Software and Systems Modeling (2023) 22:1543–1563
https://doi.org/10.1007/s10270-023-01090-9

SPECIAL SECTION PAPER

MoDALAS: addressing assurance for learning-enabled autonomous systems in the face of uncertainty

Michael Austin Langford[1] · Kenneth H. Chan[1] · Jonathon Emil Fleck[1] · Philip K. McKinley[1] · Betty H.C. Cheng[1]

Received: 14 March 2022 / Revised: 26 January 2023 / Accepted: 30 January 2023 / Published online: 18 March 2023

Abstract

Increasingly, safety-critical systems include artificial intelligence and machine learning components (i.e., learning-enabled components (LECs)). However, when behavior is learned in a training environment that fails to fully capture real-world phenomena, the response of an LEC to untrained phenomena is uncertain and therefore cannot be assured as safe. Automated methods are needed for self-assessment and adaptation to decide when learned behavior can be trusted. This work introduces a model-driven approach to manage self-adaptation of a learning-enabled system (LES) to account for run-time contexts for which the learned behavior of LECs cannot be trusted. The resulting framework enables an LES to monitor and evaluate goal models at run time to determine whether or not LECs can be expected to meet functional objectives and enables system adaptation accordingly. Using this framework enables stakeholders to have more confidence that LECs are used only in contexts comparable to those validated at design time.

Keywords Goal-based modeling · Self-adaptive systems · Artificial intelligence · Machine learning · Models at run time · Cyber physical systems · Behavior oracles · Autonomous vehicles

1 Introduction

The integration of machine learning into autonomous systems is potentially problematic for high-assurance, safety-critical applications [1,2] (e.g., autonomous vehicle features [3,4], medical applications [5,6], smart grid systems [7], etc.), particularly when training coverage is limited and fails to fully represent run-time environments. In addition to meeting functional requirements, safety-critical learning-enabled systems (LESs)[1] must account for potentially a broad range of possible operating scenarios and guarantee that all system responses are safe [8]. However, machine learning components, such as deep neural networks (DNNs), are associated with uncertainties concerning generalizability, robustness, and interpretability [9–11]. A rigorous *software assurance* [12] process is needed to account for these issues of uncertainty. For example, DNNs are used as LECs in numerous safety-critical applications, such as autonomous vehicles, to process onboard camera inputs [3,4]. Failure of these LECs may lead to collisions with pedestrians or nearby objects. Recently, several conferences, workshops, and major US federal funding programs for assured autonomy [13–17] have focused on exploring how the assurance of autonomous systems can be rigorously addressed. This paper proposes a goal-oriented modeling approach to address the assurance

Communicated by Shiva Nejati and Daniel Varro.

Kenneth H. Chan, Jonathon Emil Fleck, Philip K. McKinley and Betty H.C. Cheng have contributed equally to this work.

✉ Kenneth H. Chan
chanken1@msu.edu

✉ Betty H.C. Cheng
chengb@msu.edu

Michael Austin Langford
langfo37@msu.edu

Jonathon Emil Fleck
fleckjo1@msu.edu

Philip K. McKinley
mckinle3@msu.edu

1 Department of Computer Science and Engineering, Michigan State University, 428 S Shaw Ln, East Lansing, MI 48824, USA

[1] This paper refers to any functional software component with behavior that is refined or optimized based on training experience (e.g., an object detector trained by camera images) as a learning-enabled component (LEC). An LES is any system containing one or more LECs (e.g., an autonomous rover).

of LECs and manage the run-time adaptation of a cyber-physical LES.

Although verification of an LEC can include steps to validate learning algorithms *offline*, additional *online* steps are needed to provide confidence that an LES will perform reliably and safely at run time [18,19]. At design time, mathematical proofs can show that convergence criteria of a learning algorithm are satisfied, and empirical testing through cross-validation can help estimate the generalizability of a trained LEC. However, when all conceivable situations cannot be included in training/validation data, methods are needed to dynamically monitor and assess the trustworthiness of LECs to determine whether assurance evidence collected at design time remains valid for previously unseen and/or uncertain run-time conditions. More importantly, an LES must be able to determine when results from an LEC can, or *cannot*, be trusted to correctly respond to current conditions.

This paper presents a model-based framework for Model-Driven Assurance for Learning-enabled Autonomous Systems (MoDALAS). MoDALAS uses goal models to manage the run-time assurance of LESs, providing three key capabilities. First, MoDALAS enables run-time monitoring of LESs with respect to goal models. Second, MoDALAS uses behavior oracles to assess the trustworthiness of LECs based on functional and safety requirements expressed in the goal model. Third, MoDALAS uses goal models to manage the run-time adaptation of the LES to ensure its safe operation in untrusted contexts.

In MoDALAS, online system verification is established by the run-time monitoring of KAOS goal models [20] for functional requirements [21]. Controlled by a self-adaptive feedback loop [22], MoDALAS includes a *behavior oracle* [23] for each LEC. Analogous to a *test oracle* [24], a behavior oracle predicts how an LEC will behave in response to given inputs. In MoDALAS, behavior oracles are used to assess the capability of LECs operating under varying run-time conditions. The resulting self-adaptive LES can then detect when its LECs are operating outside of performance boundaries and adapt accordingly, including possible transitions to fail-safe modes in extreme circumstances. By combining goal models with behavior oracles for an LES, developers can specify requirements concerning the confidence in an LEC and implement alternative strategies to ensure assurance claims are supported. MoDALAS also supports the use of fuzzy logic RELAX goal specifications [25,26] and corresponding analysis techniques to assess system assurance in order to explicitly account for uncertainties in the goal models due to environmental and onboard conditions.

A proof-of-concept prototype of MoDALAS is described for a robotic operating system (ROS)-based [27] autonomous rover LES equipped with a camera-based object detector LEC [28]. DNNs play two roles in this example system: (1) a DNN provides object detection capabilities for the rover and (2) a separate DNN acts as a behavior oracle within MoDALAS to assess the object detector's performance at run time. The object detector has been trained offline by a supervised training dataset, which includes mostly clear-weather examples. However, the autonomous rover must be assured to also function as expected in adverse weather. Without MoDALAS, the object detector would be used regardless of how closely run-time contexts match its trained experience, which could risk accidents under adverse environmental conditions (e.g., haze from a dust plume at a construction site). In contrast, MoDALAS determines when the rover's object detector is operating outside of training coverage and then triggers the rover to adapt accordingly by entering a more cautious operating mode. The remainder of this paper is organized as follows. Section 2 reviews background topics. Section 3 describes how goal models are processed by MoDALAS. Section 4 describes how MoDALAS manages LESs at run time to mitigate the impact of uncertain conditions. Section 5 presents an implementation of MoDALAS for an autonomous rover. Section 6 reviews related work. Finally, Sect. 7 summarizes the paper and briefly discusses future work.

2 Background

This section reviews background topics and enabling technologies that are relevant to the design and operation of MoDALAS, as well as assurance challenges for LECs. MoDALAS is a goal-based (e.g., Goal Structuring Notation (GSN) assurance cases and KAOS requirements models) model-based framework that extends and integrates a number of disparate techniques previously used for different purposes and contexts in order to explicitly address run-time assurance of LESs. For reader convenience, Table 1 provides an "at-a-glance" overview of the main enabling technologies that we have leveraged/extended and integrated to support our objectives in this work.

2.1 Challenges for LESs

Promising results from the use of deep learning [29] to solve traditionally difficult problems such as image classification [30] and object detection [31] have led to an increase in the use of LECs in autonomous systems [32], many of which are safety-critical (e.g., onboard autonomous systems [3,4,33]) where assurance and safety are paramount. For example, DNNs have been implemented for the planning [34], perception [35], and mapping/localization [36] of autonomous vehicles. An advantage for using DNNs in these tasks is to reduce dependence on human feature engineering, as features are learned directly from input data [37]. However, increased

Table 1 Summary of main technologies used in MoDALAS

Technology Name	Description
Assurance Cases	Declarative specifications of assurance claims, organized hierarchically, where the leaf nodes refer to evidence used to establish parent claims
DNNs	Multilayered artificial neural networks with behavior determined by machine learning. For this paper, DNNs are used for two separate purposes
	(1) Functional behavior of an LEC (e.g., an object detector)
	(2) Behavior oracles (e.g., Enlil [23]) to predict how LECs respond to given uncertainty factors
KAOS goal models	A goal modeling technique that supports hierarchical AND/OR decomposition of system goals, where the leaf nodes refer to requirements or environmental expectations [20]
LEC	Control component of an LES with behavior optimized or refined by exposure to training data (e.g., object detector, speech-to-text analyzer, etc.).
MAPE-K Loop	An adaptation manager that supports system reconfiguration at run time in response to environmental changes
RELAX	Requirements specification language used to explicitly specify and account for uncertainty
Utility Functions	Used to monitor system behavior and assess degree of goal model satisfaction

dependence on input data has introduced new challenges concerning data bias [38] and data quality [39]. Furthermore, research has shown that DNNs are sensitive to *surface statistical regularities* [40], causing decisions to be based on superficial, statistically common features in training data rather than *semantically relevant* [41] features to the target task (e.g., deciding an image contains a dog based only on a pattern of grass that is frequently present in the background of training images of dogs). Although DNNs can ease the burden of programming solutions manually with domain expertise, determining the applicability of DNNs to situations not covered by training/validation data poses significant challenges to system assurance.

One major concern is the *robustness* of a DNN (i.e., the ability of a DNN to predict correctly in the face of minor input perturbations) [10,42]. Research has shown that *adversarial examples* [43–45] can be constructed by adding human-imperceptible noise to known inputs in order to deceive DNNs into making incorrect decisions. Such results raise concerns about the capability of DNNs to *locally generalize* [43] (i.e., the expectation that inputs only slightly different from training inputs will lead to similar results). Increasing the robustness of a DNN makes it more locally generalizable and less sensitive to superficial noise.

Automated methods have been developed to augment existing datasets to improve the robustness of DNNs and alleviate the burden of manual data collection. Recent techniques, such as DeepXplore [46], DeepTest [47], DeepGauge [48], DeepRoad [49], DeepConcolic [50], DeepHunter [51], Enki [52], and TensorFuzz [53], are designed to enhance existing data with adverse characteristics in order to uncover vulnerabilities in a DNN. Through data augmentation at design time, these tools can incrementally improve the performance of DNNs to new forms of adversity [54]. However, an empirical study by Ma et al. [55] found certain test selection criteria used by state-of-the-art DNN testing methods to be ineffective at uncovering erroneous DNN behaviors (e.g., selecting test inputs by *neuron coverage* has been found to be sometimes worse than random selection). Moreover, the inclusion of additional training data does not enable the LES to recognize when the performance of an LEC is degraded to the point where it cannot provide useful behavior, or worse, provide unacceptable behavior. For example, a DNN that has been trained with additional synthetic dust cloud data still cannot correctly identify an obstacle hidden behind the dust cloud if it is impossible to see past the occlusion. Therefore, new techniques are required to enable the LES to identify such detrimental situations and adapt to alternate fail-safe modes accordingly, similar to how a human operator will operate with caution in response to the occlusion. Furthermore, DNN testing tools only provide example inputs that lead to specific errors; they do not provide the ability to *predict the expected performance* of a DNN when given new inputs at run time.

Model inference [56] enables the prediction of LEC behaviors at run time. In contrast to software testing, where program inputs are generated to *produce* an intended system behavior, model inference *deduces* resulting system behavior from a given input. Black-box tools have used model inference to improve test generation for software programs by inferring the behavior of traditional software components [57,58]. Aichernig et al. [59] have also described how to construct *behavior models* of cyber-physical systems through deep learning. Langford and Cheng [23] proposed *behavior oracles* to predict and categorize how an LEC will behave in response to environmental conditions different from those covered by the training process. Their system, ENLIL, generates a behavior oracle for an LEC by exposing it to simulated "known unknown" adverse conditions (i.e., conditions that can be *described in appearance* but have an *uncertain impact* on LECs). For example, fog may be considered a *known unknown* condition when the appearance of foggy conditions can be described and simulated but the resulting LEC response for any given example of fog is not known *a priori*. In contrast to *out-of-distribution* methods that assess *confidence* in DNN outputs [60,61], behavior oracles can be constructed to predict the resulting DNN behavior with respect to additional user-specified performance metrics (e.g., predicting a specific level of object detection degradation in adverse conditions). As described in this paper, MoDALAS leverages and extends ENLIL behavior oracles to 1) assess the ability of LECs to satisfy KAOS goals at run time and 2) adapt the LES accordingly.

2.2 Self-adaptive systems

Self-adaptation provides software systems the capability to adjust system behavior in response to the environment [22]. Self-adaptive systems (SASs) commonly operate with a centralized feedback controller (i.e., *adaptation manager*) to observe and adapt managed components of a system [62]. Figure 1 depicts a commonly used approach to manage adaptation, called the *Monitor-Analyze-Plan-Execute over a Knowledge base* (MAPE-K) loop [63,64]. The MAPE-K loop comprises four steps to *monitor* managed elements of the system, *analyze* the current system state to determine a type of adaptation, *plan* what actions need to be taken, and *execute* the operations needed to realize an adaptation. The shared *knowledge base* informs each step in the adaptation process (e.g., system data, adaptation goals, optional tactics, etc.). Thus, the MAPE-K loop enables reconfiguration of an SAS at run time in response to changes in the system or the external environment.

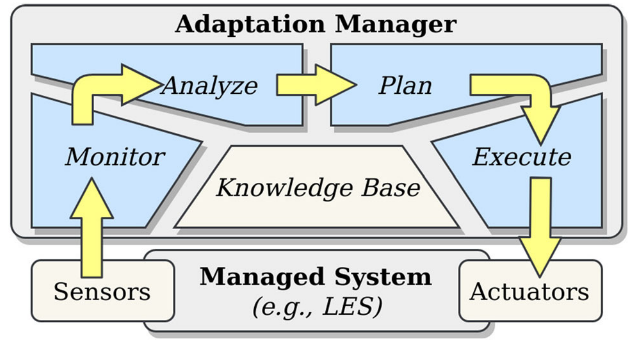


Fig. 1 High-level depiction of a MAPE-K feedback loop to manage adaptations for an SAS [62]

2.3 Utility functions

One approach to monitoring SAS behavior is to use *utility functions* [65,66]. Utility functions map system attributes (i.e., the system state) into real scalar values to express a degree of goal (i.e., requirements) satisfaction [67]. Specifically, a utility function takes the following form.

$$u = f(\mathrm{v}) \quad (1)$$

The *utility value* is a real scalar value ($u \in [0, 1]$), and the *system state* vector ($\mathrm{v} = [s_0, \ldots, s_n]$) reflects specific attributes (s_i) of a system and its environment (e.g., speed or battery level of a rover). Thus, utility functions enable a quantifiable comparison of low-level system states in terms of high-level task-oriented objectives. Furthermore, utility functions help simplify the computational overhead of the MAPE-K *analyze* step when assessing the current state of a system and choosing a method for adaptation [21]. Notably, MoDALAS demonstrates that utility functions can provide a common, unified approach to characterize the behavior of *both* LECs and non-learning system components.

The proposed MoDALAS framework enables run-time verification of an LES by associating utility functions to KAOS goal models, reviewed below, for the LES and its LECs. The associated utility functions are then evaluated by a MAPE-K feedback loop with behavior oracles to assess the capability of LECs at run time. Guided by KAOS goal models that reference different behavior categories for its LECs, an LES can adapt accordingly to mitigate any risks resulting from use of an LEC under conditions for which it has not been adequately trained. Furthermore, results from the run-time evaluation of a KAOS goal model can provide evidence to support assurance claims about the run-time verification of an LES.

2.4 Assurance cases and goal-based modeling

In this subsection, we overview the modeling technologies used in MoDALAS. MoDALAS uses two types of goal models to manage the assurance of LESs at run time. GSN models specify assurance cases for the system, and KAOS goal models specify the functional and performance requirements of the system.

GSN assurance cases

Safety-critical LESs require a rigorous process for describing how functional requirements will be satisfied, including when LECs are presented with uncertain contexts. The purpose of software assurance is to provide a level of confidence to stakeholders that a software system conforms to established requirements [68]. *Assurance cases* provide a means to certify that software operates as intended by describing the validation process and supporting evidence [69]. In an assurance case argument, claims are made about how functional and non-functional requirements are met, and each claim must be supported with verifiable evidence. One way to document an assurance case argument is through the use of GSN [70], which depicts an assurance case as a graphical model. Using GSN, an assurance case is depicted with an overarching assurance claim as its *root goal* (e.g., a rover navigates its environment safely). The root goal is then decomposed into lower-level assurance *strategies*, *assumptions*, *justifications*, *contexts*, and further *subgoals* to explain methods of proof and reasoning for an assurance argument. At the leaf level of a GSN model, *solutions* provide supporting evidence for each respective branch of the assurance argument (e.g., specific results from test cases, proofs, reviews, etc.). Thus, a GSN assurance case graphically decomposes the key elements of an assurance argument, connecting assurance claims to each relevant artifact of supporting evidence, including which validation strategies are required to demonstrate assurance claims are supported.

KAOS goal modeling

Whereas the focus of GSN is on software certification, KAOS goal modeling [20] supports a hierarchical decomposition of high-level functional and performance objectives into leaf-level system requirements (i.e., goal-oriented requirements engineering [71]). KAOS goal models enable a formal goal-oriented analysis of how system requirements are interrelated as well as threats to requirements satisfaction. *Goals* represent atomic objectives of a system at varying levels of abstraction, with subgoals refining and clarifying higher-level goals. Any event threatening the satisfaction of a specific goal is represented as an *obstacle*. Resolutions for obstacles can be specified by attaching additional subgoals with alternative system requirements to the corresponding obstacle. Finally, *agents* (i.e., system components) are assigned responsibility for each system requirement. KAOS goal models enable developers to decompose the expected behavior of a software system, including information about threats to specific system requirements and how system requirements relate to each system component.

RELAX specifications

In this paper, we use the RELAX language [25] to explicitly specify uncertainties affecting an LES. RELAX is a requirements specification language that enables developers to identify, evaluate, and "relax" brittle requirements to address and mitigate uncertainty factors during run time. During requirements engineering, developers may describe system behaviors with strict and highly constrained properties. However, due to the numerous sources of uncertainty potentially impacting an LEC, it may not always be possible to strictly satisfy all requirements. RELAX allows for non-invariant requirements to be temporarily unsatisfied due to uncertain environmental and onboard conditions. RELAX operators add flexibility to the conditions for which a given requirement is considered satisfied, thereby adding the notion of degrees of satisfaction (i.e., "satisficement" [20,72]) in a goal model. For example, RELAX operators such as *AS CLOSE AS POSSIBLE* can be used to reduce the brittleness of a given goal to RELAX elements (i.e., **ENV**, **MON**, and **REL** to specify the relation (REL) of the variables that are used to monitor (MON) an environmental condition with uncertainty (ENV)) [25]. Table 2 enumerates the RELAX operators, with the names of the operators provided in the first column and corresponding descriptions provided in the second. RELAX semantics have been defined in terms of fuzzy logic [25].

RELAX enables developers to create more flexible requirements to ensure robustness against uncertainties. However, modifications to textual requirements do not translate to run-time evaluation. During run time, LESs monitor system values and use utility functions to assess whether system performance and/or configuration satisfy the current goal model. Traditionally, utility functions returned a Boolean value (i.e., 0 or 1) based on goal satisfaction. To address run-time uncertainty, RELAX operators have been mapped to *fuzzy logic* semantics [75,76]. Fuzzy logic is a branch of logic that enables developers to specify a partially satisfied goal. Using fuzzy logic, a utility function can return normalized real values ranging from 0 (i.e., not satisfied) to 1 (i.e., satisfied). A goal that returns a partially satisfied utility function is known as *satisficed* [20]. Since fuzzy logic allows utility functions to return real numbers, goal refinement (i.e., *AND* and *OR* goal decompositions) for parent goal evaluation must be redefined. In the parent goal of *RELAX*-ed goals, the goal's utility function is evaluated by applying mathematical operations (e.g., min and max) on subgoals' satisficement values. While several popular approaches to fuzzy logic evaluation exist, this work uses Zadeh fuzzy operators [77], a common convention for resolving fuzzy logic (e.g., conjunctions, disjunctions, and implications). Using Zadeh fuzzy operators, when the subgoals are related by an *OR* relationship, the maximum value of all subgoals' utility functions determines the evaluation of the parent goal. In contrast, when the subgoals are part of an *AND* relationship, their minimum value determines the parent goal instead. A parent goal may be converted to Boolean satisfaction if the evaluation of the value of the *RELAX*-ed subgoals exceeds a specified threshold (e.g., 0.5). To illustrate an example of a RELAX specification, we describe a component of an autonomous vehicle that detects obstacles. A traditional requirement may be described as follows:

Table 2 Example of RELAX operators and uncertainty factors [73,74]

Operator	Description
Modal operators	
SHALL	A requirement must hold
MAY...OR	A requirement specifies one or more alternatives
Temporal operators	
EVENTUALLY	The requirement that must hold eventually
UNTIL	A requirement must hold until a future position
BEFORE/AFTER	A requirement must hold before or after a particular event
AS EARLY AS POSSIBLE	A requirement specifies something that should hold as soon as possible
AS LATE AS POSSIBLE	A requirement specifies something that should be delayed as long as possible
AS CLOSE AS POSSIBLE TO [frequency t]	A requirement specifies something that happens repeatedly, though the frequency may be relaxed
Ordinal operators	
AS FEW/MANY AS POSSIBLE	A requirement specifies a countable quantity, though the exact count may be relaxed
AS CLOSE AS POSSIBLE TO [quantity q]	A requirement specifies a countable quantity, though the exact count may be relaxed

Uncertainty factors	Description
ENV	Defines a set of properties that define the system's environment
MON	Defines the set of properties that can be monitored by the system
REL	Defines the relationship between **ENV** and **MON** properties
DEP	Defines the dependencies between the (relaxed and invariant) requirements

S1: The system *SHALL* detect obstacles within 10 meters.

This requirement represents an ideal situation. However, instead of a rigid requirement, a developer may wish to relax the requirement to account for uncertainty factors (e.g., speed variance of two vehicles, the sensitivity of the sensors, etc.). For example, **S1** may be RELAX-ed to the expression **S1**′ if the vehicle is traveling below 10 ms per second, since there is more time for the system to react to detected obstacles.

S1′: The system *SHALL* detect obstacles *AS CLOSE AS POSSIBLE* to 10 meters.
ENV: location of obstacle
MON: obstacle detection system
REL: system detects obstacle

Using **S1**′, the system can still handle the requirement of "detect obstacle within 10 ms," and also support a more flexible requirement should the system detect an obstacle within 8 ms. Specifically, developers can replace rigid Boolean utility functions (i.e., the system detected obstacle within 10 ms or not) with fuzzy logic utility functions (e.g., degree of *satisficement* from 0 to 1) using RELAX. As such, the system can adapt and temporarily trade-off non-critical requirements to maintain the satisfaction of more critical requirements.

3 Modeling in MoDALAS

This section describes how modeling and specification technologies (e.g., GSN, KAOS, RELAX) are applied at design time in MoDALAS. We describe how we have integrated KAOS goal modeling and corresponding analysis into the GSN assurance approach. In addition, our approach to KAOS goal modeling enables developers to identify uncertainty in the form of both obstacles and RELAX specifications. We also include utility functions in the leaf level nodes of the KAOS model as a means to assess whether the individual goals and the full goal model are satisfied at run time.

3.1 Assurance cases in MoDALAS

MoDALAS accepts as inputs assurance cases and goal models that have been constructed and validated through methods such as *model checking* at design time [78]. In this work, assurance cases are modeled using GSN, though alternative methods may also be used to describe an assurance case [12]. A simple example GSN model is shown in Fig. 2, which claims a rover will navigate its environment safely (claim *C1*). Strategies are implemented to support claim *C1* through offline validation (strategy *S1*) and run-time analysis (strategy *S2*). At design time, software testing, model checking, and formal analysis are conducted offline to support assurance claim *C2*, with results provided as evidence in solutions *Sn1*-*Sn3*. At run time, evaluation of a KAOS

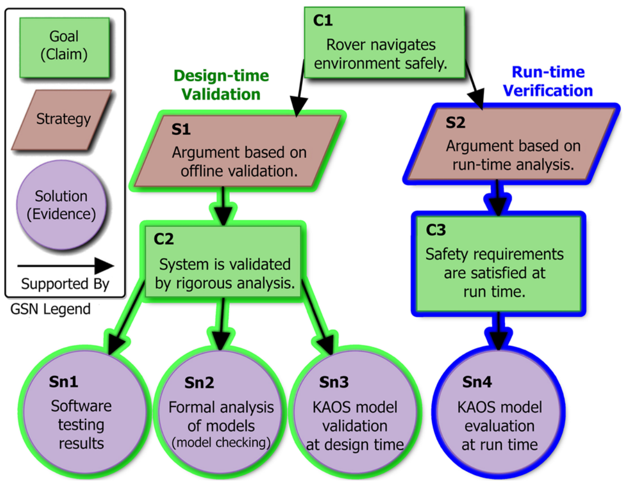


Fig. 2 Example GSN assurance case for *design-time* and *run-time* validation of a rover. At design time, validation is supported by formal proofs, test results, and simulation (highlighted in green). At run time, verification is supported by the evaluation of a KAOS goal model (highlighted in blue)

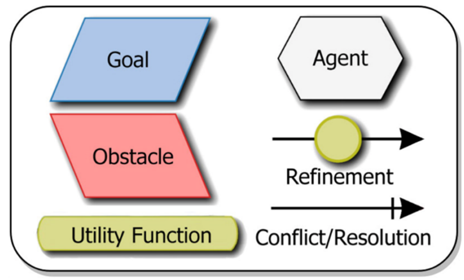


Fig. 3 Legend key for interpreting the KAOS goal model notation

goal model for the rover provides evidence (solution *Sn4*) to demonstrate system requirements remain satisfied under changing run-time conditions (claim *C3*). As such, a GSN model provides context for our work, where evidence generated for assurance solution *Sn4* is provided by the evaluation of KAOS goal models at run time.

3.2 Goal modeling in MoDALAS

This section describes how KAOS goal models are automatically processed by MoDALAS to hierarchically decompose high-level goals into leaf-level system requirements for analysis. Figure 3 shows a legend for KAOS goal modeling elements, while Fig. 4 shows an example KAOS goal model for a rover that must navigate its environment. In this example, a rover is expected to sense objects in its environment and plan its trajectory around objects according to object type (e.g., when pedestrians are present (*G10*) or not (*G9*)). The rover implements a DNN-based *object detector* that can *locate* zero or more objects within a camera image and *classify* the type of each object [28]. The trustworthiness of the object detector depends on how similar the run-time environment is to its training experience. The rover also ensures there is sufficient braking power (*G20*) using sensor values from the tire pressure monitoring system and friction sensor. In Fig. 4, *utility functions* (shown in yellow) are attached to the bottom of each goal. Utility functions enable the KAOS goal model to be evaluated at run time to determine goal satisfaction.

In KAOS notation, any event that can threaten the satisfaction of a goal is represented as a KAOS *obstacle*. In Fig. 4, obstacles *O1* and *O2* represent events in which the object detector is operating in a state not explored during design time. Obstacle *O1* represents events where the object detector's performance is *degraded* (i.e., statistical performance is less than ideal), and obstacle *O2* represents events where the object detector is *compromised* (i.e., statistical performance is unacceptable). When the object detector is compromised, goal *G16* is given as an obstacle resolution, where the rover is expected to perform a fail-safe procedure (e.g., halt movement). When the object detector is only degraded, goal *G17* is given as a resolution, where the rover is expected to slow down and increase its minimum buffer distance from objects encountered in the environment. Additional KAOS obstacles and resolutions can be included, depending on the LEC, targeted behavior categories, and LES requirements.

3.3 Relaxing goals in MoDALAS

To address environmental uncertainties, developers may use RELAX to temporarily allow specific requirements to be relaxed within acceptable ranges. During the design step, the developers identify non-invariant requirements that may be relaxed. Next, developers specify specific requirements in terms of a KAOS goal model, including various RELAX operators for goals that face uncertainty factors. During this step, developers must also define utility value thresholds for goals that convert fuzzy logic utility function values to Boolean utility function values.

Figure 5 shows a modified version of the KAOS goal model from Fig. 4, where RELAX operators are used in goals *G20*, *G21*, and *G22*. In the new goal model, we modified *G20* with the RELAX language to allow partial goal satisfaction. The fuzzy logic utility functions of the RELAX-ed goal models are evaluated to return a real number ranging from 0 to 1 to represent the degree of satisficement for the specific goal. Specifically, *G20* is considered satisfied if the threshold value of both the tire pressure sensor monitor and the friction monitor are satisfied to the degree of 0.5.

To demonstrate the use of RELAX with an autonomous rover, Fig. 6 shows an example of a RELAX-ed goal branch. Consider goal *G20*, where the rover ensures that there is sufficient braking power for the rover to stop should it detect a

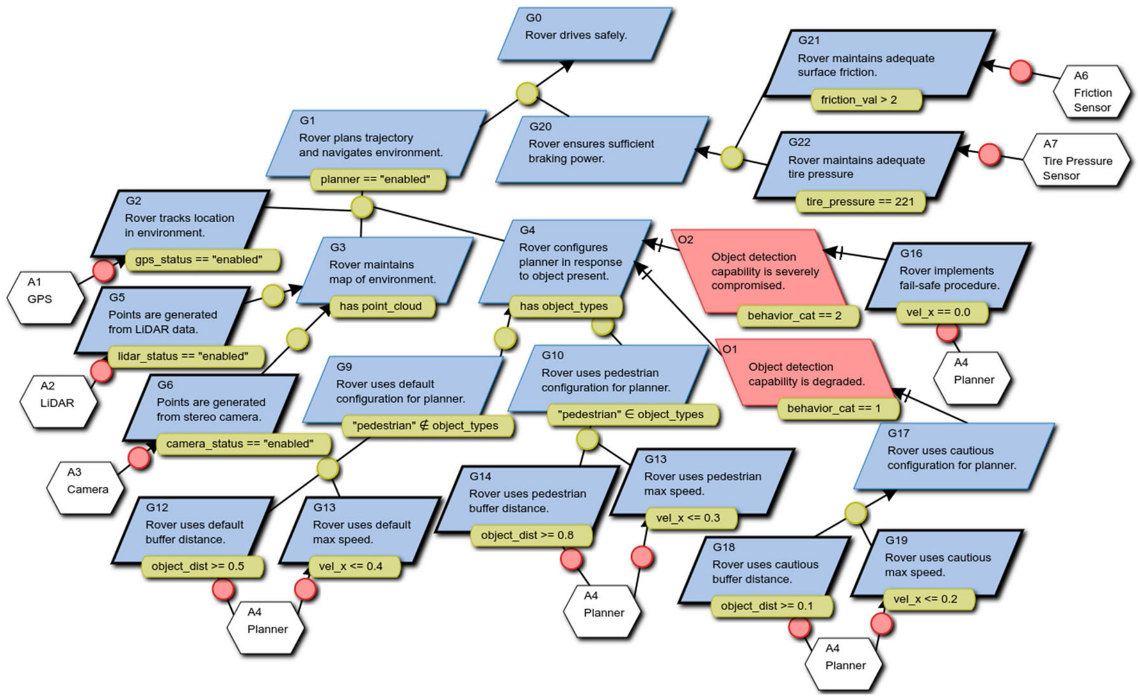


Fig. 4 Example KAOS goal model. *Goals* (blue parallelograms) represent system objectives. The top-level goal (*G1*) is refined by subgoals down to leaf-level system requirements. *Agents* (white hexagons) represent entities responsible for accomplishing requirements. *Obstacles* (red parallelograms) represent threats to the satisfaction of a goal (e.g., *O1* and *O2*)

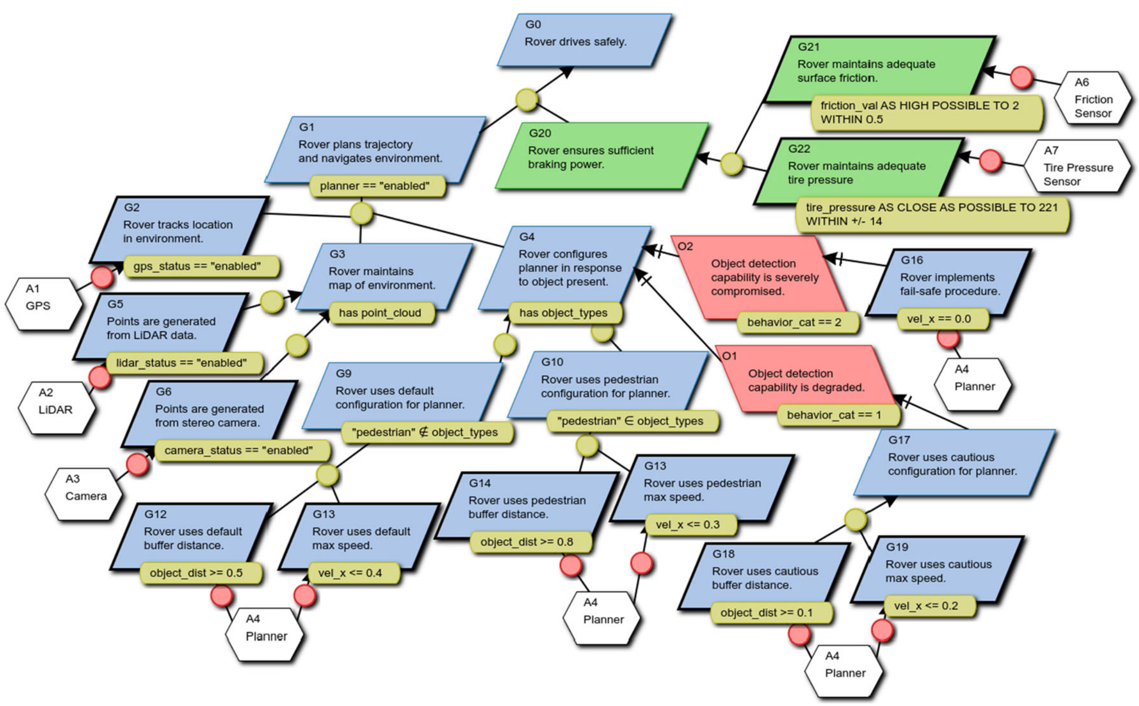


Fig. 5 Example a RELAX-ed KAOS goal model. Goals *G20*, *G21*, and *G22* use fuzzy logic to denote degree of goal satisfaction

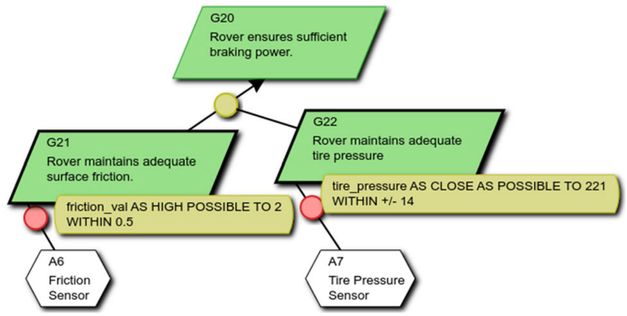


Fig. 6 Example of the KAOS goal model for *G20*, *G21*, and *G22* demonstrating a RELAX-ed goal hierarchy

potential collision. Previously, *G20* returns a Boolean value to parent goals indicating whether there is enough braking power or not. *G21* and *G22* returned Boolean values depending on whether or not the specific sensor values are satisfied. In order to add flexibility and account for environmental uncertainties, *G21* and *G22* are RELAX-ed to explicitly capture uncertainty (e.g., if the visibility is poor and the rover is traveling under 10 m/s). To check if *G20* is satisfied, the system ensures that i) the friction sensor detects a friction rate of 2 Newtons, with an acceptable range of −0.5 Newtons and ii) the tire pressure is within 221 kPa, with an acceptable range of ±14 kPa. Fuzzy logic is used to express the degree of *satisficement* in the RELAX utility functions. For example, if the system detects that the value of a RELAX-ed goal is satisfied (e.g., tire pressure is at 221 kPa), then the corresponding utility function returns 1. The value returned reduces to 0 as the value reaches the lower bound of the acceptable range.

Figure 7 shows an example of the range of values returned for *G21*. The utility function used to evaluate *G21* is shown in Fig. 8. *G21* returns 1 if the value of the friction sensor reads 2 Newtons. The returned value linearly decreases as the sensor value reduces to the lower bound of the acceptable range. In contrast, Fig. 9 shows an example of the range of values returned in *G22*. Unlike *G21* where the goal has an acceptable range below a set value, *G22* allows for satisficement in both directions (i.e., a triangular fuzzy logic function is used). The utility function used to derive the return value for *G22* is shown in Fig. 10. It returns 1 when the tire pressure monitor reads 221 kPa. Since the acceptable range of the utility function is defined as 221 ± 14 kPa, the value returned by the utility function decreases linearly to 0 as it approaches 207 kPa or 235 kPa, forming a triangular relationship.

The value of the parent AND goal, *G20*, is based on the evaluation of the utility functions of its two children subgoals, *G21* and *G22*. The satisfaction of *G20* is determined by a threshold value defined by a domain expert. Figure 6 shows the logical relationship between *G20*, *G21*, and *G22*, where the subgoals form an *AND* relationship. Expression 2 specifies the utility function used to evaluate goal *G20*.

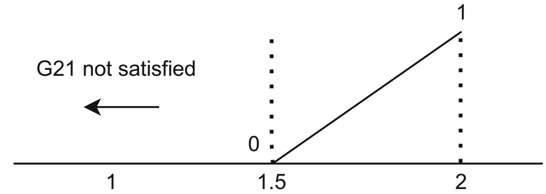


Fig. 7 Range of values returned by the friction sensor (*G21*) using the RELAX language with fuzzy logic

```
Function getFrictionSensorValue(measured, desired=2,
  bounds=0.5):
    if (measured ≥ bounds) then
        return 1.0
    else if (measured ≤ desired − bounds) then
        return 0.0
    else
        return (desired − bounds − measured) / bounds
    end
```

Fig. 8 Utility function to calculate goal satisficement for *G21*

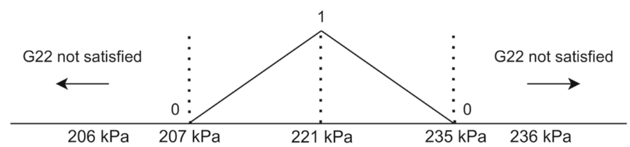


Fig. 9 Range of values returned by the tire pressure monitor system (*G22*) using the RELAX language with fuzzy logic

```
Function getTirePressureSensorValue(measured,
  desired=221, bounds=14):
    if (measured ≤ desired − bounds) ∨
       (measured ≥ desired + bounds) then
        return 0.0
    else if (measured ≤ desired) then
        return (desired − bounds − measured) / bounds
    else
        return (desired + bounds − measured) / bounds
    end
```

Fig. 10 Utility function to calculate goal satisficement for *G22*

$$G20_{util} = \begin{cases} 1.0 & \text{if } \min(G21_{util}, G22_{util}) \\ & \quad > \text{threshold} \\ 0.0 & \text{otherwise} \end{cases} \tag{2}$$

While we demonstrate the use of RELAX to explicitly specify uncertainty in this paper, MoDALAS can also accommodate other types of requirement specification languages and corresponding utility functions used to address uncertainty, such as FLAGS [79], probabilistic utility functions, etc.

4 Managing systems with MoDALAS

This section describes how the MoDALAS framework supports goal-based self-adaptation of an LES. Figure 11 shows a data flow diagram (DFD) of the framework. Circles repre-

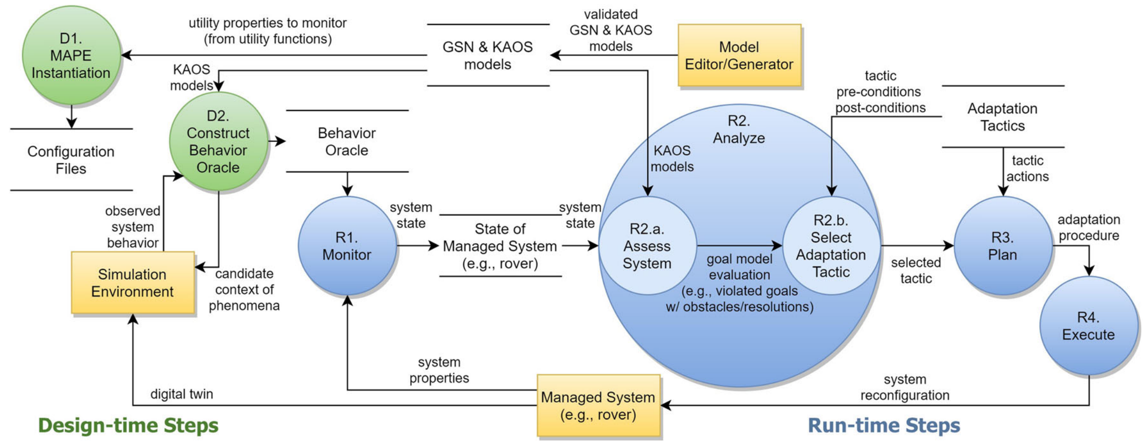


Fig. 11 High-level data flow diagram of MoDALAS. Processes are shown as circles, external entities are shown as boxes, and persistent data stores are shown within parallel lines. Directed lines between entities show the flow of data

sent computational steps, boxes represent external entities, directed arrows show the flow of data, and persistent data stores are shown within parallel lines. Design-time steps (green) include the construction of an assurance case, a goal-oriented requirements model of the LES, and a behavior oracle for each LEC. Run-time steps (blue) implement a MAPE-K feedback loop driven by the models constructed at design time.

Although MoDALAS is platform-independent, to aid the reader, the following descriptions include an example of an autonomous rover with a learning-enabled object detector. Specific implementation details on how MoDALAS is applied to the autonomous rover are provided in Sect. 5. Each step for the DFD in Fig. 11 is described in detail as follows.

4.1 MAPE instantiation

In *Step D1*, an adaptation manager (implemented as a MAPE-K loop) is instantiated to manage adaptations of the LES. To determine the system state and evaluate KAOS goal models at run time, the adaptation manager must be configured to monitor the same system attributes referenced by KAOS goal models. KAOS goal models are parsed, and utility functions are extracted from each KAOS element. MoDALAS requires that KAOS goal models have been converted into a machine parsable file format (e.g., XML) that includes attributes for each KAOS element and its associated utility function. A set of *utility parameters* is then compiled by identifying each unique variable referenced by a utility function. Since utility parameters may refer to abstract concepts, a manual mapping must be made by the user to link each utility parameter to a platform-specific property of the LES. For example, for the utility function `object_dist >= 0.8` in Fig. 4, goal *G14*, the utility parameter `object_dist` refers to the buffer distance between the rover and any object in the environment. It is the responsibility of the adaptation manager to link this abstract parameter to a real, platform-specific property of the rover. Configuration files are generated by *Step D1* to initialize the monitor processes of the MAPE-K loop with references to the platform-specific properties to observe.

4.2 Constructing behavior oracles

To monitor and assess the trustworthiness of LECs at run time, MoDALAS leverages *behavior oracles* generated by Enlil [23] for each individual LEC in *Step D2*. In contrast to traditional monitoring techniques used in the MAPE-K loop (e.g., physical sensors, data-based monitors [80], etc.), behavior oracles are uniquely implemented as DNNs in MoDALAS to *infer* behavior of each LEC when exposed to new forms of environmental uncertainty under simulation. For example, when a rover implements a learning-enabled object detector that has been trained only in clear weather, Enlil can be used to simulate adverse weather conditions and model the capability of the object detector under a variety of adverse conditions. The resulting behavior oracle can then predict different *behavior categories* for the object detector when presented with sensor data under various weather conditions. These categories are application-specific and must be defined according to the user for the given task and LECs involved. Furthermore, we codify the behavior oracle evaluations in the form of utility functions to enable real-time assessment of the type of LEC behavior to expect under varying run-time conditions.

The KAOS goal model in Fig. 4 reflects that three different behavior categories have been specified for the behavior oracle of a rover's object detector. Two of these (`behavior_cat == 1` and `behavior_cat == 2`) are attached to obstacles *O1* and *O2*, respectively. The third

Table 3 Behavior categories for an object detector

Category	Classification		Definition
0	*"Little impact"*	● (green)	$0\% \leq \delta_{recall} < 5\%$
1	*"Degraded"*	● (yellow)	$5\% \leq \delta_{recall} < 10\%$
2	*"Compromised"*	● (red)	$10\% \leq \delta_{recall}$

(`behavior_cat == 0`) is the default and not explicitly shown in Fig. 4. (The number of behavior categories depends on the granularity and spectrum of available behaviors and also the number of alternative resolutions required to satisfy system requirements.) Categories are determined by assessing the object detector's performance under a variety of adverse environmental contexts in simulation. In this example, the object detector's *recall*[2] is measured when a newly-introduced adverse condition is present versus when it is not. The change in recall (δ_{recall}) is then used to measure the effect on object detector's performance. The value of δ_{recall} is computed statistically by measuring the object detector's recall for a set of validation images with and without exposure to the given environmental phenomena. Table 3 provides a description of each behavior category reflected in Fig. 4. When $\delta_{recall} < 5\%$, the given context is considered to have *"little impact"* on object detection (*Category 0*). When $5\% <= \delta_{recall} < 10\%$, the object detector is considered *"degraded"* (*Category 1*). Finally, when $\delta_{recall} > 10\%$, the object detector is considered *"compromised"* (*Category 2*).

ENLIL automatically assesses an LEC by generating unique contexts of simulated environmental phenomena (via *evolutionary computation* [82]) to uncover examples that lead to each targeted behavior category. Figure 12 shows a scatter plot generated by ENLIL when simulating dust clouds. Each point represents a different dust cloud context with the resulting recall for the object detector LEC. Colors correspond to the observed behavior category for each respective point (i.e., categories 0, 1, and 2 are *green*, *yellow*, and *red*, respectively). Data collected during this assessment phase are used by ENLIL to train a behavior oracle that can map LEC inputs to expected behavior categories (i.e., *model inference*).

Behavior oracles created in *Step D2* are used at run time to predict the resulting behavior category of an LEC for any given run-time context. For the example object detector, inputs to the behavior oracle are the same camera inputs given to the object detector. Output from the behavior oracle includes a description of the *perceived context* of the environmental condition and the *inferred behavior category* for the object detector. Figure 13 shows three behavior categories to represent different degrees of impact dust clouds can have

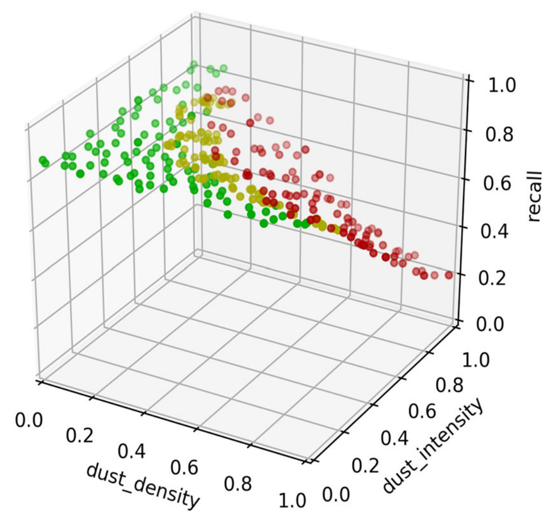


Fig. 12 Scatter plot of object detector recall when exposed to simulated dust clouds from ENLIL. Each point represents a different dust cloud context with the corresponding *density* and *intensity*. Green, yellow, and red points correspond to behavior categories 0, 1, and 2, respectively

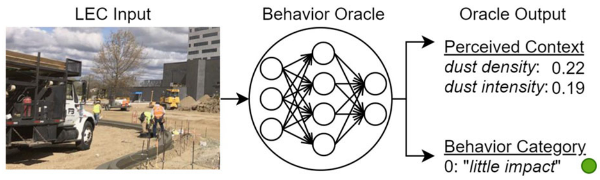
on an object detector. Effectively, this information is used to assess the *trustworthiness* of the object detector.

4.3 Self-adaptation at run time

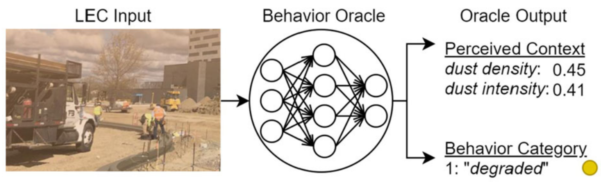
A MAPE-K loop adaptation manager is executed at run time to monitor and reconfigure the managed LES with respect to the models constructed at design time. Responsibilities include assessing the current state of the LES, predicting the capability of LECs via behavior oracles, determining when system requirements are not satisfied by referencing KAOS goal models, and planning adaptations to ensure mitigating actions are taken to maintain requirements satisfaction.

Step R1. Monitor: In order to inform self-adaptations, *monitor* processes observe and record relevant attributes of the managed LES, which includes executing the behavior oracles constructed in *Step D2*. In KAOS notation, *agents* indicate which system components are responsible for each system requirement (e.g., *A1-A4* in Fig. 4). Specific attributes of a system component are monitored when referenced by *utility functions* in the models constructed at design time (see *Step D1*). Monitor processes are responsible for observing functional system components (e.g., controllers, mechanical parts, physical sensors, etc.) as well as behavior oracles for LECs. For example, when using a behavior oracle for a camera-based object detector, the behavior oracle is executed for each new camera input to predict the impact of run-time conditions on the object detector's performance. Through the use of utility functions, MoDALAS enables LECs to be monitored in a similar manner to traditional system components, using behavior oracles to determine whether or not LECs can be trusted in the current system state.

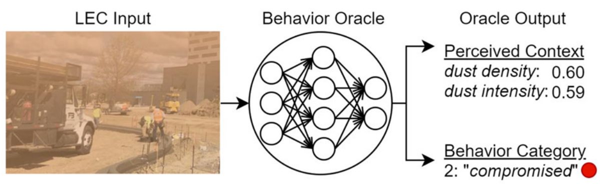
[2] *Recall* is the ratio of correctly detected objects to all detectable objects (i.e., the ratio of *true positives* to both *true positives* and *false negatives*) [81].



(a) "light" dust cloud resulting in a *category 0* classification



(b) "medium" dust cloud resulting in a *category 1* classification



(c) "heavy" dust cloud resulting in a *category 2* classification

Fig. 13 Example behavior oracle input/output for an image-based object detector LEC. Input is identical to the input given to the LEC. Output is a "*perceived context*" to describe the environmental condition and a "*behavior category*" to describe the expected LEC behavior. Examples of behavior categories 0, 1, and 2 are shown in (a), (b), and (c), respectively

```
planner_status == ``enabled''
  AND gps_status == ``enabled''
  AND has point_cloud
      AND lidar_status == ``enabled''
         OR camera_status == ``enabled''

AND has object_types
    AND
        ``pedestrian'' ∈ object_types
          AND object_dist ≥ 0.5
            AND vel_x ≤ 0.4

       OR ``pedestrian'' ∉ object_types
          AND object_dist ≥ 0.8
            AND vel_x ≤ 0.3

    AND IF behavior_cat == 1
       THEN object_dist ≥ 1.0
          AND vel_x ≤ 0.2

    AND IF behavior_cat == 2
       THEN vel_x == 0
```

Fig. 14 Logical expression parsed from KAOS model in Fig. 4

```
tactic "Reconfigure to Cautious Mode"
  pre-condition: (KAOS O1)
     behavior_cat == 1

  actions:
     set object_dist ← 1.0
       set vel_x ← 0.2

  post-condition: (KAOS G17 ∧ G18 ∧ G19)
     object_dist ≥ 1.0
       AND vel_x ≤ 0.2
```

Fig. 15 Example tactic for reconfiguring a rover to a "cautious mode." *Pre-conditions* and *post-conditions* refer to KAOS elements and utility functions (see Fig. 4). *Actions* are abstract operations to achieve the post-condition

Step R2. Analyze: KAOS goal models of the LES are evaluated in *Step R2.a* to determine if adaptation is needed to resolve violated system requirements. Utility functions from the KAOS goal model are extracted, and a logical expression is formed via top-down graph traversal of the KAOS goal model. For example, Fig. 14 shows the logical expression parsed from the KAOS goal model in Fig. 4. Variables in this expression are substituted with corresponding values recorded by *Step R1*, and the entire expression is evaluated to determine satisfaction of the KAOS goal model. If the logical expression is satisfied, then no adaptation is needed. However, in the event that the resulting evaluation is unsatisfied, then the type of adaptation is determined based on the set of violated utility functions.

Methods for adaptation are implemented as *adaptation tactics* [83], which are stored in a repository accessible by the MAPE-K loop (example tactic in Fig. 15). Each tactic comprises a *pre-condition*, *post-condition*, and set of *actions* to perform on the managed LES. Pre-conditions and post-conditions for tactics reference the satisfaction of KAOS *goals/obstacles*, where pre-conditions are defined by the utility functions for KAOS obstacles and post-conditions are defined by the utility functions associated with the resolution goals for KAOS obstacles. *Step R2.b* retrieves a tactic from the repository with pre-conditions that most closely match (e.g., via logical implication) the current evaluation of the KAOS goal model. For example, in the event that obstacle *O1* is satisfied and goal *G17* is not satisfied, then the tactic in Fig. 15 with a pre-condition matching the utility function for *O1* is selected. The post-condition in Fig. 15 includes the utility functions for *G17* and its subgoals (*G18* and *G19*). The actions associated with the tactic are platform-independent operations required to satisfy the post-condition. When multiple tactics fit the given pre-conditions, the tactic with *higher priority* is chosen, where priorities can be manually assigned and/or adjusted based on the success of subsequent goal model evaluations in future iterations of the MAPE-K loop.

Step R3. Plan: After a platform-independent adaptation tactic has been selected in *Step R2*, a platform-specific procedure is generated for implementing the *actions* associated with the selected tactic. For example, a platform-independent action to turn the autonomous platform 15° will be translated into the corresponding operations for a wheeled rover versus a legged-robot. Additionally, actions may be modified to consider the dynamic state of the system during execution of a tactic (e.g., actions may change or be preempted to account for emergent mechanical issues in a rover) [84].

Step R4. Execute: After an adaptation procedure has been planned, *Step R4* is responsible for interfacing with and reconfiguring the managed LES. Depending on the nature of the adaptation and the current system state, different methods of adaptation may be considered to ensure the managed LES functions correctly while safely transitioning into its new configuration (e.g., *one-point*, *guided*, or *overlap* adaptations) [85]. Since adaptations may not be safe to perform in all states of the LES, *Step R4* is responsible for determining *quiescent states* where the LES can be safely reconfigured (e.g., prevent halting a rover during a high-speed turn) [86].

5 Proof-of-concept demonstration

To illustrate the operation of MoDALAS, we consider a scenario where an autonomous rover is used within a construction site.[3] Compared to autonomous automobiles operated on public roads, autonomous construction vehicles operate within relatively tight behavioral constraints and physical areas, leading to rapid growth in this market segment [87]. In addition to large earth-moving vehicles, smaller rovers are used to carry tools and materials for construction workers, periodically record the progress of construction, and provide surveillance of the site outside of normal operating hours [88]. For such rovers, detecting and avoiding objects in the environment, including pedestrians and other vehicles, are safety-critical requirements [89]. Increasingly, machine learning techniques have been used to provide object detection in such applications [90]. However, ensuring requirements satisfaction of learning-enabled autonomous rovers is a challenging task, as transient environmental conditions (i.e., rainfall or dust clouds) can impede object detection and potentially lead to serious accidents and even fatalities. To demonstrate the operation of MoDALAS in the construction site application domain, we have implemented a prototype and integrated it into the software for an autonomous robot in our laboratory.

[3] Due to COVID-19 restrictions, we were unable to deploy our approach for a full-scale physical experiment at the remote construction site.

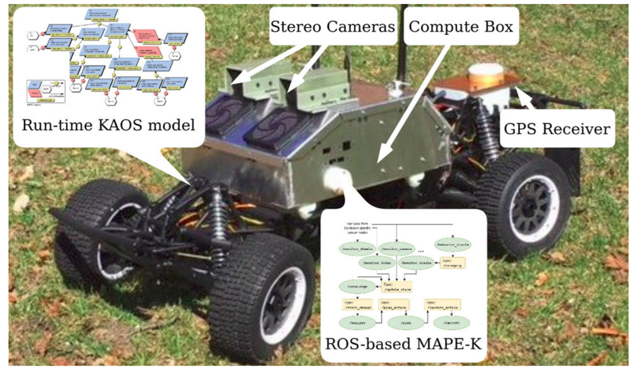


Fig. 16 A 1:5-scale (1.1 m × 0.6 m) autonomous vehicle for demonstration. A KAOS goal model governs run-time behavior and adaptation. A MAPE-K loop, integrated in the ROS infrastructure, identifies and acts on required adaptations

5.1 Rover platform

Our rover, shown in Fig. 16, is a 1:5-scale vehicle based on a design published by Goldfain et al. [91]. The rover is powered by an electric motor and includes wheel speed sensors, an Inertial Measurement Unit (IMU), GPS, and an optional lidar unit. Of particular relevance to this study are stereo cameras mounted atop the rover. A compute box contains an Intel i7 quad-core processor, 32-gigabyte RAM, 2-terabyte SSD, and an Nvidia GPU for image processing. In addition, a Gazebo-based [92] simulation of the vehicle is used for offline testing.

ROS-Based Platform The rover's software infrastructure is based on ROS [27], a popular middleware platform for robotics. A ROS implementation comprises a set of processes, called ROS *nodes*, that communicate with other ROS processes using a publish-subscribe mechanism called ROS *topics*. Multiple ROS nodes can publish messages on a ROS topic, and multiple ROS nodes can subscribe to the same ROS topic. Commonly, and in our case, ROS is implemented atop the Linux operating system with ROS nodes realized as Linux processes. For a non-trivial robot such as our rover, this design produces an intricate software infrastructure that can be visualized with a ROS *graph*. The full ROS graph for our rover software comprises over 30 nodes that implement tasks such as processing of sensor data, localization, path planning, and generating the corresponding commands to control the vehicle. Over 200 ROS topics are used to convey raw and preprocessed sensor data, exchange of information among controller nodes, and deliver commands to actuators for throttle control, steering, and braking.

ROS-Based Adaptation Manager Fig. 17 shows an (elided) ROS graph of the MAPE-K loop implemented for the rover. The `/knowledge` ROS node is a process that manages access to the data stores depicted in Fig. 11. Data stores for goal models and adaptation tactics are populated at startup time and remain static during operation. However,

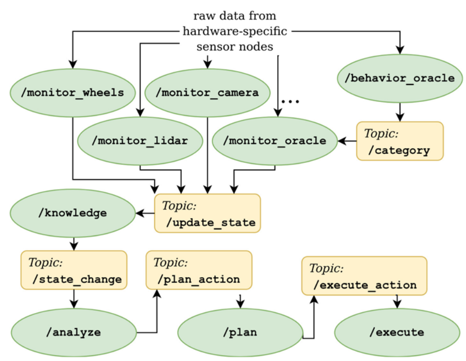


Fig. 17 Elided ROS graph for MAPE-K loop in rover software. ROS nodes shown as green ellipses and ROS topics as yellow boxes. Arrows indicate data flow

the *managed system state* data store is highly dynamic, comprising sensor readings and other state information that are updated continually. The MAPE-K *monitor* step (*Step R1* in Fig. 11) is implemented as a collection of ROS nodes (e.g., `/monitor_lidar`, `/monitor_wheels`, `/monitor_camera`) that receive raw sensor data collected by hardware-specific ROS nodes. These nodes preprocess data streams and publish results to the `/update_state` topic in order to modify the managed system state. Examples include direct measurements (e.g., wheel speed), derivative measurements (e.g., rate of battery drain), and operational status of hardware components (e.g., delays in GPS localization reporting). The remaining three MAPE-K steps (*Steps R2-R4*) are implemented as singleton ROS nodes, respectively, `/analyze`, `/plan`, and `/execute`.

KAOS goal model evaluation by the `/analyze` node is triggered by state changes published on the `/state_change` ROS topic. If the KAOS goal model is not satisfied and an adaptation is necessary, then the `/analyze` node determines the type of adaptation needed and relays the adaptation type to the `/plan` node via the `/plan_action` topic. The `/plan` node retrieves actions for the corresponding tactic from the knowledge base and forwards an adaptation procedure to the `/execute` node. The `/execute` node directly interfaces with and reconfigures components of the target platform.

5.2 Camera data and the behavior oracle

In our proof-of-concept demonstration, we use images from the mounted cameras atop the rover for *object detection* [28,31] and *triangulation* from stereo vision [93]. A three-dimensional *point cloud* [94] is generated by fusing stereo camera triangulations and lidar sensor readings. As shown in Fig. 17, both the `/monitor_camera` and `/behavior_oracle` nodes receive raw camera data published from onboard cameras. The `/monitor_camera` node processes camera data and delivers relevant information (e.g., frame rate, etc.) to the `/knowledge` node. For example, lack of input or a slow frame rate might indicate a problem with one or both cameras, thus necessitating a run-time adaptation.

The `/behavior_oracle` node processes camera images *online* with the behavior oracle DNN that was trained *offline* by Enlil for model inference. Specifically, the `/behavior_oracle` node infers the behavior of the onboard object detector LEC by evaluating input images given to the object detector. The behavior category produced by the `/behavior_oracle` node is published on the `/category` ROS topic, which is monitored by the `/monitor_oracle` node. At run time, if the `/monitor_oracle` node reports any change in the behavior category, then the `/analyze` node will execute to address the situation, as follows.

5.3 Run-time adaptation

When adverse run-time conditions are detected, MoDALAS prevents the use of an LEC in environments for which they have not been adequately trained by switching to fail-safe modes of operation. In the absence of such a run-time self-assessing framework, an LEC might provide inappropriate behavior when encountering unsafe operating conditions, where the LES is not aware that the LEC is operating beyond its scope of capabilities. MoDALAS provides a means to identify these situations and adapt the LES to execute validated fail safes when potential failure cases are detected.

We consider a scenario in which the behavior oracle triggers run-time adaptations to the rover. At design time, the behavior oracle was created to account for three types of adverse environmental conditions that can impact object detection: *rainfall*, *dust clouds*, and *lens flares* (where a bright light source obscures part of the image). Additional environmental phenomena can be included by introducing them into the simulation environment used by Enlil. Figure 18 shows examples of each simulated phenomenon, with different levels of intensity, and the resulting object detector behavior category inferred by the behavior oracle. Referencing the behavior categories in Table 3, examples in columns (i), (ii), and (iii) are expected result in *Categories 0*, *1*, and *2*, respectively, (i.e., "*little impact*," "*degraded*," and "*compromised*.")

<u>*Scenario 1. Dust Clouds*</u> To demonstrate MoDALAS in practice, we explore a scenario where the autonomous rover navigates a construction site to periodically record progress on the project at different locations. The rover begins with `behavior_category = 0`. As the rover approaches a construction worker, a dust cloud is produced by a dump truck leaving the construction area. When the `/behavior_oracle` node receives images from the rover's cameras, the dust-obscured images are evaluated by the behavior oracle DNN, which infers that the object detector will be *degraded* by the current environment. Thus, the `/behavior_oracle` node publishes `behavior_category = 1` on the `/category` topic.

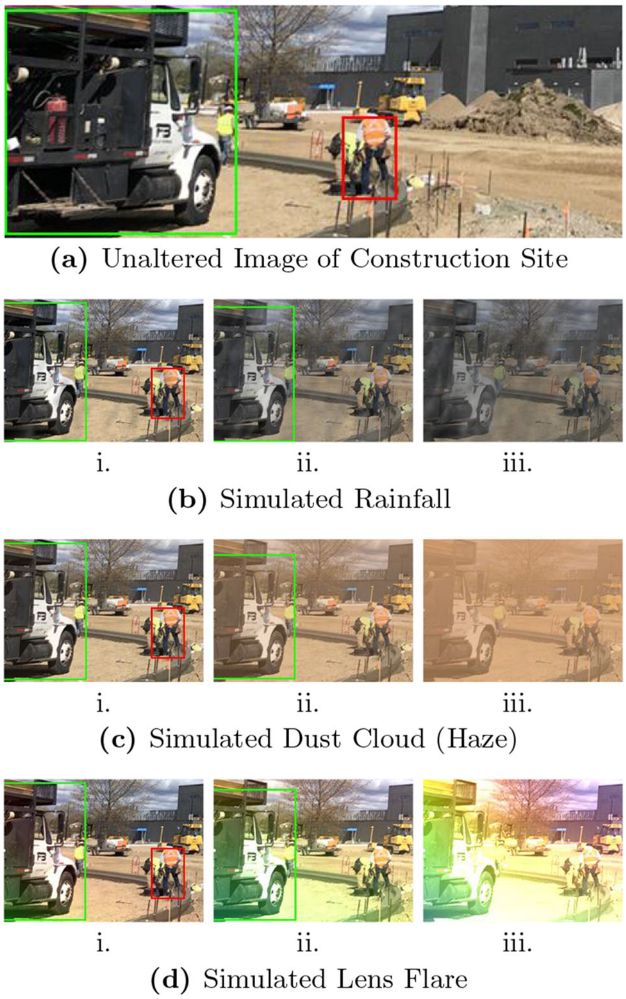


Fig. 18 Example of object detection at a construction site. A pedestrian is detected by an image-based object detector (a). New environmental phenomena are introduced in simulation, such as (b) rainfall, (c) dust clouds, and (d) lens flares. ENLIL explores different contexts to find examples that have (i) little impact, (ii) degrade, or (iii) compromise the object detector's ability to achieve validated design-time performance

The `/monitor_oracle` node forwards this change to the `/knowledge` node. The state change triggers execution of the `/analyze` node to evaluate the logical expression (Fig. 14) of the KAOS goal model depicted in Fig. 4. Upon evaluation, the `/analyze` node determines that adaptation is necessary, since the pre-condition associated with KAOS obstacle *O1* applies but the resolving goal *G17* is not satisfied. The tactic in Fig. 15 is selected and forwarded to `/plan`, which finds that the tactic's actions involve reducing the maximum velocity of the rover and increasing the buffer distance between the rover and objects in the environment. The `/plan` node then maps abstract tactic actions to a platform-specific procedure. Our rover uses a Timed Elastic Band (TEB) [95] planner provided with ROS to compute trajectories around objects in the environment. The abstract actions in Fig. 15 can be accomplished by setting the `min_obstacle_dist` and `max_vel_x` parameters of the TEB planner. Finally, the platform-specific procedure is forwarded to the `/execute` node, which is responsible for executing the reconfiguration of the rover. As a result, the rover moves slower and takes a wider berth around objects in the environment while the dust cloud is present.

Eventually, as the dust settles, the behavior oracle determines that the new environmental condition is expected to have *little impact* on the object detector (i.e., *Category 0*). Through the same sequence of steps described above, the `/analyze` node is triggered to execute by the state change. The `/analyze` node then determines that KAOS obstacle *O1* no longer applies and the KAOS goal model is satisfied. The `/analyze` node publishes a message to notify the `/plan` node that the selected tactic is no longer applicable. The `/plan` node then triggers the `/execute` node so that the previous operating parameters are restored (i.e., reset the minimum object distance and maximum rover velocity to their prior values).

Scenario 2. Lens Flare In a second scenario, the rover is navigating around a parked vehicle. Suppose the reflection of the sun on the windshield of the vehicle causes a momentary lens flare that blinds the cameras. The behavior oracle processes the camera image and determines that the impacted images will *compromise* the ability of the rover's object detector to perform as validated at design time (i.e., *Category 2*). The `/monitor_oracle` node publishes `behavior_category = 2` to `/knowledge`, triggering the `/analyze` node similar to the dust cloud scenario. The KAOS goal model is evaluated, but this time obstacle *O2* applies and its resolving goal *G16* is not satisfied. A tactic with a pre-condition matching *O2* and post-condition matching *G16* is selected and forwarded to the `/plan` node. The actions associated with the selected tactic are to halt the rover, and the `/plan` node generates a procedure to set the rover's maximum velocity to zero. Finally, the adaptation procedure is forwarded to the `/execute` node to update the rover accordingly, thereby transitioning it to a fail-safe state. When the lens flare eventually disappears (e.g., due to changing angle of the sun or cloud movements), the `/monitor_oracle` node publishes `behavior_category = 0`. The change in behavior category triggers the `/analyze` node to re-evaluate the KAOS goal model. The `/analyze` node then determines that *O2* no longer applies, subsequently triggering the `/plan` and `/execute` nodes to reset the selected tactic and restore the rover to its original configuration.

Scenario 3. Relaxation of Goals In a third scenario, we explore how RELAX may be used to explicitly deal with uncertainty on the rover. Suppose the rover uses the KAOS goal model in Fig. 4, where the KAOS goal model is initially not RELAX-ed to address run-time uncertainties. Figure 19 shows example utility values published on a ROS node

```
utility_params = {
    'behavior_cat':     0,
    'camera_status':    'enabled',
    'friction_val':     1.9,
    'gps_status':       'enabled',
    'lidar_status':     'enabled',
    'object_dist':      3.0,
    'object_types':
        ['pedestrian', 'car', 'car'],
    'planner':          'enabled',
    'point_cloud':      True,
    'tire_pressure':    220,
    'vel_x':            0.1,
}
```

Fig. 19 Example utility value input for Scenario 3

by the rover during operation. Table 4 shows the resulting MoDALAS evaluation of the RELAX-ed goal model to be *unsatisfied* since the original KAOS goal model expects a friction rate of 2 Newtons and a tire pressure of 221 kPa (individual goal results of *G21* (0.0) and *G22* (0.0), depicted in red). In this instance, an unsatisfactory evaluation of the goal model would trigger MoDALAS to execute an adaptation tactic to mitigate brake failure (e.g., notifying a human supervisor for intervention). However, the rover may be able to operate under the given values as the deviation is insignificant (e.g., inaccurate readings due to sensor noise). Thus, if we use the RELAX-ed KAOS goal model in Fig. 5, the system uncertainties may be tolerated and avoid the need for an immediate mitigation strategy that could negatively impact performance. Table 5 shows the resulting evaluation of the same rover configuration from Fig. 19, but instead using the RELAX-ed goal model from Fig. 5. Using fuzzy logic, the new model is tolerant to the sensor values with an accepted deviation range (individual goal results of *G21* (0.799) and *G22* (0.95), depicted in blue).

6 Related work

This paper explores methods for the assurance of cyber-physical LESs via models at run time [96]. Related studies have investigated the verification of robotic systems [97], including construction site applications [98]. Those efforts apply formal methods for verification but do not explicitly address LECs faced with uncertain conditions. RoCS [99] has been introduced as a self-adaptive framework for robotic systems, but in contrast to MoDALAS, it is not model-driven

Table 4 Example evaluation of a non-RELAX-ed goal model (Fig. 4)

Goal #	Evaluation Result
'G1'	1.0
'G2'	1.0
'G3'	1.0
'G4'	1.0
'G5'	1.0
'G6'	1.0
'G9'	0.0
'G10'	1.0
'G12'	1.0
'G13'	1.0
'G14'	1.0
'G16'	0.0
'G18'	1.0
'G19'	1.0
'G21'	0.0
'G22'	0.0
'O1'	0.0
'O2'	0.0
OVERALL EVALUATION	0.0
OVERALL SATISFACTION	0.0

Table 5 Example evaluation of a RELAX-ed goal model (Fig. 5)

Goal #	Evaluation Result
'G1'	1.0
'G2'	1.0
'G3'	1.0
'G4'	1.0
'G5'	1.0
'G6'	1.0
'G9'	0.0
'G10'	1.0
'G12'	1.0
'G13'	1.0
'G14'	1.0
'G16'	0.0
'G18'	1.0
'G19'	1.0
'G21'	0.799
'G22'	0.95
'O1'	0.0
'O2'	0.0
OVERALL EVALUATION	0.799
OVERALL SATISFACTION	1.0

nor focused on software assurance. To the best of our knowledge, MoDALAS is the first to include run-time assessment of LECs with respect to goal-oriented (i.e., KAOS) models.

Self-adaptive frameworks have used different approaches to address assurance. Zhang and Cheng [85] developed a state-based modeling approach for model checking assurance properties of SASs. Weyns and Iftikhar [100] proposed the use of model-based simulation to evaluate system requirements and determine adaptation procedures. ENTRUST [101] supports the development of an SAS driven by GSN assurance cases and verified by probabilistic models at run time. Similarly, AC-ROS [102] is a GSN model-driven self-adaptive framework for ROS-based applications, which includes self-assessment through utility functions as assurance evidence. In contrast, MoDALAS uses KAOS goal models to assess the satisfaction of system requirements of an LES at run time. Furthermore, these other approaches do not address uncertainty for LECs. MoDALAS enables an LES to self-adapt to mitigate failure from the use of LECs in untrusted contexts.

A number of design-time approaches have addressed how LECs handle uncertainty [103,104]. Smith et al. [105] have also explored the construction of assurance cases at design time to categorize LEC behavior with respect to hazardous behaviors. However, these methods do not enable self-assessments at run time and have limited applications for handling uncertain environmental conditions. MoDALAS differs from these works by using model inference (via *behavior oracles*) to assess LEC behavior at run time for *known unknown* environmental conditions.

Requirements modeling and specification research has also addressed environmental uncertainties for LECs. Whittle et al. [25,26] proposed RELAX as a requirements specification language that allows for the relaxation of requirements to adapt to environmental uncertainties. Fredericks et al. [74] and Ramirez et al. [106] proposed automation of relaxation of goal models and derivation of utility functions using RELAX. Letier et al. [72], Ramirez et al. [106], and Bencomo et al. [107] proposed the use of various utility functions (e.g., probabilistic) to evaluate and quantify partial satisfaction of a goal. Letier et al. [108] also proposed using Monte Carlo simulation to calculate the consequences of certain uncertainty factors. The MoDALAS framework has been designed to accommodate different requirements specification languages and support the corresponding analysis techniques, such as those mentioned above, to address uncertainty.

Recently, other researchers have explored system assurance for LESs and LECs. Asaadi et al. [109] proposed a probabilistic quantification of LES system confidence based on functional capabilities and dependability attributes. Boursinos et al. [110] proposed a conformal prediction framework, leveraging previous normal values to check for abnormalities. Weyns et al. [111] proposed combining MAPE, control theory, and machine learning for better adaptive systems. Ferreira et al. [80] proposed an orthogonal approach to *assess* the effectiveness of safety monitors [112]: i) to identify out-of-distribution data for image classification machine learning algorithms and ii) to assess the safety monitors' abilities to correct the incorrect behavior of the model. In contrast, MoDALAS is a framework that uses evolutionary computing to *create* behavior oracles to determine which LEC is appropriate for a given operational context, which prevents the use of LECs in conditions for which they have not been adequately trained. In essence, behavior oracles can be used as safety monitors. As such, future work may explore the use of Ferreira et al.'s framework to assess behavior oracles generated with MoDALAS for various performance metrics of interest.

7 Conclusion

This paper introduced the MoDALAS framework for using requirements models at run time to automatically address the assurance of safety-critical systems with machine learning components. Due to uncertainties about the ability of LECs to generalize to complex environments, methods are needed to assess their capability at run time and adapt LESs to mitigate the use of LECs in uncertain run-time conditions. MoDALAS assesses the trustworthiness of LECs with *behavior oracles* and reconfigures an LES to maintain satisfaction of system requirements at run time.

MoDALAS addresses uncertainties about the assurance of an autonomous LES when facing uncertain run-time conditions (e.g., *known unknown* phenomena). This paper described a proof-of-concept prototype of MoDALAS for an autonomous rover LES with an object detector LEC. MoDALAS adapts the rover to maintain safety requirements under run-time conditions where the object detector is deemed unreliable. This paper also demonstrated how MoDALAS can leverage the RELAX language and fuzzy logic run-time evaluation to manage uncertainties in requirements.

Future work will explore the management of dynamic assurance cases and goal models, as well as the inclusion of security assurance cases within the MoDALAS framework [113]. Working with our industrial collaborators, we will perform additional empirical studies. We will expand our studies with additional benchmark datasets and other sources of training data (e.g., various ranges of hazard types and different types of behavior categories, etc.) in order to continue to improve the ability of MoDALAS to address assurance of LESs in the face of a broad range of uncertainty factors and diverse operating contexts.

Acknowledgements We greatly appreciate contributions from Robert Jared Clark on our preliminary work. This work has been supported in part by a grant from the National Science Foundation (NSF) (DBI-0939454) and by the Air Force Research Laboratory (AFRL) under agreements FA8750-16-2-0284 and FA8750-19-2-0002. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFRL, or other sponsors.

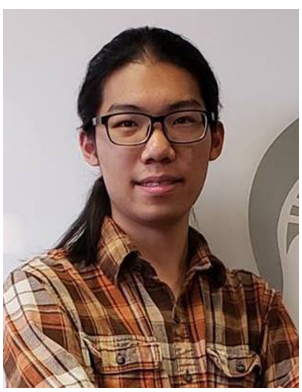
References

1. Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.): Assurances for Self-Adaptive Systems: Principles, Models, and Techniques. Springer, Berlin, Heidelberg (2013)
2. Langari, Z., Maibaum, T.: Safety cases: a review of challenges. Paper presented at 1st Int. Workshop on Assurance Cases for Software-Intensive Systems (ASSURE 2013) (2013)
3. Kocić, J., Jovičić, N., Drndarević, V.: An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. Sensors **19**(9), 2064 (2019)
4. Wu, B., Iandola, F., Jin, P.H., Keutzer, K.: Squeezedet: unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving (2017)
5. Yao, X., Wang, X., Wang, S.-H., Zhang, Y.-D.: A comprehensive survey on convolutional neural network in medical image analysis. Multimed. Tools Appl. 1–45 (2020)
6. Abdou, M.A.: Literature review: efficient deep neural networks techniques for medical image analysis. Neural Comput. Appl. 1–22 (2022)
7. Lu, R., Hong, S.H.: Incentive-based demand response for smart grid with reinforcement learning and deep neural network. Appl. Energy **236**, 937–949 (2019)
8. Tuncali, C.E., Kapinski, J., Ito, H., Deshmukh, J.V.: Reasoning about safety of learning-enabled components in autonomous cyber-physical systems. Paper presented at 55th Annual Design Automation Conf. (DAC 2018) (2018)
9. Kawaguchi, K., Kaelbling, L.P., Bengio, Y.: Generalization in deep learning. Tech. Rep., MIT (2018). https://lis.csail.mit.edu/pubs/kawaguchi-techreport18.pdf
10. Yu, F., et al.: Interpreting and evaluating neural network robustness. Paper presented at 28th International Joint Conf. on Artificial Intelligence (IJCAI 2019) (2019)
11. Knight, W.: The Dark Secret at the Heart of AI. MIT Technology Review **Artificial intelligence/Machine learning** (2017). https://www.technologyreview.com/2017/04/11/5113/the-dark-secret-at-the-heart-of-ai/
12. Rushby, J.: The Interpretation and Evaluation of Assurance Cases. Tech. Rep. SRI-CSL-15-01, Computer Science Laboratory, SRI International, Menlo Park, CA (2015). https://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf
13. 2022 IEEE Conference on Assured Autonomy (ICAA) (2022)
14. Assured autonomy workshop series. https://cra.org/ccc/visioning/visioning-activities/2019-activities/assured-autonomy/. Accessed: 2022-08-22
15. Workshop on assured autonomous systems 2020 (2020)
16. Air Force Office of Scientific Research.: Center of excellence: Assured autonomy in contested environments. https://www.federalgrants.com/CENTER-OF-EXCELLENCE-Assured-Autonomy-in-Contested-Environments-71233.html (2018)
17. Neema, S.: Assured autonomy. https://www.darpa.mil/program/assured-autonomy (2017)
18. Schumann, J., Gupta, P., Liu, Y.: Application of Neural Networks in High Assurance Systems: A Survey of Studies in Computational Intelligence (SCI), vol. 268. Springer, Berlin, Heidelberg (2010)
19. Hartsell, C., et al.: Model-based design for CPS with learning-enabled components. Paper presented at Workshop on Design Automation for CPS and IoT (DESTION 2019) (2019)
20. van Lamsweerde, A., Letier, E.: From object orientation to goal orientation: a paradigm shift for requirements engineering. Paper presented at Radical Innovations of Software and Systems Engineering in the Future (RISSEF 2002) (2004)
21. Kephart, J.O., Das, R.: Achieving self-management via utility functions. IEEE Internet Comput. **11**(1), 40–48 (2007). https://doi.org/10.1109/MIC.2007.2
22. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
23. Langford, M.A., Cheng, B. H.C.: "Know What You Know": predicting behavior for learning-enabled systems when facing uncertainty. Paper presented at 16th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021) (2021)
24. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. Trans. Softw. Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785
25. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C.,Bruel, J.-M.: RELAX: incorporating uncertainty into the specification of self-adaptive systems. Paper presented at 17th IEEE Int. Requirements Engineering Conf. (RE 2009) (2009)
26. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010). https://doi.org/10.1007/s00766-010-0101-0
27. Quigley, M., et al.: ROS: an open-source robot operating system. Paper presented at Int. Conf. on Robotics and Automation Workshop on Open Source Software (ICRA Workshop 2009) (2009)
28. Jiao, L., et al.: A survey of deep learning-based object detection. IEEE Access **7**, 128837–128868 (2019). https://doi.org/10.1109/ACCESS.2019.2939201
29. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT, Cambridge (2016)
30. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017). https://doi.org/10.1145/3065386
31. Liu, L., et al.: Deep learning for generic object detection: a survey. Int. J. Comput. Vis. **128**, 261–318 (2018). https://doi.org/10.1007/s11263-019-01247-4
32. Kuutti, S., Bowden, R., Jin, Y., Barber, P., Fallah, S.: A vehicle survey of deep learning applications to autonomous control. IEEE Trans. Intell. Transp. Syst. **22**(2), 712–733 (2021). https://doi.org/10.1109/TITS.2019.2962338
33. Ravindran, R., Santora, M.J., Jamali, M.M.: Multi-object detection and tracking, based on DNN, for autonomous vehicles: a review. IEEE Sens. J. **21**(5), 5668–5677 (2020)
34. Schwarting, W., Alonso-Mora, J., Rus, D.: Planning and decision-making for autonomous vehicles. Annu. Rev. Control Robot. Auton. Syst. **1**(1), 187–210 (2018). https://doi.org/10.1146/annurev-control-060117-105157
35. Janai, J., Güney, F., Behl, A., Geiger, A.: Computer vision for autonomous vehicles: problems, datasets and state-of-the-art. CoRR (2017). arXiv:1704.05519
36. Kuutti, S., et al.: A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications. IEEE Internet Things J. **5**(2), 829–846 (2018). https://doi.org/10.1109/JIOT.2018.2812300
37. Borg, M., et al.: Safely entering the deep: a review of verification and validation for machine learning and a challenge elicitation in

the automotive industry. J. Automot. Softw. Eng. **1**, 1–19 (2019). https://doi.org/10.2991/jase.d.190131.001
38. Calikli, G., Bener, A.: Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. Paper presented at Proc. 6th Int. Conf. on Predictive Models in Software Engineering (PROMISE 2010) (2010)
39. Whang, S.E., Lee, J.-G.: Data collection and quality challenges for deep learning. Proc VLDB Endow. **13**(12), 3429–3432 (2020). https://doi.org/10.14778/3415478.3415562
40. Jo, J., Bengio, Y.: Measuring the Tendency of CNNs to Learn Surface Statistical Regularities. CoRR (2017). arXiv:1711.11561
41. Bengio, Y.: Priors for deep learning of semantic representations. Keynote at ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS) (2020)
42. Barredo Arrieta, A., et al.: Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf. Fusion **58**, 82–115 (2020). https://doi.org/10.1016/j.inffus.2019.12.012
43. Szegedy, C., et al.: Intriguing Properties of Neural Networks. CoRR (2013). Appeared in ICLR 2014. arXiv:1312.6199
44. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and Harnessing Adversarial Examples. CoRR (2014). Appeared in ICLR 2015. arXiv:1412.6572
45. Nguyen, A., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: high confidence predictions for unrecognizable images. Paper presented at IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2015) (2015)
46. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: automated whitebox testing of deep learning systems. Paper presented at 26th Symposium on Operating Systems Principles (SOSP 2017) (2017)
47. Tian, Y., Pei, K., Jana, S., Ray, B.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. Paper presented at 40th Int. Conf. on Software Engineering (NeurIPS 2018) (2018)
48. Ma, L., et al.: DeepGauge: multi-granularity testing criteria for deep learning systems. Paper presented at 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018) (2018)
49. Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S.: DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. Paper presented at 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018) (2018)
50. Sun, Y., et al.: DeepConcolic: testing and debugging deep neural networks. Paper presented at 41st Int. Conf. on Software Engineering (ICSE 2019) (2019)
51. Xie, X., et al.: DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. Paper presented at 28th ACM SIGSOFT Int. Symposium on Software Testing and Analysis (2019)
52. Langford, M.A., Cheng, B. H.C.: Enhancing learning-enabled software systems to address environmental uncertainty. Paper presented at 16th IEEE Int. Conf. on Autonomic Computing (ICAC 2019) (2019)
53. Odena, A., Olsson, C., Andersen, D., Goodfellow, I.: TensorFuzz: debugging neural networks with coverage-guided fuzzing. Paper presented at 36th Int. Conf. on Machine Learning (PMLR 2019) (2019)
54. Berend, D., et al.: cats are not fish: deep learning testing calls for out-of-distribution awareness. Paper presented at 35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2020) (2020)
55. Ma, W., Papadakis, M., Tsakmalis, A., Cordy, M., Traon, Y.L.: Test selection for deep learning systems. ACM Trans. Softw. Eng. Methodol. **30**(2), 1–22 (2021). https://doi.org/10.1145/3417330
56. Naeem Irfan, M., Oriat, C., Groz, R.: Model inference and testing. Adv. Comput. **89**, 89–139 (2013). https://doi.org/10.1016/B978-0-12-408094-2.00003-5
57. Fraser, G., Walkinshaw, N.: Assessing and generating test sets in terms of behavioural adequacy. Softw. Test. Verif. Reliab. **25**(8), 749–780 (2015). https://doi.org/10.1002/stvr.1575
58. Papadopoulos, P., Walkinshaw, N.: Black-box test generation from inferred models. Paper presented at 4th Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2015) (2015)
59. Aichernig, B.K., et al.: Learning a behavior model of hybrid systems through combining model-based testing and machine learning. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) Testing Software and Systems. Lecture Notes in Computer Science (ICTSS 2019), vol. 11812. Springer, Cham (2019)
60. Gawlikowski, J., et al.: A survey of uncertainty in deep neural networks. CoRR (2021). arXiv:2107.03342
61. Cortés-Ciriano, I., Bender, A.: Deep confidence: a computationally efficient framework for calculating reliable prediction errors for deep neural networks. J. Chem. Inf. Model. **59**(3), 1269–1281 (2018). https://doi.org/10.1021/acs.jcim.8b00542
62. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops, pp. 48–70. Springer, Berlin, Heidelberg (2009)
63. IBM.: an architectural blueprint for autonomic computing. Tech. Rep. 3rd ed., IBM (2005). https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf
64. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and analyzing MAPE-K feedback loops for self-adaptation. Paper presented at 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015) (2015)
65. Cheng, S.-W.: Rainbow: Cost-effective software architecture-based self-adaptation. Ph.D. thesis, Carnegie Mellon University (2008)
66. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. Paper presented at Int. Conf. on Autonomic Computing (ICAC 2004) (2004)
67. deGrandis, P., Valetto, G.: Elicitation and utilization of application-level utility functions. Paper presented at 6th Int. Conf. on Autonomic Computing (ICAC 2009) (2009)
68. Object Management Group. Structured assurance case metamodel (SACM) Version 2.1. Tech. Rep., OMG (2020). https://www.omg.org/spec/SACM
69. Goodenough, J., Weinstock, C., Klein, A.: Toward a Theory of Assurance Case Confidence. Tech. Rep. CMU/SEI-2012-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2012). https://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067
70. ACWG.: Goal structuring notation community standard (Version 2). Tech. Rep., Assurance Case Working Group, Safety-Critical Systems Club (2018). https://scsc.uk/r141B:1
71. Lapouchnian, A.: goal-oriented requirements engineering: an overview of the current research. Tech. Rep., University of Toronto (2005). http://www.cs.utoronto.ca/~alexei/pub/Lapouchnian-Depth.pdf
72. Letier, E., van Lamsweerde, A.: Reasoning about partial goal satisfaction for requirements and design engineering. Paper presented at 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT 2004/FSE-12) (2004)
73. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.-M.: RELAX: incorporating uncertainty into the specification of self-adaptive systems. Paper presented at 17th IEEE Int. Requirements Engineering Conf. (RE 2009) (2009)
74. Fredericks, E.M., DeVries, B., Cheng, B.H.C.: Autorelax: automatically relaxing a goal model to address uncertainty. Empir. Softw. Engg. **19**(5), 1466–1501 (2014). https://doi.org/10.1007/s10664-014-9305-0

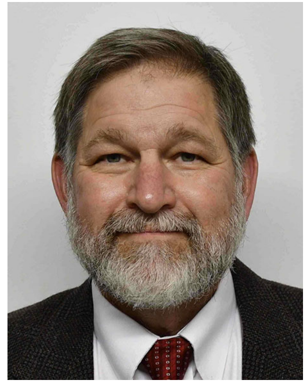
75. Zadeh, L.: Fuzzy logic. Computer **21**(4), 83–93 (1988). https://doi.org/10.1109/2.53
76. Hájek, P.: Metamathematics of Fuzzy Logic. Springer, Dordrecht (2013)
77. Zadeh, L.A.: Fuzzy Sets (1996)
78. Clarke, E.M., Jr., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking, 2nd edn. MIT Press, Cambridge (2018)
79. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. Paper presented at 18th IEEE Int. Requirements Engineering Conf. (RE 2010) (2010)
80. Ferreira, R.S., Arlat, J., Guiochet, J., Waeselynck, H.: Benchmarking safety monitors for image classifiers with machine learning, 7–16 (IEEE, 2021)
81. Sokolova, M., Lapalme, G.: A systematic analysis of performance measures for classification tasks. Inf. Process. Manag. **45**, 427–437 (2009). https://doi.org/10.1016/j.ipm.2009.03.002
82. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing, 2nd edn. Springer-Verlag, Berlin, Heidelberg (2015)
83. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. Paper presented at Int. Workshop on Self-Adaptation and Self-Managing Systems (SEAMS (2006) (2006)
84. Palmerino, J., Yu, Q., Desell, T., Krutz, D.: Improving the decision-making process of self-adaptive systems by accounting for tactic volatility. Paper presented at 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2019) (2019)
85. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. Paper presented at 28th Int. Conf. on Software Engineering (ICSE 2006) (2006)
86. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16**(11), 1293–1306 (1990). https://doi.org/10.1109/32.60317
87. Melenbrink, N., Werfel, J., Menges, A.: On-site autonomous construction robots: towards unsupervised building. Autom. Constr. **119**, 103312 (2020). https://doi.org/10.1016/j.autcon.2020.103312
88. Malone, D.: Rovers set to invade construction jobsites (2019). https://www.bdcnetwork.com/rovers-set-invade-construction-jobsites
89. Weyns, D., Holvoet, T., Schelfthout, K., Wielemans, J.: Applying multi-agent systems in practice, decentralized control of automatic guided vehicles (2008)
90. Dersten, S., Wallin, P., Fröberg, J., Axelsson, J.: Analysis of the information needs of an autonomous hauler in a quarry site. Paper presented at 11th System of Systems Engineering Conf. (SoSE 2016) (2016)
91. Goldfain, B., et al.: AutoRally: an open platform for aggressive autonomous driving. IEEE Control Syst. Mag. **39**(1), 26–55 (2019). https://doi.org/10.1109/MCS.2018.2876958
92. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. Paper presented at IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (2004)
93. Hartley, R.I., Sturm, P.: Triangulation. Comput. Vis. Image Underst. **68**(2), 146–157 (1997). https://doi.org/10.1006/cviu.1997.0547
94. Rusu, R.B., Cousins, S.: 3D is here: point cloud library (PCL). Paper presented at IEEE Int. Conf. on Robotics and Automation (ICRA 2011) (2011)
95. Obstacle Avoidance and Robot Footprint Model (2019). http://wiki.ros.org/teb_local_planner/Tutorials/Obstacle%20Avoidance%20and%20Robot%20Footprint%20Model
96. Dalpiaz, F., Borgida, A., Horkoff, J., Mylopoulos, J.: Runtime goal models: keynote. Paper presented at 7th IEEE Int. Conf. on Research Challenges in Information Science (RCIS 2013) (2013)
97. Huang, W., et al.: Formal verification of robustness and resilience of learning-enabled state estimation systems for robotics. CoRR (2020). arXiv:2010.08311
98. Gu, R., Marinescu, R., Seceleanu, C., Lundqvist, K.: Formal verification of an autonomous wheel loader by model checking. Paper presented at 6th Conf. on Formal Methods in Software Engineering (FormaliSE 2018) (2018)
99. Ramos, L., et al.: The RoCS framework to support the development of autonomous robots. J. Softw. Eng. Res. Dev. **7**, 1–14 (2019). https://doi.org/10.5753/jserd.2019.470
100. Weyns, D., Iftikhar, M.U.: Model-based simulation at runtime for self-adaptive systems. Paper presented at 15th Int. Conf. on Autonomic Computing (ICAC 2016) (2016)
101. Calinescu, R., et al.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. **44**(11), 1039–1069 (2018)
102. Cheng, B.H.C., Clark, R.J., Fleck, J.E., Langford, M.A., McKinley, P.K.: AC-ROS: assurance case driven adaptation for the robot operating system (2020). Paper presented at 23rd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2020)
103. Song, Q., Shepperd, M., Cartwright, M., Mair, C.: Software defect association mining and defect correction effort prediction. IEEE Trans. Softw. Eng. **32**(2), 69–82 (2006). https://doi.org/10.1109/TSE.2006.1599417
104. Rodriguez, D., Ruiz, R., Riquelme, J.C., Harrison, R.: A study of subgroup discovery approaches for defect prediction. Inf. Softw. Technol. **55**(10), 1810–1822 (2013). https://doi.org/10.1016/j.infsof.2013.05.002
105. Smith, C., Denney, E., Pai, G.: Hazard contribution modes of machine learning components. Tech. Rep., OSTI (2020). https://www.osti.gov/biblio/1606667. (AAAI Workshop: SafeAI 2020)
106. Ramirez, A.J., Cheng, B.H.C.: Automatic derivation of utility functions for monitoring software requirements. Paper presented at 14th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2011) (2011)
107. Bencomo, N., Belaggoun, A.: Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks. Paper presented at Int. Working Conf. on Requirements Engineering Foundation for Software Quality (REFSQ) (2013) (2013)
108. Letier, E., Stefan, D., Barr, E.T.: Uncertainty, risk, and information value in software requirements and architecture. Paper presented at 36th Int. Conf. on Software Engineering (ICSE 2014) (2014)
109. Asaadi, E., Denney, E., Pai, G.: quantifying assurance in learning-enabled systems. Paper presented at Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP 2020) (2020)
110. Boursinos, D., Koutsoukos, X.: Assurance Monitoring of Learning-Enabled Cyber-Physical Systems Using Inductive Conformal Prediction Based on Distance Learning. Artif. Intell. Eng. Des. Anal. Manuf. **35**(2), 251–264 (2021). https://doi.org/10.1017/S089006042100010X
111. Weyns, D., et al.: towards better adaptive systems by combining MAPE, control theory, and machine learning. Paper presented at 16th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021) (2021)
112. Machin, M., et al.: SMOF: a safety monitoring framework for autonomous systems. IEEE Trans. Syst. Man Cybern. Syst. **48**(5), 702–715 (2016)
113. Jahan, S., et al.: MAPE-K/MAPE-SAC: An Interaction Framework for Adaptive Systems with Security Assurance Cases. Futur. Gener. Comput. Syst. **109**, 197–209 (2020). https://doi.org/10.1016/j.future.2020.03.031

Michael Austin Langford is currently employed by The Aero space Corporation's Data Science and Artificial Intelligence Department, where he is actively researching applications of machine learning and human-machine teaming with an emphasis on mission assurance. He received the Ph.D. degree in computer science from Michigan State University and M.S. degree for computer science from Columbia University. His research interests include artificial intelligence, autonomous systems, computer vision, deep learning, evolutionary computation, and software engineering.



Kenneth H. Chan is a Ph.D.student in the Department of Computer Science and Engineering at Michigan State University, where he also obtained his M.S. and B.S. in computer science. His research interests include software assurance, system security, robustness for learning-enabled systems, evolutionary computing, and software engineering.

Jonathon E. Fleck is a Ph.D.student in the Department of Mathematics at the University of Utah. He obtained his M.S. and B.S. in computer science at Michigan State University. His current research interests include geometric group theory and homological algebra.



Philip K. McKinley is a Professor in the Department of Computer Science and Engineering at Michigan State University, where he has been on the faculty since 1990. He was previously a Member of Technical Staff at Bell Laboratories. Dr. McKinley received the Ph.D. from the University of Illinois at Urbana-Champaign. His research interests include autonomous systems, evolutionary robotics, artificial life, and self-adaptive software.

Betty H.C. Cheng is a professor in the Department of Computer Science and Engineering at Michigan State University. She is also the Industrial Relations Manager and senior researcher for BEACON, the National Science Foundation Science and Technology Center in the area of Evolution in Action. Her research interests include self-adaptive autonomous systems, requirements engineering, model-driven engineering, automated software engineering, and harnessing evolutionary computation and search-based techniques to address software engineering problems. These research areas are used to support the development of high-assurance adaptive systems that must continuously deliver acceptable behavior, even in the face of environmental and system uncertainty. Example applications include intelligent transportation and vehicle systems. She collaborates extensively with industrial partners in her research projects in order to ensure real-world relevance of her research and to facilitate technology exchange between academia and industry. Her collaborators include Ford, General Motors, ZF, BAE, Motorola, and Siemens. Previously, she was awarded a NASA/JPL Faculty Fellowship to investigate the use of new software engineering techniques for a portion of the NASA space shuttle software. She has recently launched new projects in the areas of model-driven approaches to sustainability, cyber security for automotive systems, and feature interaction detection and mitigation for autonomic systems, all in the context of operating under uncertainty while maintaining assurance objectives. Her research has been funded by several federal funding agencies, including NSF, AFRL, ONR, DARPA, NASA, ARO, and numerous industrial organizations. She serves on the journal editorial boards for Requirements Engineering and Software and Systems Modeling; she is Co-Associate Editor-in-Chief for IEEE Transactions for Software Engineering, where she previously served twice as an Associate Editor. She was the Technical Program Co-Chair for IEEE International Conference on Software Engineering (ICSE-2013), the premier and flagship conference for software engineering. She received her Bachelor of Science degree from Northwestern University, and her MS and PhD from the University of Illinois-Urbana Champaign, all in computer science. She may be reached at the Department of Computer Science and Engineering, Michigan State University, 3115 Engineering Building, 428 S. Shaw Lane, East Lansing, MI 48824; chengb@msu.edu; www.cse.msu.edu/~chengb.