional connect nodes), with Properties (named data values of type string, number, Boolean, or array), on both Nodes and Relationships.

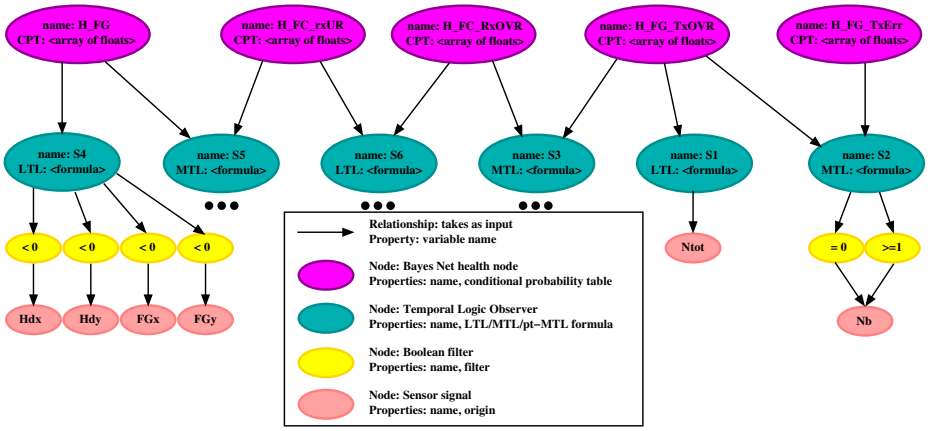

Fig. 7. A property graph database storage scheme for the Fluxgate Magnetometer failure specifications of Figure 6 with additional details from the case study in [18].

Figure 7 re-draws Figure 6 with the Neo4j database schema we are currently investigating for R2U2 specifications. We have four types of Nodes: Bayes Net health nodes that contain conditional probability tables, Temporal Logic Observer nodes that store logic formulas, Boolean filter nodes that filter direct sensor signals, and Sensor signal nodes that designate which system signals we are reasoning about. All relationships pictured are of type "takes as input" and are labeled with the name of the variable whose value is set by the given input. Note that nodes can mix properties, so we can define our Temporal Logic Observer nodes to have one type of formula, either LTL, MTL, or pt-MTL.

6 Conclusions and Outlook

Going forward, as a community, we need to continuously re-assess our answer to the question "Where are we now?" with regards to specifications. For the foreseeable future, specifications remain arguably the biggest bottleneck in formal methods and autonomy. While there are several promising research thrusts in specification debugging, updated system design processes that encourage specification extraction, and specification patterns, we still do not have a clear path forward, particularly in the context of

[2] `https://neo4j.com`

cyber-physical, autonomous systems. The questions posed by this paper of where we will get specifications from, how should we measure their quality, how should we best use them, and how should we organize them, continue to drive future research directions.

In future work, we plan to devise formal definitions of the functional specification patterns introduced here. There are many experimental evaluations in the pipeline, including use of functional specification patterns and technical analysis and performance evaluation of a new Neo4j specification organization scheme for R2U2 specifications. We also plan to advance capabilities for specification debugging, particularly satisfiability checking, and methods for efficiently reasoning about specifications in new logics now appearing in industrial settings, such as MTL [1].

6.1 Acknowledgments

Thanks to the VSTTE chairs Sandrine Blazy, Marsha Chechik, and Temesghen Kahsai for inviting this paper, which is the expansion of an invited talk delivered July 18, 2016. Thanks to Julia Badger for instigating the framing of the specification bottleneck as a series of questions for our NFM2014 panel. Thanks to André Platzer for encouraging me to update and expand on these challenges; a shorter, preliminary version of this talk appeared at the NSF Workshop on "Cyber-Physical System (CPS) Verification & Validation Industrial Challenges & Foundations (I&F): CPS and AI Safety" in May, 2016.[3] Thanks to Arie Gurfinkel, Eric Rozier, and Johann Schumann for technical discussions on earlier drafts of this paper. Information on our recent work can be found at: `http://laboratory.temporallogic.org`.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)
2. Backes, J.D., Whalen, M.W., Gacek, A., Komp, J.: On implementing real-time specification patterns using observers. In: NFM. pp. 19–33. Springer (2016)
3. Badger, J., Rozier, K.Y. (eds.): NFM, LNCS, vol. 8430. Springer (May 2014)
4. Badger, J., Rozier, K.Y.: Panel: Future directions of specifications for formal methods. In: Badger, J., Rozier, K.Y. (eds.) NFM. LNCS, vol. 8430, pp. XX–XXI. Springer (May 2014)
5. Badger, J., Throop, D., Claunch, C.: Vared: Verification and analysis of requirements and early designs. In: Requirements Engineering. pp. 325–326. IEEE (2014)
6. Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J., Kratochvíla, T.: Analysing sanity of requirements for avionics systems. Formal Aspects of Computing 28(1), 45–63 (2016)
7. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. Formal Methods in System Design 18(2), 141–162 (2001)
8. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY–a new requirements analysis tool with synthesis. In: CAV. pp. 425–429. Springer (2010)
9. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL synthesis. In: CAV. pp. 652–657. Springer (2012)

[3] `http://www.ls.cs.cmu.edu/CPSVVIF-2016/index.html`

10. Castillos, K.C., Dadeau, F., Julliand, J., Kanso, B., Taha, S.: A compositional automata-based semantics for property patterns. In: IFM. pp. 316–330. Springer (2013)
11. Chockler, H., Kupferman, O., Kurshan, R.P., Vardi, M.Y.: A practical approach to coverage in model checking. In: CAV. pp. 66–78. Springer (2001)
12. Cimatti, A., Roveri, M., Schuppan, V., Tchaltsev, A.: Diagnostic information for realizability. In: VMCAI. pp. 52–67. Springer (2008)
13. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA. pp. 85–96. ACM (2010)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP. pp. 7–15. ACM (1998)
15. Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.: A framework for inherent vacuity. In: HVC. pp. 7–22. LNCS 5394, Springer (2008)
16. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: CAV. LNCS, vol. 9780, pp. 3–22. Springer (2016)
17. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. Presentation: https://es-static.fbk.eu/projects/nasa-aac/download/CAV2016_presentation.pdf#21 (2016-07-22)
18. Geist, J., Rozier, K.Y., Schumann, J.: Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In: RV. vol. 8734, pp. 215–230. Springer-Verlag (2014)
19. Gheorghiu, M., Gurfinkel, A., Chechik, M.: VaqUoT: A tool for vacuity detection. Posters & research tools track, FM 2006 (2006)
20. Ghosh, S., Shankar, N., Lincoln, P., Elenius, D., Li, W., Steiener, W.: Automatic Requirements Specification Extraction from Natural Language (ARSENAL). Tech. rep., DTIC Document (2014)
21. Gross, K.H., Fifarek, A.W., Hoffman, J.A.: Incremental formal methods based design approach demonstrated on a coupled tanks control system. In: HASE. pp. 181–188. IEEE (2016)
22. Gundy-Burlet, K.: Validation and verification of LADEE models and software. In: 51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition (2013)
23. Gurfinkel, A., Chechik, M.: Robust vacuity for branching temporal logic. ACM Transactions on Computational Logic (TOCL) 13(1), 1 (2012)
24. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. 5(3), 231–261 (1996)
25. Hoffman, J.A.: Utilizing assume guarantee contracts to construct verifiable simulink model blocks. S5: http://mys5.org/Proceedings/2015/Day_1/2015-S5-Day1_1255_Hoffman.pdf (2015)
26. Hoffman, J.A.: V&V of Autonomy: UxV Challenge Problem (UCP). S5: http://mys5.org/Proceedings/2016/Day_3/2016-S5-Day3_0805_Hoffman.pdf (2016)
27. Jackson, C.: Face it: The engineering V is outdated. https://www.linkedin.com/pulse/20140721140340-5687591-face-it-the-engineering-v-is-outdated (2014)
28. Jafer, S., Chhaya, B., Durak, U., Gerlach, T.: Formal scenario definition language for aviation: Aircraft landing case study. In: AIAA MST (2016)
29. Könighofer, R., Hofferek, G., Bloem, R.: Debugging unrealizable specifications with model-based diagnosis. In: HVC. pp. 29–45. Springer (2010)
30. Kupferman, O.: Sanity checks in formal verification. In: CONCUR, Proc. 17th Int'l Conf. LNCS, vol. 4137, pp. 37–51. Springer (2006)
31. Kupferman, O., Vardi, M.: Vacuity detection in temporal model checking. J. on Software Tools For Technology Transfer (STTT) 4(2), 224–233 (Feb 2003)

32. Kurshan, R.: FormalCheck User's Manual. Cadence Design, Inc. (1998)
33. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: TIME. pp. 91–98. IEEE (2013)
34. Li, J., Zhu, S., Pu, G., Vardi, M.Y.: SAT-Based Explicit LTL Reasoning. In: HVC. pp. 209–224. Springer (2015)
35. Li, W., Dworkin, L., Seshia, S.A.: Mining assumptions for synthesis. In: MEMOCODE. pp. 43–50. IEEE (2011)
36. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: Specification. Springer Science & Business Media (2012)
37. Mattarei, C., Cimatti, A., Gario, M., Tonetta, S., Rozier, K.Y.: Comparing different functional allocations in automated air traffic control design. In: FMCAD. IEEE/ACM (2015)
38. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innovations in Systems and Software Engineering 9(4), 235–255 (2013)
39. Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive (1) designs. In: VMCAI. pp. 364–380. Springer (2006)
40. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190 (1989)
41. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: TACAS. LNCS, vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
42. Rodriguez, M.A., Neubauer, P.: The graph traversal pattern. arXiv preprint arXiv:1004.1001 (2010)
43. Rozier, K., Vardi, M.: LTL satisfiability checking. In: SPIN. LNCS, vol. 4595, pp. 149–167. Springer-Verlag (2007)
44. Rozier, K., Vardi, M.: LTL satisfiability checking. International Journal on Software Tools for Technology Transfer (STTT) 12(2), 123 – 137 (March 2010)
45. Rozier, K., Vardi, M.: A multi-encoding approach for LTL symbolic satisfiability checking. In: FM. LNCS, vol. 6664, pp. 417–431. Springer-Verlag (2011)
46. RTCA: DO-178B: Software Considerations in Airborne Systems and Equipment Certification (1992), `http://www.rtca.org`
47. RTCA: DO-254: Design assurance guidance for airborne electronic hardware (April 2000)
48. RTCA: DO-178C/ED-12C: Software considerations in airborne systems and equipment certification (2012), `http://www.rtca.org`
49. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In: RV. Springer-Verlag (2015)
50. Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime Analysis with R2U2: A Tool Exhibition Report. In: RV. Springer-Verlag, Madrid, Spain (September 2016)
51. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. IJPHM 6(1), 1–27 (June 2015)
52. Vardi, M.Y.: From verification to synthesis. VSTTE 5295, 2 (2008)
53. Whalen, M.W., Rayadurgam, S., Ghassabani, E., Murugesan, A., Sokolsky, O., Heimdahl, M.P., Lee, I.: Hierarchical multi-formalism proofs of cyber-physical systems. In: MEMOCODE. pp. 90–95. IEEE (2015)
54. Zeller, A.: Specifications for free. In: NFM. pp. 2–12. Springer (2011)
55. Zhao, Y., Rozier, K.Y.: Formal specification and verification of a coordination protocol for an automated air traffic control system. In: AVoCS. Electronic Communications of the EASST, vol. 53 (2012)
56. Zhao, Y., Rozier, K.Y.: Formal specification and verification of a coordination protocol for an automated air traffic control system. SCP Journal 96(3), 337–353 (December 2014)
57. Zhao, Y., Rozier, K.Y.: Probabilistic model checking for comparative analysis of automated air traffic control systems. In: ICCAD. pp. 690–695. IEEE/ACM (2014)