Analysis and Monitoring of Cyber-Physical Systems via Environmental Domain Knowledge & Modeling

Byron DeVries
School of Computing
Grand Valley State University
Allendale, Michigan
Email: devrieby@gvsu.edu

Erik M. Fredericks
School of Computing
Grand Valley State University
Allendale, Michigan
Email: frederer@gvsu.edu

Betty H.C. Cheng
Computer Science and Engineering
Michigan State University
East Lansing, Michigan
Email: chengb@msu.edu

***Abstract*—While verifying adherence to a specification (i.e., specification-based testing) is important, the results are only as valid as the specification itself. Problematically, verifying a system specification must be done within the context of changing or even unknown environmental domain knowledge that could render the specification ineffective or incorrect. This issue is even more apparent in the context of self-adaptive systems, where uncertainty in both the system configuration and environment can impact the validity of the system. This paper introduces a method to explicitly model domain knowledge of the environment as a secondary system to enable design-time verification against documented environmental assumptions (i.e., those elements external to the system). In addition, run-time monitors are used to detect scenarios in the actual environment not specified by the modeled environmental domain knowledge. Rather than simply identifying unexpected inputs, our approach is able to identify run-time violations of the environmental domain knowledge, even when inputs appear valid based on the domain assumptions embedded in the system specification. These violations can then be used to correspondingly update the system and environmental specifications via automated run-time adaptation or subsequent design-time revisions. We illustrate our approach by applying our method to a running example of a goal-based model of a baby monitor.**

***Index Terms*—specification, cyber-physical systems, uncertainty**

I. Introduction

For assurance purposes, cyber-physical systems (CPS) should be verified across all possible environmental scenarios. However, it is infeasible to enumerate all possible environmental conditions that a system may encounter [1]–[4]. Verifying the system for only expected or enumerated environmental scenarios based on our domain knowledge is insufficient to ensure correct operation in the face of unexpected or changing scenarios. Worse, a CPS may *appear* functional when operating in an environmental scenario that was previously unknown or undocumented (i.e., unenumerated) due to inconsistencies between the domain knowledge and reality (i.e., the reality gap). A CPS, therefore, must be able to recognize and mitigate unexpected environmental scenarios outside of the documented environmental domain knowledge where the system and its own internal measures of success (e.g., run-time monitors) have not been verified. This paper introduces *Aether*,[1] an approach to more rigorously and explicitly model domain knowledge as the range of environmental scenarios used to verify the system at design time, as well as recognize unexpected and unverified environmental scenarios at run time.

When a requirements specification is completely free from implementation details, the specification describes the system based on its interactions with the environment [5]. However, methods that ensure correct behavior for CPSs typically limit the enumerated environmental scenarios to a finite set of obstacles [6], [7] or attempt to verify the entire range of possible environmental inputs to the CPS [8]–[10]. These two methods are at opposite ends of the spectrum for representing domain knowledge of the environment. Fundamentally, the specification (S) of a system can only provably satisfy the user requirements (R) with the addition of the appropriate domain knowledge (K), as described by Zave and Jackson [5] and represented in the following expression:

$$S, K \vdash R. \tag{1}$$

Problematically, enumerating a finite set of obstacles representing domain knowledge leads to additional obstacles that are unrepresented due to unknowns or changing domain knowledge. However, exploring all possible environmental scenarios often includes unrealistic environmental scenarios and false positives where unrealistic or infeasible environmental scenarios are needlessly verified.

Verification of a specification must be within the context of applicable environmental scenarios based on domain knowledge. Rather than enumerating all possible obstacles to success (i.e., blacklisting), enumerating a set of known acceptable environmental scenarios (i.e., whitelisting) enables classification of unverified environmental scenarios due to unknowns or changing domain knowledge. This paper introduces *Aether*, a method to explicitly model domain knowledge of the environment as a secondary system to enable design-time verification against documented environmental assumptions. Run-time monitors are used to detect scenarios in the actual environment not enumerated by the modeled domain knowledge. Rather than simply identifying unexpected inputs, run-time violations of the domain model detect feasible environmental scenarios that are outside of the existing domain knowledge and are otherwise not exhaustively enumerable. The information gained via *Aether* can then be used in a self-

[1] *Aether* is the Greek primordial god of light, air, and atmosphere that filled the environment between the ground and the sky.

adaptive system feedback loop (e.g., MAPE-K [11]) to support adaptation strategies.

The contributions of this paper are as follows:

- We present a method for modeling environmental domain knowledge as an interacting system,
- We introduce a design-time method to analyze a specification limited to enumerated environmental scenarios documented in the modeled domain knowledge, and
- We introduce a run-time method to detect unexpected, and thus unverified at design-time, environmental scenarios due to inconsistencies between the documented domain knowledge of the environment and the actual environment (i.e., the reality gap).

The remainder of this paper is organized as follows. Section II provides an overview of the background information. Section III introduces the *Aether* approach via a running example. Section IV covers related work, and Section V discusses conclusions and future work.

II. Background

This section covers background in Goal-Oriented Requirements Engineering (GORE) and run-time monitors while describing our example system.

A. Goal-Oriented Requirements Engineering

GORE is an approach for guiding the elicitation and analysis of system requirements in a graphical, goal-oriented manner by specifying objectives and constraints [12]. Modeling of a system with GORE decomposes high-level goals into sub-goals using a directed, acyclic graph [12], where each edge between goals represents a *goal refinement*.[2] Goal refinements may be enabled using AND and OR refinements. AND-refined goals are satisfied if *all* of their sub-goals are also satisfied, and OR-refined goals are satisfied if at least *one* of their sub-goals is satisfied. For this paper, we also consider XOR- (i.e., exclusive OR) refined goals that require *exactly one* sub-goal to be satisfied. Goal refinement continues until all leaf-level goals have been assigned an agent responsible for goal satisfaction. Leaf-level goals are considered to be requirements/expectations where requirements are satisfied by agents of the system and expectations are satisfied by agents of the environment. For example, Figure 1 presents a realistic, but simplified, baby monitor with the top-level goal to maintain the safety of the child.[3] In the case that the child is currently safe, as indicated by a threshold weight (i.e., one pound, an amount sufficient to establish the presence of a child of unknown weight) and motion has occurred within the last 3 seconds, no alarm is needed. Otherwise, the weight threshold is not met or there is no motion, either of which necessitates an alarm. For clarity, goal model elements are referred to using sans serif text (e.g., "Maintain(Safe Child)").

[2]Note that we do not follow the formal KAOS decomposition strategy [12], [13] for this paper.

[3]The baby monitor presented in this paper closely represents a baby monitor used by the primary author's first child.

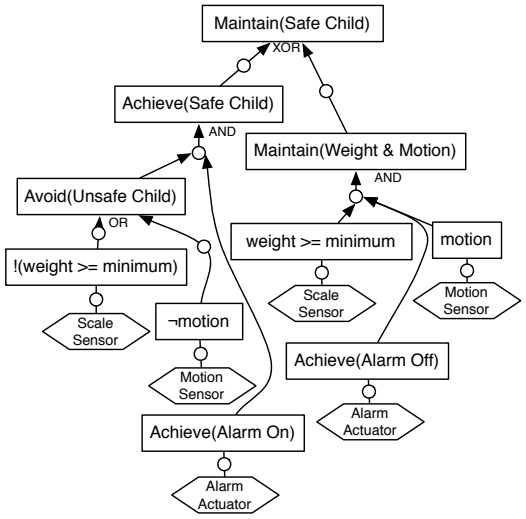

Fig. 1. Baby Monitor System

B. Run-Time Monitors

Run-time monitors provide a method for understanding the behavior of a system at run time. One such technique for enabling run-time monitoring is utility functions, where a utility function is a mathematical formula that has been derived, per requirement (or goal), to quantify its relative level of satisfaction (i.e., satisficement) [14]–[16].

Utility functions are defined for the goals, expectations, and requirements in Figure 1. The utility functions for goals are based on the composed utility functions of the decomposed elements (i.e., goals, requirements, or expectations) using the decomposition operator (i.e., AND, OR, or XOR). Similar to the state-, metric-, or fuzzy-logic based satisficement functions derived using a relationship property in Athena [15], *Aether* state-based utility functions are derived directly from the text for expectations (e.g., weight $\geq$ minimum) while utility functions for the requirements are defined by the system designer. The requirement utility functions are shown in Table I.

TABLE I
Baby Monitor Requirement Utility Functions

Requirement	Utility Function
Achieve(Alarm On)	alarm
Achieve(Alarm Off)	¬alarm

III. Approach & Running Example

This section details the steps that comprise *Aether* and provides a running example. The system designer manually defines models of both the system and the environment relevant to the system as the first step. Subsequently, in the second step, *Aether* generates analysis logic derived from the previously manually-defined models. In Step 3, *Aether* performs a design-time analysis of the system and enumerated environment. Any errors detected in the system in Step 3 are returned to the system designer to facilitate manual updates to the system and environmental models. Alternative to the design-time analysis, Step 4 generates run-time analysis code that can

be incorporated into the modeled CPS (and executed via Step 5) to enable the system to detect unexpected environmental scenarios. Next, we describe each of the steps in turn using the baby monitor system as a running example.

A. Step (1): Model Environment Domain Knowledge & System

In an initial manual step, the system designer models the environment domain knowledge and system to represent a set of enumerated environmental scenarios based on domain knowledge and a set of system specifications (i.e., S), respectively. The system specification follows the standard procedure for GORE: a top-level goal is defined (e.g., "Maintain(Safe Child)" in Figure 1) and decomposed into additional goals, requirements, and expectations. Goals are decomposed until all leaf-level elements are either requirements, that can be satisfied by a single agent of the system, or expectations, that can be satisfied by a single agent of the environment. For example, in Figure 1, the requirement "Achieve(Alarm On)" can be satisfied by a single agent of the system (i.e., the Alarm Actuator). Similarly, expectation "motion" can be satisfied by a single agent of the environment (i.e., the Motion Sensor).

When modeling the system, we call any agent of the environment an instance of a *sensor*, while any agent of the system is an instance of an *actuator*. Agents of the environment are *inputs* to the system, while agents of the system are *outputs* of the system that have an impact on the environment. From the perspective of the system, *sensors* are *inputs* that the system cannot control while *actuators* are *outputs* that the system can control. The environment, based on domain knowledge, is modeled similarly as a second interacting system where an *input* to the environment is supplied by an *output* from the system and vice-versa. The decomposition of the system and the environment match along the system boundary as defined by the agents of the system.

Figure 2 describes such an *environmental* goal model, based on domain knowledge, with sensors and actuators that correspond to the sensors and actuators in the *system* goal model described in Figure 1. The environmental domain knowledge for the baby monitor includes cases when the child is either in the crib or not in the crib. If the child is not in the crib, then there must be no motion and a lack of sufficient weight. In this case, the environmental expectation is that the alarm would be produced by the system. Note: the environmental model not only describes the scenario enumerated by the domain knowledge, but *also* the expected results from the system based on that domain knowledge. That is, the environment domain knowledge *verifies* the expected response from the system as the goal model specification describes the system based on its interactions with the environment. In the case where the child is in the crib, two additional cases may exist. First, the child may be both moving (e.g., breathing) and providing sufficient weight to be detected, resulting in an environmental expectation of no alarm (i.e., ¬alarm). Second, the child may not be moving but is providing sufficient weight to be detected, resulting in an environmental expectation of an alarm (i.e., alarm) due to a non-breathing child in the crib.

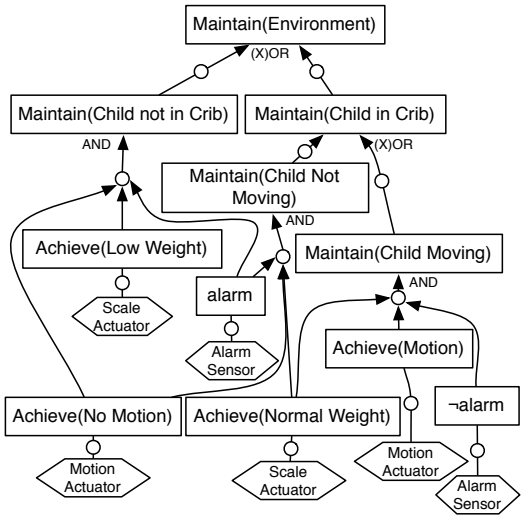

Fig. 2. Baby Monitor Environment

The utility functions for environmental goals are based on the composed utility functions of the decomposed elements, just as with the system goals. The utility functions for expectations are derived directly from the expression of the expectation (e.g., ¬alarm), and utility functions for the requirements are defined by the system designer, as shown in Table II.

TABLE II
ENVIRONMENTAL REQUIREMENT UTILITY FUNCTIONS

Requirement	Utility Function
Achieve(No Motion)	¬motion
Achieve(Motion)	motion
Achieve(Low Weight)	weight $\geq 0.0 \wedge$ weight < 1.0
Achieve(Normal Weight)	weight $\geq 1.0 \wedge$ weight < 15.0

The system and environmental goal models, along with their utility functions, are used to generate logical representations of the goal models in the subsequent step.

B. Step (2): Generate Analysis Logic

Given a system and environmental goal model representing the domain knowledge, along with their utility functions, *Aether* automatically generates logical expressions for evaluating utility functions defined for each of the goal models. These logical expressions are used to measure the satisfaction of the goals, requirements, and expectations. In this instance, the utility function for "Maintain(Safe Child)" in the system goal model represented in Figure 1 represents the satisfaction of the system as a whole, based on the composition of the leaf-level utility functions representing requirements and expectations using the decompositional operators (e.g., OR, AND, and XOR). The environmental goal model is treated similarly where "Maintain(Environment)" is the top-level goal in the environmental goal model in Figure 2. Importantly, a satisfied top-level utility function for the environment indicates the current environmental scenario is in the enumerated set of domain knowledge. For the remainder of the paper, the following sets are used to refer to the system and environment, respectively:

- S refers to the set of scenarios where the goal model of the system (i.e., "Maintain(Safe Child)") is measured as satisfied by the composition of its utility functions (i.e., *utility*(Maintain(Safe Child))).
- $K_{\text{enumerated}}$ refers to the set of environmental scenarios where the goal model of the environment (i.e., "Maintain(Environment)") is measured as satisfied by the composition of its utility functions (i.e., *utility*(Maintain(Environment))).

The intent is for the system designer to decompose the system goal model such that the *set of enumerated domain knowledge environmental scenarios* is a subset of the *set of environmental scenarios* where the system goal model is satisfied (i.e., $K_{\text{enumerated}} \subseteq S$). That is, the system goal model is designed to be successful for all the known environmental scenarios (i.e., "known-knowns"). This property, along with several others, is analyzed in the next step.

C. Step (3): Design-Time Analysis

Using a Satisfiability Modulo Theory (SMT) Solver, the logical expressions representing the composed system utility functions and composed environmental utility functions are analyzed for counterexamples. Table III enumerates the possible memberships of a single scenario, i, in the sets S and $K_{\text{enumerated}}$, representing system and environmental utility function satisfaction for the given scenario, respectively.

TABLE III
SYSTEM AND ENVIRONMENTAL SATISFACTION

$i \in S$	$i \in K_{\text{enumerated}}$	**Description**
false	false	The system *appears* to fail in an unexpected scenario outside of the domain knowledge.
false	true	The scenario is enumerated by the domain knowledge (i.e., expected), but the system fails.
true	false	The system *appears* to succeed in an unexpected scenario (i.e., outside domain knowledge).
true	true	The system is known to succeed in an expected (e.g., enumerated) scenario.

The system should succeed in any scenario enumerated by the domain knowledge, since it is expected (i.e., a "known-known"). Measured using utility functions for the system and environmental goal models previously introduced (i.e., Figure 1 and Figure 2, respectively), we expect that:

$$\textit{utility}(\text{Maintain(Environment)}) \implies \textit{utility}(\text{Maintain(Safe Child)})$$

If the environmental scenario is within the domain knowledge of the environment, then the system goal model is expected to also be satisfied. That is, the system should succeed in all known and expected scenarios. A counterexample exists when the scenario is in the environmental domain knowledge (i.e., the environmental goal model is satisfied), but the system fails to succeed (i.e., the system goal model is not satisfied):

$$\textit{utility}(\text{Maintain(Environment)}) \wedge \neg \textit{utility}(\text{Maintain(Safe Child)})$$

If no counterexamples exist, then the system has been designed to operate successfully within the context of the explicitly defined environmental model.

D. Step (4): Generate Run-Time Analysis

Problematically, even if the system is designed to operate successfully within the context of the environmental domain knowledge, unexpected environmental scenarios threaten the successful operation of the system. Intrinsically, the environment, as it actually exists, is always satisfied (i.e., any real-world environmental scenario would be in K_{feasible}). However, the set of environmental scenarios that are feasible in the real-world (i.e., K_{feasible}) is unlikely to be equivalent to environmental scenarios enumerated via domain knowledge (i.e., $K_{\text{enumerated}}$) in the environmental goal model. That is, there *may* be environmental scenarios such that an individual scenario, i:

- is enumerated in the domain knowledge, but is infeasible and will not occur (i.e., $i \in K_{\text{enumerated}} \wedge i \notin K_{\text{feasible}}$),
- is enumerated in the domain knowledge, and is feasible (i.e., $i \in K_{\text{enumerated}} \wedge i \in K_{\text{feasible}}$), and
- is not enumerated in the domain knowledge, but is feasible (i.e., $i \notin K_{\text{enumerated}} \wedge i \in K_{\text{feasible}}$).

Importantly, while our domain knowledge as specified in the system is typically fixed (i.e., $K_{\text{enumerated}}$), the reality (i.e., K_{feasible}) changes over time. That is, the set of possible environmental scenarios is not fixed over time and domain knowledge must be able to adapt. The system has been verified against the explicit environmental domain knowledge in the enumerated environmental scenarios and any enumerated but infeasible scenarios do not currently matter as they cannot occur. The feasible scenarios that are unenumerated in the domain knowledge have no guarantees for system behavior based on the design-time analysis. The reasoning for unexpected environmental scenarios outside of the domain knowledge includes, but is not limited to:

- The infeasibility of exhaustively enumerating all possible scenarios at design-time,
- A specific scenario may be mistakenly thought to be infeasible and, therefore, not included in the enumerated scenarios, or
- An infeasible scenario became feasible due to an environmental or system change after system deployment.

Regardless, it is likely that feasible, but unenumerated, scenarios will arise during run-time operations of a CPS due to limited or changing environmental domain knowledge.

Any feasible environmental scenario that the system would encounter at run time must occur in the K_{feasible} set. Further, any feasible environmental scenario where the system has previously been verified in Step (3) must exist in both sets K_{feasible} and $K_{\text{enumerated}}$ (i.e., $K_{\text{enumerated}} \bigcap K_{\text{feasible}}$). Problematically, for the previously listed reasons, it is impractical and/or impossible to enumerate the extent of the feasible region (i.e., K_{feasible}) at design time.

However, using utility functions that measure the satisfaction of the environment domain knowledge and system, it is

possible to classify the following sets regardless of changes to the feasible scenarios:

- $K_{\text{enumerated}}$ only: Infeasible at run time and will not occur.
- $K_{\text{enumerated}} \bigcap K_{\text{feasible}}$: Real-world scenarios for which the system was designed and analyzed.
- K_{feasible} only: While *not enumerable*, this region is detectable given a *real world* scenario that is not in $K_{\text{enumerated}}$.

For the baby monitor goal model and the associated environment domain knowledge, the K_{feasible} set is measured as real-world environmental scenarios, experienced at run time. An unsatisfied environmental goal model utility function (e.g., *utility*(Maintain(Environment))) indicates the scenario was not enumerated (i.e., not in $K_{\text{enumerated}}$). Therefore, regardless of the perceived satisfaction of the system, as measured by the system goal model utility function *utility*(Maintain(Safe Child)), the system has not been verified in design-time analysis. In such environmental scenarios where the scenario is unenumerated in the domain knowledge a fail-safe, fail-silent, and/or adaptation is needed.

E. Step (5): Execute Monitoring System

The generated run-time monitoring code, combined with the CPS code, is compiled and deployed within the system under study. The monitoring system is executed each time the sensors or actuators are read from or written to, respectively.

When an environmental scenario is detected as outside of the environmental domain knowledge (i.e., unenumerated) as measured by the utility function *utility*(Maintain(Environment)), the system utility function (i.e., *utility*(Maintain(Safe Child))), and therefore the system itself, *appears* to be successful or unsuccessful. In either case, the system result has not been verified and cannot be trusted. Therefore, a run-time adaptation or future design-time system revision is required. For example, consider the unexpected environmental scenario resulting in the *apparently* successful system shown in Figure 3. Representations of the utility functions are included in the requirements and expectations. Satisfied portions of the goal model are illustrated with thick bold lines, while unsatisfied portions are illustrated with thin dashed lines.

In this scenario, the following requirements and expectations in the system goal model are true:

- Motion is detected (motion $==$ true),
- The measured weight (i.e., 18.0 pounds) is over the minimum threshold (i.e., 1.0 pounds), and
- The alarm is off or has been turned off (alarm $==$ false).

In a typical run-time monitoring system, the baby monitor CPS would be considered to be successful. However, consider the unexpected and dangerous scenario of a cat joining the child in the crib. While the warmth of a young child may be an appealing place to sleep, a cat may also smother the child with which it is sleeping - a common concern for new parents with cats [17]. The baby monitor system, however, would continue to register motion from the cat, and weight from both the

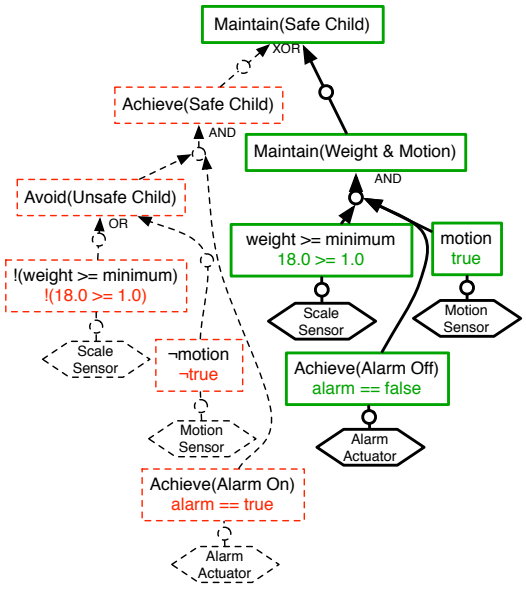

Fig. 3. Baby Monitor System Scenario

cat and child, and sound no alarm. This devastating scenario renders the baby monitor useless.

Detection of such an unexpected environmental scenario that is not enumerated in our domain knowledge is possible using the explicit environmental domain knowledge goal model defined in Figure 2. Using the same scenario shown in Figure 3, we can determine the satisfaction of the utility function *utility*(Maintain(Environment)). Figure 4, which illustrates that the environmental scenario is, in fact, unenumerated in the domain knowledge and therefore unexpected. Consequently, we know that we cannot accept the result of the system utility function alone as the system has not been analyzed at design time in this previously unknown and unenumerated environmental scenario. As before, thick bold lines and thin dashed lines are used to represent satisfied and unsatisfied portions of the goal model, respectively.

Given that we cannot trust the result of the system utility function solely due to an unexpected environmental scenario, an automated mitigation to unexpected adverse scenarios (e.g., fail-safe, fail-silent, and/or adaptation) or documenting the unexpected scenario to support later revision by the developer is required. In the case presented with the baby monitor, a fail-safe alarm and/or alert would be an appropriate mitigation.

F. Limitations

An explicit environmental model of the domain knowledge enables us to detect unexpected (i.e., unenumerated) environmental scenarios not in the domain knowledge, and verify enumerated environmental scenarios within the domain knowledge. However, if enumerated environmental scenarios cannot be differentiated with the existing set of agents, a superficially similar scenario (e.g., two different objects of similar weight that move where only one is a child) may be treated incorrectly as an enumerated scenario within the domain knowledge. Similarly, a bad sensor or actuator may cause the system to

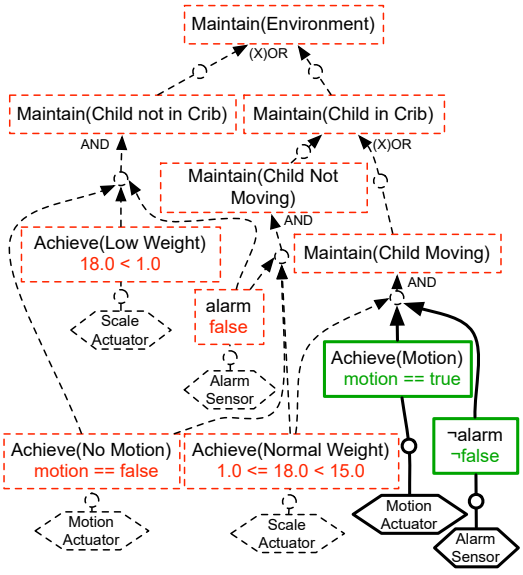

Fig. 4. Baby Monitor Enumerated Environment Scenario

incorrectly assess the system or environment (e.g., a weight sensor is "stuck" and continues to return the same weight regardless of changes). This limitation is based on the current design of the system having access to only the "shared" actions of the environment, while being unaware of the "unshared" actions [5]. For example, consider a child that exits a crib at the same moment that a cat enters the same crib. The alarm should sound due to the lack of a child. However, given the existing set of actors, the scenario is indistinguishable from a child alone in the crib. Additional actors would be needed to shift existing "unshared" actions to "shared" actions in order to differentiate.

IV. Related Work

This section presents related work on uncertainty in the context of CPSs. Uncertainty can impact CPSs in numerous ways, including inducing hardware faults, causing timing violations, and fuzzing sensor readings. As CPSs tend to be safety-critical systems, such violations can result in loss of life, disruption of critical manufacturing processes, and loss of user trust. To this end, Zhang *et al.* introduced the *U-model* as a UML/OCL-based formalism for modeling uncertainty in CPSs [18]. This model includes extensions to represent uncertainty via multiple mechanisms, including Beliefs, Evidence, and Measurements. Similarly, MARTE introduced UML refinements to support real-time/embedded systems [19]. *UncerTum* was later released as a framework for generating test-ready models using the *U-model* [20]. Conversely, *Aether* models uncertainty at the goal level, prior to UML refinement. Uncertainty in the CPS domain has also been shown to be reduced via formal verification using mathematically-defined models of computation, albeit in a simulation environment [21]. Similarly, SMT solvers have been applied to reduce uncertainty as well for temporal concerns, however not specifically in the context of the CPS domain [22]. Rather than directly represent uncertainty, we represent the known-known (i.e., enumerated) environmental scenarios to detect those that are unexpected (i.e., unknown-unknowns or known-unknowns) or even undocumented (i.e., unknown-knowns).

Obstacles to goal achievement and expectations [6], [7], or prerequisites to system success [12], have been used in KAOS goal modeling to model portions of the environment. *Aether* extends system modeling by enumerating expected scenarios rather than enumerating a potentially changing set of obstacles while verifying the system against environmental expectations.

To highlight the difficulties in mitigating uncertainty in the CPS domain, Zheng and Julien have presented a research roadmap for supporting verification and validation for CPSs, positing that existing tools require extensions to support specific CPS concerns [23]. To this end, we introduce extensions to the GORE process to support reduction of uncertainty in a goal-oriented fashion. Hissam *et al.* [24] have discussed multiple levels of assurance for distributed, adaptive real-time (DART) systems via formal specification, verification, and code generation. *Aether* similarly introduces formal verification via uncertainty specification, albeit at the goal level for an explicit model of environmental domain knowledge.

Uncertainty is especially problematic in CPSs that are self-adaptive. Previously, Esfahani and Malek provided an in-depth overview of the different implications that uncertainty can have for self-adaptive systems [25]. Esfahani *et al.* then introduced POISED, an approach for quantifying uncertainty using possibilistic approaches [26]. Additionally, Bencomo has used Bayesian surprise to detect statistically unexpected scenarios [27], [28]. Uncertainty has also been considered from the goal model level [29], including via fuzzy-logic functions in RELAX [30] and FLAGS [31]. Perez-Palacin and Mirandola have also provided a taxonomy of the varying sources of uncertainty that impact self-adaptive systems, including associated techniques available for mitigation [32]. *Aether* aims to support existing techniques for mitigating uncertainty via a goal-based formal approach, rather than a probabilistic approach.

V. Conclusions

In this paper, we have presented *Aether*, a design-time and run-time approach to explicitly model environmental domain knowledge, verify expected system operation in the context of that domain knowledge, and detect unexpected scenarios at run time outside of our documented domain knowledge.

We demonstrated *Aether* on a set of goal models that describe a baby monitor and its associated environmental domain knowledge. We showed that *Aether* is able to automatically detect environmental scenarios outside of the explicitly documented and design-time analyzed environmental domain knowledge.

Future research directions include exploring how *Aether* can be extended to address temporal and RELAXed properties [30], [33], as well as the distribution of sensors for multi-system analysis.

REFERENCES

[1] B. H. C. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26.

[2] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 468–483.

[3] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[4] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein, "Requirements-aware systems: A research agenda for re for self-adaptive systems," in *2010 18th IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 95–103.

[5] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 1, pp. 1–30, 1997.

[6] A. Van Lamsweerde and E. Letier, "Integrating obstacles in goal-driven requirements engineering," in *Proceedings of the 20th international conference on Software engineering*. IEEE, 1998, pp. 53–62.

[7] ——, "Handling obstacles in goal-oriented requirements engineering," *IEEE Transactions on software engineering*, vol. 26, no. 10, pp. 978–1005, 2000.

[8] A. J. Ramirez, A. C. Jensen, B. H. C. Cheng, and D. B. Knoester, "Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 568–571.

[9] M. A. Langford and B. H. C. Cheng, "Enhancing learning-enabled software systems to address environmental uncertainty," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2019, pp. 115–124.

[10] M. A. Langford, G. A. Simon, P. K. McKinley, and B. H. C. Cheng, "Applying evolution and novelty search to enhance the resilience of autonomous systems," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 63–69.

[11] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, no. 1, pp. 41–50, 2003.

[12] A. Van Lamsweerde *et al.*, "Requirements engineering: from system goals to uml models to software specifications," 2009.

[13] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of computer programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[14] P. DeGrandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 107–116.

[15] A. J. Ramirez and B. H. C. Cheng, "Automatic derivation of utility functions for monitoring software requirements," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 501–516.

[16] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, 2004, pp. 70–77.

[17] M. S. Kearney, L. B. Dahl, and H. Stalsberg, "Can a cat smother and kill a baby?" *British medical journal (Clinical research ed.)*, vol. 285, no. 6344, p. 777, 1982.

[18] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding uncertainty in cyber-physical systems: a conceptual model," in *European conference on modelling foundations and applications*. Springer, 2016, pp. 247–264.

[19] OMG, "UML profile for modeling and analysis of real-time and embedded systems (MARTE)," 2006.

[20] M. Zhang, S. Ali, T. Yue, R. Norgren, and O. Okariz, "Uncertainty-wise cyber-physical system test modeling," *Software & Systems Modeling*, vol. 18, no. 2, pp. 1379–1418, 2019.

[21] C. Radojicic, C. Grimm, A. Jantsch, and M. Rathmair, "Towards verification of uncertain cyber-physical systems," *arXiv preprint arXiv:1705.00519*, 2017.

[22] A. Cimatti, A. Micheli, and M. Roveri, "Solving strong controllability of temporal problems with uncertainty using smt," *Constraints*, vol. 20, no. 1, pp. 1–29, 2015.

[23] X. Zheng and C. Julien, "Verification and validation in cyber physical systems: research challenges and a way forward," in *2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems*. IEEE, 2015, pp. 15–18.

[24] S. A. Hissam, S. Chaki, and G. A. Moreno, "High assurance for distributed cyber physical systems," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, 2015, pp. 1–4.

[25] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 214–238.

[26] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 234–244.

[27] N. Bencomo and A. Belaggoun, "A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 460–463.

[28] N. Bencomo, "Quantun: Quantification of uncertainty for the reassessment of requirements," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*. IEEE, 2015, pp. 236–240.

[29] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, "Living with uncertainty in the age of runtime models," in *Models@ run. time*. Springer, 2014, pp. 47–100.

[30] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 79–88.

[31] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010, pp. 125–134.

[32] D. Perez-Palacin and R. Mirandola, "Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation," in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, 2014, pp. 3–14.

[33] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "Relax: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010.