2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS) | 978-1-6654-3495-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/MODELS50736.2021.00027

MoDALAS: Model-Driven Assurance for Learning-Enabled Autonomous Systems

Michael Austin Langford, Kenneth H. Chan, Jonathon Emil Fleck, and Philip K. McKinley, Betty H.C. Cheng
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA
{langfo37, chanken1, fleckjo1, mckinle3, chengb}@msu.edu

***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. Using this framework enables stakeholders to have more confidence that LECs are used only in contexts comparable to those validated at design time.**

***Index Terms*—goal-based modeling, self-adaptive systems, artificial intelligence, machine learning, models at run time, cyber physical systems, behavior oracles, autonomous vehicles**

I. Introduction

The integration of machine learning into autonomous systems is potentially problematic for high-assurance [1], safety-critical [2] applications, 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 all possible operating scenarios and guarantee that all system responses are safe [3]. However, machine learning components, such as Deep Neural Networks (DNNs) [4], are associated with uncertainties concerning generalizability [5], robustness [6], and interpretability [7]. A rigorous *software assurance* [8] process is needed to account for these issues of uncertainty. This paper proposes a goal-oriented modeling approach to address the assurance 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 [9] [10]. 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 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 framework for Model-Driven Assurance for Learning-enabled Autonomous Systems (MoDALAS) through the use of goal-oriented *models at run time*. With run-time self-assessment of its LECs, a goal model-driven LES can adapt to satisfy system requirements when exposed to environmental uncertainty. MoDALAS supports the run-time monitoring of an LES with respect to functional goal models, assesses the trustworthiness of LECs, and adapts the LES to mitigate the use of LECs in untrusted contexts. Specifically, predictions of LEC behavior under various operating conditions are directly referenced by goal models that include potential conflicts and resolutions relating to different LEC behaviors (including untrusted contexts). Moreover, MoDALAS enables seamless monitoring and management of both LECs and "traditional" system components (both hardware and software) that do not involve machine learning.

In MoDALAS, online system verification is established by the run-time monitoring of KAOS goal models [11] for functional requirements [12]. Controlled by a self-adaptive feedback loop [13], MoDALAS includes a *behavior oracle* [14] for each LEC. Analogous to a *test oracle* [15], a behavior oracle determines how an LEC will behave in response to given inputs. In MoDALAS, behavior oracles are used to assess the capability of LECs 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.

A proof-of-concept prototype of MoDALAS is described for an autonomous rover LES equipped with a camera-

[1]This paper terms 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 implementing one or more LECs (e.g., an autonomous rover).

978-1-6654-3495-9/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS50736.2021.00027

based object detector LEC [16]. DNNs play two roles in this work: 1) a DNN provides object detection capabilities for an autonomous 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 be detrimental for 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. Furthermore, MoDALAS enables the rover to adapt accordingly by entering a more cautious operating mode or, under extreme conditions, a fail-safe mode. The remainder of this paper is organized as follows. Section II reviews background topics. Section III describes MoDALAS in detail. Section IV presents an implementation of MoDALAS for an autonomous rover. Section V reviews related work. Finally, Section VI summarizes the paper and briefly discusses future work.

II. Background

This section reviews background topics and enabling technologies that are relevant to the design and operation of MoDALAS, including assurance challenges for LECs, assurance with goal-oriented modeling, and self-adaptive systems.

A. Challenges For Learning-Enabled System

Promising results from the use of deep learning [4] to solve traditionally difficult problems such as image classification [17] and object detection [18] have led to an increase in use of LECs in autonomous systems [19]. For example, DNNs have been implemented for the planning [20], perception [21], and mapping/localization [22] of autonomous vehicles. An advantage for using DNNs in these tasks is less dependence on human feature engineering, as features are learned directly from input data [23]. However, increased dependence on input data has introduced new challenges concerning data bias [24] and data quality [25]. Furthermore, research has shown that DNNs are sensitive to *surface statistical regularities* [26], causing decisions to be based on superficial, statistically common features in training data rather than *semantically relevant* [27] features to the target task (e.g., deciding an image contains a dog based only on a pattern of grass that is regularly 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 assurance guarantees.

One major concern is the *robustness* [6] [28] of a DNN to input perturbations. Research has shown that *adversarial examples* [29] [30] [31] 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* [29] (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 [32], DeepTest [33], DeepGauge [34], DeepRoad [35], DeepConcolic [36], DeepHunter [37], ENKI [38], and TensorFuzz [39], 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 [40]. However, an empirical study by Ma et al. [41] 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). 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 [42] 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 [43] [44]. Aichernig et al. [45] have also described how to construct *behavior models* of cyber-physical systems through deep learning. Langford and Cheng [14] 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*. Since ENLIL uses simulated phenomena when creating behavior oracles, a *reality gap* may exist [46]. We assume such phenomena can be simulated to an acceptable degree of accuracy and behavior oracles can be updated as real-world data is acquired. In contrast to *out-of-distribution* methods that assess *confidence* in DNN outputs [47] [48], 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 uses ENLIL behavior oracles to 1) assess the ability of LECs to satisfy functional goals at run time and 2) adapt accordingly.

B. Software Assurance and Goal-Based Modeling

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 [49]. *Assurance cases* provide a means to certify that software operates as intended by describing the validation process and supporting evidence [50]. 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 Goal Structuring Notation (GSN) [51]. GSN enables an argument to be defined as a graphical model. Each claim is represented as a *goal* and each piece of supporting evidence (e.g., test cases, documentation, etc.) is represented as a *solution*.

Whereas the focus of GSN is on software certification, KAOS goal modeling [11] supports a hierarchical decomposition of high-level functional and performance objectives into leaf-level system requirements (i.e., goal-oriented requirements engineering [52]). KAOS goal models enable a formal goal-oriented analysis of how system requirements are interrelated as well as threats to requirement satisfaction. *Goals* represent atomic objectives of a system at varying levels of abstraction, with sub-goals 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 sub-goals 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.

C. Self-Adaptive Systems and Utility Functions

Self-adaptation provides software systems the capability to adjust system behavior in response to the environment [13]. Self-Adaptive Systems (SASs) commonly operate with a centralized feedback controller (i.e., *adaptation manager*) to observe and adapt managed components of a system [53]. Fig. 1 illustrates a common implementation of an adaptation manager called the *Monitor-Analyze-Plan-Execute over a Knowledge base* (MAPE-K) loop [54] [55]. 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.

One approach to monitoring SAS behavior is to implement *utility functions* [56] [57]. Utility functions map system attributes (i.e., the system state) into real scalar values to express a degree of goal (i.e., requirements) satisfaction [58]. Explicitly, a utility function takes the following form.

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

The *utility value* is a real scalar value ($u \in [0, 1]$), and the *system state* vector ($\mathrm{v} = [s_0, ..., s_n]$) 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 [12]. Notably, MoDALAS demonstrates that utility functions can provide a common, unified approach to characterize the behavior of *both* LECs and non-learning system components.

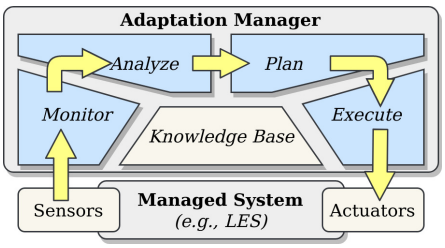


Fig. 1: High-level depiction of a MAPE-K feedback loop to manage adaptations for an SAS [53].

The proposed MoDALAS framework enables run-time verification of an LES by associating utility functions to KAOS goal models 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.

III. Methodology/Framework

This section describes the proposed MoDALAS framework for goal-based self-adaptation of an LES. Fig. 2 shows a data flow diagram (DFD) of the framework. Circles represent 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.

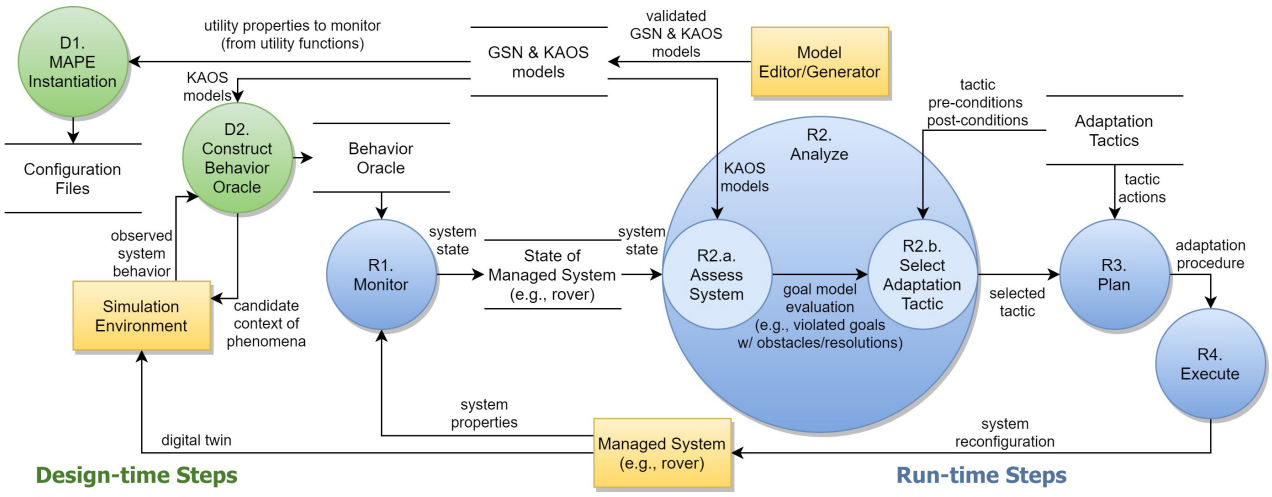


Fig. 2: High-level data flow diagram of MoDALAS. Processes are shown as circles, external entities are shown as boxes, and persistent data stores are shown within parallel lines. Directed lines between entities show the flow of data.

Specific implementation details on how MoDALAS is applied to the autonomous rover are provided in Section IV. Each step in Fig. 2 is described in detail as follows.

A. System Modeling at Design Time

MoDALAS assumes that assurance cases and goal models have been constructed and validated at design time through methods such as *model checking* [59]. In this paper, assurance cases are modeled using GSN, though alternative methods may also be used to describe an assurance case [8]. A simple example GSN model is shown in Fig. 3, 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 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.

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 is capable of *locating* zero or more objects within a camera image and *classifying* the type of each object [16]. The trustworthiness of the object detector depends on how similar the run-time environment is to its training experience. 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.

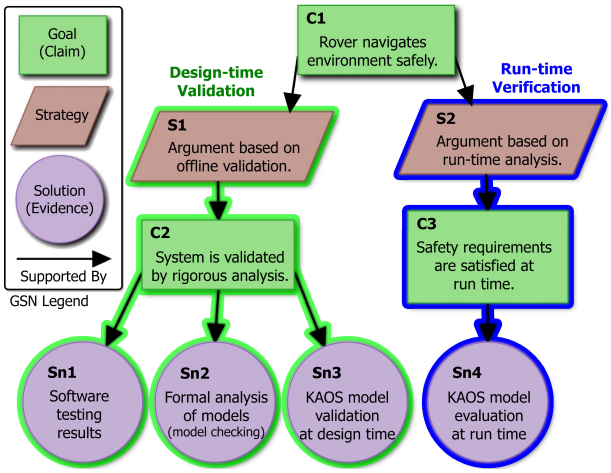


Fig. 3: Example GSN assurance case for *design-time* and *run-time* validation of a rover. At design time, validation is supported by formal proofs, test results, and simulation (highlighted in green). At run time, verification is supported by the evaluation of a KAOS goal model (highlighted in blue).

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

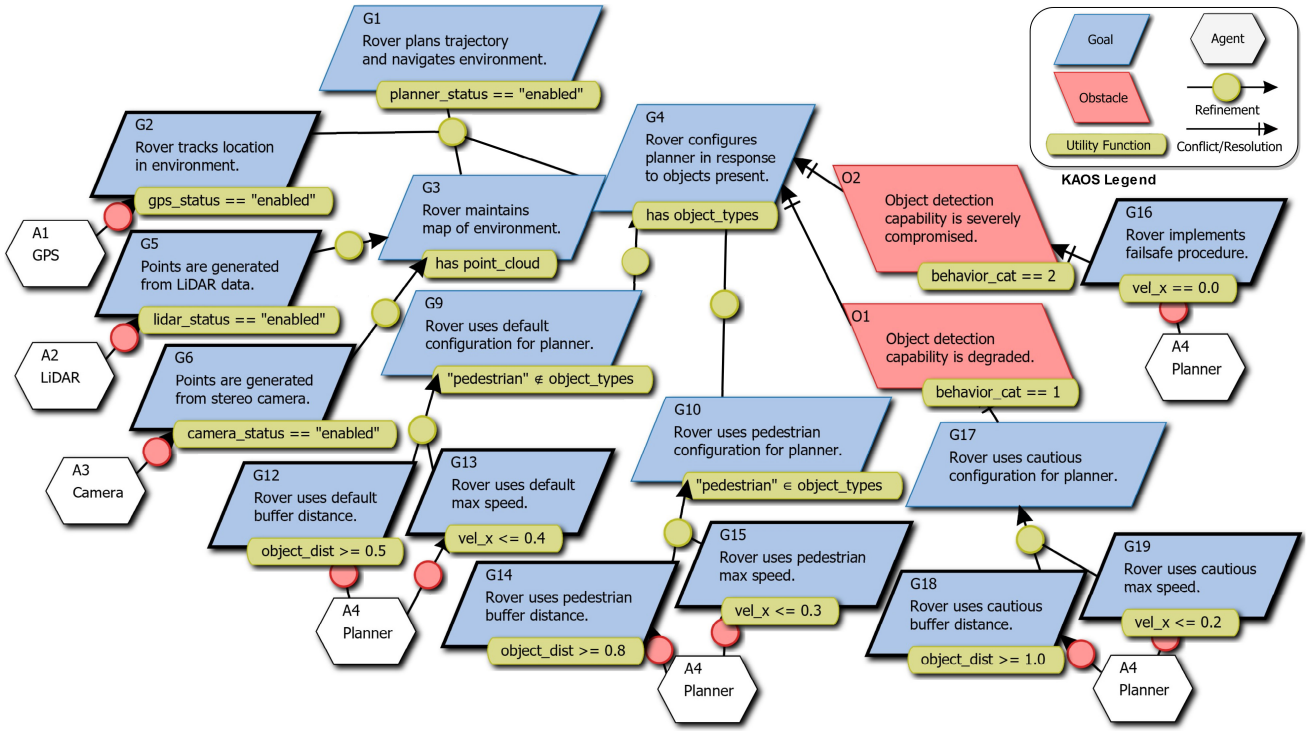


Fig. 4: Example KAOS goal model. *Goals* (blue parallelograms) represent system objectives. The top-level goal (*G1*) is refined by sub-goals down to leaf-level system requirements. *Agents* (white hexagons) represent entities responsible for accomplishing requirements. *Obstacles* (red parallelograms)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

encountered in the environment. Additional KAOS obstacles and resolutions can be included, depending on the LEC, targeted behavior categories, and LES requirements.

Step D1. MAPE Instantiation: 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.

Step D2. Construction of Behavior Oracle: To monitor and assess the trustworthiness of LECs at run time, MoDALAS leverages *behavior oracles* generated by ENLIL [14] for each individual LEC. Behavior oracles are implemented as DNNs 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. In essence, behavior oracles serve as a utility function for assessing 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 (`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 I 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*).

TABLE I: Behavior categories for an object detector.

Category	Classification		Definition
0	*"little impact"*	●	$0\% \leq \delta_{recall} < 5\%$
1	*"degraded"*	●	$5\% \leq \delta_{recall} < 10\%$
2	*"compromised"*	●	$10\% \leq \delta_{recall}$

ENLIL automatically assesses an LEC by generating unique contexts of simulated environmental phenomena (via *evolutionary computation* [61]) to uncover examples that lead to each targeted behavior category. Fig. 5 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 is 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. Fig. 6 shows three behavior categories to represent different degrees of impact dust clouds can have on an object detector. Effectively, this information is used to assess the *trustworthiness* of the object detector.

B. 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.

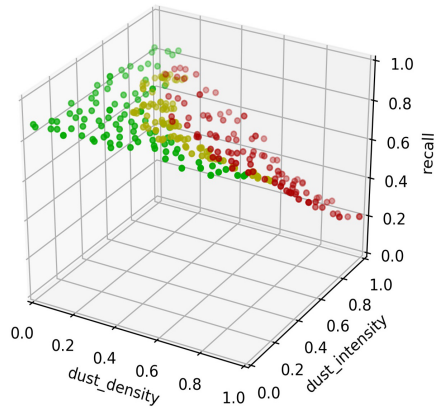
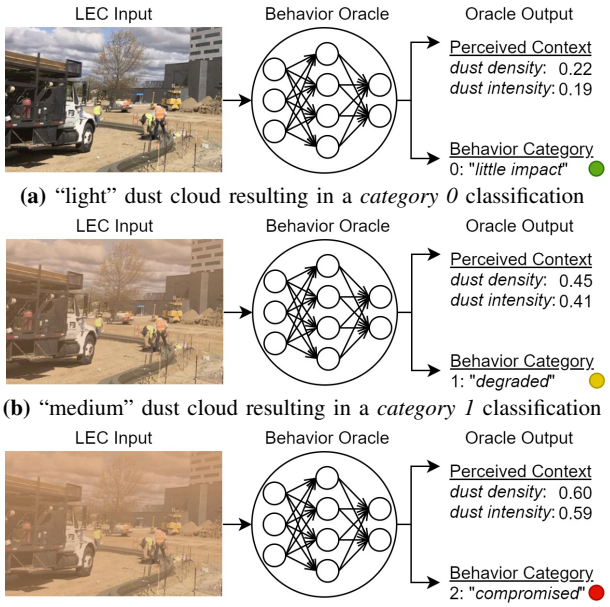


Fig. 5: Scatter plot of object detector recall when exposed to simulated dust clouds from ENLIL. Each point represents a different dust cloud context with the corresponding *density* and *intensity*. Green, yellow, and red points correspond to behavior categories 0, 1, and 2, respectively.



(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. 6: 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.

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

[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*) [60].

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.

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. 7 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.

```
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. 7: Logical expression parsed from KAOS model in Fig. 4.

Methods for adaptation are implemented as *adaptation tactics* [62], which are stored in a repository accessible by the MAPE-K loop (example tactic in Fig. 8). 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 *O1* is satisfied and goal *G17* is not satisfied, the tactic in Fig. 8 with a pre-condition matching the utility function for *O1* is selected. The post-condition in Fig. 8 includes the utility functions for *G17* and its sub-goals (*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.

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. 8: 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 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) [63].

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) [64]. 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) [65].

IV. Proof-of-Concept Demonstration

To illustrate the operation of MoDALAS, we consider a scenario where an autonomous rover is used within a construction site. 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 [66]. 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 past operation hours [67]. For such rovers, detecting and avoiding objects in the environment,

including pedestrians and other vehicles, are safety-critical requirements [68]. Increasingly, machine learning techniques have been used to provide object detection in such applications [69]. 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.

A. Rover Platform

Our rover, shown in Fig. 9, is a 1:5-scale vehicle based on a design published by Goldfain et al. [70]. 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 [71] simulation of the vehicle is used for offline testing.

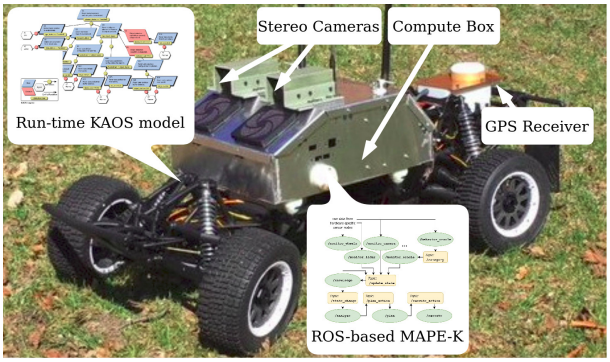


Fig. 9: A 1:5-scale (1.1m × 0.6m) autonomous vehicle for demonstration. A KAOS goal model governs run-time behavior and adaptation. A MAPE-K loop, integrated in the ROS infrastructure, identifies and acts on required adaptations.

ROS-Based Platform: The rover's software infrastructure is based on the Robot Operating System (ROS) [72], 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 delivers commands to actuators for throttle control, steering and braking.

ROS-Based Adaptation Manager: Fig. 10 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. 2. Data stores for goal models and adaptation tactics are populated at startup time and remain static during operation. However, 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. 2) 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`.

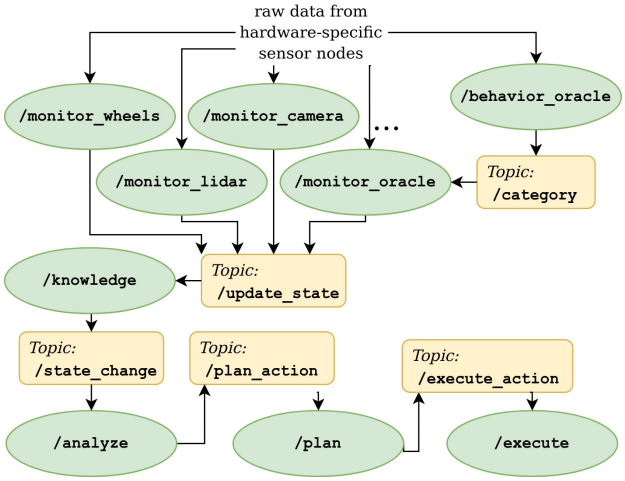


Fig. 10: Elided ROS graph for MAPE-K loop in rover software. ROS nodes shown as green ellipses and ROS topics as yellow boxes. Arrows indicate data flow.

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.

B. 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* [16] [18] and *triangulation* from stereo vision [73]. A three-dimensional *point cloud* [74] is generated by fusing stereo camera triangulations and lidar sensor readings. As shown in Fig. 10, 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 an adaptation.

The `/behavior_oracle` node processes camera images *online* with the behavior oracle DNN that was trained *offline* by ENLIL for model inference. (The DNN is initialized at startup time from a configuration file.) 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.

C. Run-Time Adaptation

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. Fig. 11 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 I, examples in columns (i), (ii), and (iii) are expected result in *Categories 0*, *1*, and *2*, respectively (i.e., "*little impact*," "*degraded*," and "*compromised*").

Scenario 1. Dust Clouds: 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. 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. 7)

(a) Unaltered Image of Construction Site

i. "little impact" ii. "degraded" iii. "compromised"

(b) Simulated Rainfall

i. "little impact" ii. "degraded" iii. "compromised"

(c) Simulated Dust Cloud (Haze)

i. "little impact" ii. "degraded" iii. "compromised"

(d) Simulated Lens Flare

Fig. 11: 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.

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. 8 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) [75] planner provided with ROS to compute trajectories around objects in the environment. The abstract actions in Fig. 8 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.

V. Related Work

This paper explores methods for the assurance of cyber-physical LESs via models at run time [76]. Related studies have investigated the verification of robotic systems [77], including construction site applications [78]. Those works apply formal methods for verification but do not explicitly address LECs faced with uncertain conditions. RoCS [79] has been introduced as a self-adaptive framework for robotic systems, but in contrast to MoDALAS, it is not model-driven 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 [64] developed a state-based modeling approach for model checking assurance properties of SASs. Weyns and Iftikhar [80] proposed the use of model-based simulation to evaluate system requirements and determine adaptation procedures. ENTRUST [81] supports the development of an SAS driven by GSN assurance cases and verified by probabilistic models at run time. Similarly, AC-ROS [82] 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 [83] [84]. Smith et al. [85] 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.

VI. 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 for LECs to generalize to complex environments, methods are needed to assess the capability of LECs at run time and adapt LESs to mitigate the use of LECs in unsafe 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. Future work will explore the inclusion of dynamic assurance cases as well as probabilistic and fuzzy logic evaluations of utility functions to expand the robustness and resilience of an LES [86].

Acknowledgments

We greatly appreciate contributions from Robert Jared Clark on our preliminary work. This work has been supported in part by a grant from the 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] J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds., *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*. Springer, 2013.

[2] Z. Langari and T. Maibaum, "Safety Cases: A Review of Challenges," in *Proc. 1st Int. Workshop on Assurance Cases for Software-Intensive Systems (ASSURE 2013)*. IEEE, 2013, p. 1–6.

[3] C. E. Tuncali, J. Kapinski, H. Ito, and J. V. Deshmukh, "Reasoning about Safety of Learning-Enabled Components in Autonomous Cyber-Physical Systems," in *Proc. 55th Annual Design Automation Conf. (DAC 2018)*. ACM, 2018.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT, 2016.

[5] K. Kawaguchi, L. P. Kaelbling, and Y. Bengio, "Generalization in Deep Learning," MIT, Tech. Rep., 2018. [Online]. Available: https://lis.csail.mit.edu/pubs/kawaguchi-techreport18.pdf

[6] F. Yu, Z. Qin, C. Liu, L. Zhao, Y. Wang, and X. Chen, "Interpreting and Evaluating Neural Network Robustness," in *Proc. 28th International Joint Conf. on Artificial Intelligence (IJCAI 2019)*. ijcai.org, 2019, pp. 4199–4205.

[7] W. Knight, "The Dark Secret at the Heart of AI," *MIT Technology Review*, vol. Artificial intelligence/Machine learning, p. 1, Apr 2017. [Online]. Available: https://www.technologyreview.com/2017/04/11/5113/the-dark-secret-at-the-heart-of-ai/

[8] J. Rushby, "The Interpretation and Evaluation of Assurance Cases," Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-15-01, Jul 2015. [Online]. Available: https://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf

[9] J. Schumann, P. Gupta, and Y. Liu, *Application of Neural Networks in High Assurance Systems: A Survey*, ser. Studies in Computational Intelligence (SCI). Springer, 2010, vol. 268.

[10] C. Hartsell, N. Mahadevan, S. Ramakrishna, A. Dubey, T. Bapty, T. Johnson, X. Koutsoukos, J. Sztipanovits, and G. Karsai, "Model-Based Design For CPS With Learning-Enabled Components," in *Proc. of the Workshop on Design Automation for CPS and IoT (DESTION 2019)*. ACM, 2019, pp. 1–9.

[11] A. van Lamsweerde and E. Letier, "From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering," in *Radical Innovations of Software and Systems Engineering in the Future. Lecture Notes in Computer Science (RISSEF 2002)*. Springer-Verlag, 2004, vol. 2941.

[12] J. O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," *IEEE Internet Computing*, vol. 11, no. 1, p. 40–48, Jan 2007.

[13] J. O. Kephart and D. M. Chess, "The Vision Of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[14] M. A. Langford and B. H. C. Cheng, ""Know What You Know": Predicting Behavior for Learning-Enabled Systems When Facing Uncertainty," in *Proc. 16th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021)*. ACM, 2021.

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[16] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A Survey of Deep Learning-Based Object Detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017.

[18] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep Learning for Generic Object Detection: A Survey," *Int J Comput Vis*, vol. 128, pp. 261–318, 2018.

[19] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A Survey of Deep Learning Applications to Autonomous Vehicle Control," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 2, pp. 712–733, 2021.

[20] W. Schwarting, J. Alonso-Mora, and D. Rus, "Planning and Decision-Making for Autonomous Vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, no. 1, pp. 187–210, 2018.

[21] J. Janai, F. Güney, A. Behl, and A. Geiger, "Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art," *CoRR*, vol. abs/1704.05519, 2017.

[22] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis, "A Survey of the State-of-the-Art Localization Techniques and Their Potentials for Autonomous Vehicle Applications," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 829–846, 2018.

[23] M. Borg, C. Englund, K. Wnuk, B. Duran, C. Levandowski, S. Gao, Y. Tan, H. Kaijser, H. Lönn, and J. Törnqvist, "Safely Entering the Deep: A Review of Verification and Validation for Machine Learning and a Challenge Elicitation in the Automotive Industry," *Journal of Automotive Software Engineering*, vol. 1, pp. 1–19, 2019.

[24] G. Calikli and A. Bener, "Empirical Analyses of the Factors Affecting Confirmation Bias and the Effects of Confirmation Bias on Software Developer/Tester Performance," in *Proc. 6th Int. Conf. on Predictive Models in Software Engineering (PROMISE 2010)*. ACM, 2010.

[25] S. E. Whang and J.-G. Lee, "Data Collection and Quality Challenges for Deep Learning," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3429–3432, Aug 2020.

[26] J. Jo and Y. Bengio, "Measuring the Tendency of CNNs to Learn Surface Statistical Regularities," *CoRR*, vol. abs/1711.11561, 2017.

[27] Y. Bengio, "Priors For Deep Learning of Semantic Representations," Keynote at ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS), 2020.

[28] A. Barredo Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, and F. Herrera, "Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges Toward Responsible AI," *Information Fusion*, vol. 58, pp. 82–115, Jun 2020.

[29] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing Properties of Neural Networks," *CoRR*, vol. abs/1312.6199, 2013, appeared in ICLR 2014.

[30] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *CoRR*, vol. abs/1412.6572, 2014, appeared in ICLR 2015.

[31] A. Nguyen, J. Yosinski, and J. Clune, "Deep Neural Networks Are Easily Fooled: High Confidence Predictions For Unrecognizable Images," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2015)*. IEEE, 2015, pp. 427–436.

[32] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," in *Proc. 26th Symposium on Operating Systems Principles (SOSP 2017)*. ACM, 2017, p. 1–18.

[33] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *Proc. 40th Int. Conf. on Software Engineering (ICSE 2018)*. ACM, 2018, p. 303–314.

[34] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, and Y. Liu, "DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems," in *Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018)*. ACM, 2018, p. 120–131.

[35] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems," in *Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018)*. ACM, 2018, p. 132–142.

[36] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "DeepConcolic: Testing and Debugging Deep Neural Networks," in *Proc. 41st Int. Conf. on Software Engineering (ICSE 2019)*. IEEE, 2019, p. 111–114.

[37] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks," in *Proc. of the 28th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*. ACM, 2019, p. 146–157.

[38] M. A. Langford and B. H. C. Cheng, "Enhancing Learning-Enabled Software Systems to Address Environmental Uncertainty," in *Proc. 16th IEEE Int. Conf. on Autonomic Computing (ICAC 2019)*. IEEE, 2019, pp. 115–124.

[39] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing," in *Proc. of the 36th Int. Conf. on Machine Learning (PMLR 2019)*, 2019, pp. 4901–4911.

[40] D. Berend, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao, "Cats Are Not Fish: Deep Learning Testing Calls for Out-Of-Distribution Awareness," in *Proc. of the 35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2020)*. ACM, 2020, p. 1041–1052.

[41] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test Selection for Deep Learning Systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan 2021.

[42] M. Naeem Irfan, C. Oriat, and R. Groz, "Model Inference and Testing," *Advances in Computers*, vol. 89, pp. 89–139, 2013.

[43] G. Fraser and N. Walkinshaw, "Assessing and Generating Test Sets in Terms of Behavioural Adequacy," *Softw. Test. Verif. Reliab.*, vol. 25, no. 8, p. 749–780, Dec 2015.

[44] P. Papadopoulos and N. Walkinshaw, "Black-Box Test Generation from Inferred Models," in *Proc. 4th Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2015)*. IEEE, 2015, p. 19–24.

[45] B. K. Aichernig, R. Bloem, M. Ebrahimi, M. Horn, F. Pernkopf, W. Roth, A. Rupp, M. Tappler, and M. Tranninger, "Learning a Behavior Model of Hybrid Systems Through Combining Model-Based Testing and Machine Learning," in *Testing Software and Systems. Lecture Notes in Computer Science (ICTSS 2019)*, C. Gaston, N. Kosmatov, and P. Le Gall, Eds. Springer, 2019, vol. 11812.

[46] N. Jacobi, P. Husbands, and I. Harvey, "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics," in *Proc. 3rd European Conf. on Advances in Artificial Life*. Springer-Verlag, 1995, p. 704–720.

[47] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher, M. Shahzad, W. Yang, R. Bamler, and X. X. Zhu, "A Survey of Uncertainty in Deep Neural Networks," *CoRR*, vol. abs/2107.03342, 2021.

[48] I. Cortés-Ciriano and A. Bender, "Deep Confidence: A Computationally Efficient Framework for Calculating Reliable Prediction Errors for Deep Neural Networks," *Journal of Chemical Information and Modeling*, vol. 59, no. 3, p. 1269–1281, Oct 2018.

[49] Object Management Group, "Structured Assurance Case Metamodel (SACM) Version 2.1," OMG, Tech. Rep., 2020. [Online]. Available: https://www.omg.org/spec/SACM

[50] J. Goodenough, C. Weinstock, and A. Klein, "Toward a Theory of Assurance Case Confidence," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2012-TR-002, 2012. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067

[51] Assurance Case Working Group, "Goal Structuring Notation Community Standard Version 2," SCSC, Tech. Rep., 2018. [Online]. Available: https://scsc.uk/r141B:1

[52] A. Lapouchnian, "Goal-Oriented Requirements Engineering: An Overview of the Current Research," University of Toronto, Tech. Rep., 2005. [Online]. Available: http://www.cs.utoronto.ca/~alexei/pub/Lapouchnian-Depth.pdf

[53] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*. Springer, 2009, pp. 48–70.

[54] IBM, "An Architectural Blueprint For Autonomic Computing," IBM, Tech. Rep. 3rd ed., June 2005. [Online]. Available: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[55] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation," in *Proc. 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*. ACM, 2015, pp. 13–23.

[56] S.-W. Cheng, "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation," Ph.D. dissertation, Carnegie Mellon University, 2008.

[57] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility Functions in Autonomic Systems," in *Proc. Int. Conf. on Autonomic Computing (ICAC 2004)*. IEEE, 2004, pp. 70–77.

[58] P. deGrandis and G. Valetto, "Elicitation and Utilization of Application-Level Utility Functions," in *Proc. 6th Int. Conf. on Autonomic Computing (ICAC 2009)*. ACM, 2009, p. 107–116.

[59] E. M. Clarke, Jr., O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*, 2nd ed. MIT Press, 2018.

[60] M. Sokolova and G. Lapalme, "A Systematic Analysis of Performance Measures for Classification Tasks," *Information Processing & Management*, vol. 45, pp. 427–437, Jul 2009.

[61] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer-Verlag, 2015.

[62] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-Based Self-Adaptation in the Presence of Multiple Objectives," in *Proc. Int. Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2006)*. ACM, 2006, p. 2–8.

[63] J. Palmerino, Q. Yu, T. Desell, and D. Krutz, "Improving the Decision-Making Process of Self-Adaptive Systems by Accounting for Tactic Volatility," in *Proc. 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2019)*, 2019, pp. 949–961.

[64] J. Zhang and B. H. C. Cheng, "Model-Based Development of Dynamically Adaptive Software," in *Proc. 28th Int. Conf. on Software Engineering (ICSE 2006)*. ACM, 2006, p. 371–380.

[65] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, p. 1293–1306, Nov 1990.

[66] N. Melenbrink, J. Werfel, and A. Menges, "On-Site Autonomous Construction Robots: Towards Unsupervised Building," *Automation in Construction*, vol. 119, 2020.

[67] D. Malone, "Rovers Set to Invade Construction Jobsites," HorizonTV, 2019. [Online]. Available: https://www.bdcnetwork.com/rovers-set-invade-construction-jobsites

[68] D. Weyns, T. Holvoet, K. Schelfthout, and J. Wielemans, "Decentralized Control of Automatic Guided Vehicles: Applying Multi-Agent Systems in Practice," in *Companion to the 23rd ACM SIGPLAN Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA Companion 2008)*. ACM, 2008, p. 663–674.

[69] S. Dersten, P. Wallin, J. Fröberg, and J. Axelsson, "Analysis of the Information Needs of an Autonomous Hauler in a Quarry Site," in *Proc. of the 11th System of Systems Engineering Conf. (SoSE 2016)*, 2016, pp. 1–6.

[70] B. Goldfain, P. Drews, C. You, M. Barulic, O. Velev, P. Tsiotras, and J. M. Rehg, "AutoRally: An Open Platform for Aggressive Autonomous Driving," *IEEE Control Systems Magazine*, vol. 39, no. 1, pp. 26–55, 2019.

[71] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, an Open-source Multi-robot Simulator," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, vol. 3, 2004, pp. 2149–2154.

[72] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: An Open-Source Robot Operating System," in *Proc. Int. Conf. on Robotics and Automation Workshop on Open Source Software (ICRA Workshop 2009)*. IEEE, 2009.

[73] R. I. Hartley and P. Sturm, "Triangulation," *Comput. Vis. Image Underst.*, vol. 68, no. 2, p. 146–157, 1997.

[74] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA 2011)*. IEEE, 2011, pp. 1–4.

[75] "Obstacle Avoidance and Robot Footprint Model," ROS.org, 2019. [Online]. Available: http://wiki.ros.org/teb_local_planner/Tutorials/Obstacle%20Avoidance%20and%20Robot%20Footprint%20Model

[76] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos, "Runtime Goal Models: Keynote," in *Proceedings of 7th IEEE Int. Conf. on Research Challenges in Information Science (RCIS 2013)*. IEEE, 2013, pp. 1–11.

[77] W. Huang, Y. Zhou, Y. Sun, A. Banks, J. Meng, J. Sharp, S. Maskell, and X. Huang, "Formal Verification of Robustness and Resilience of Learning-Enabled State Estimation Systems for Robotics," *CoRR*, vol. abs/2010.08311, 2020.

[78] R. Gu, R. Marinescu, C. Seceleanu, and K. Lundqvist, "Formal Verification of an Autonomous Wheel Loader by Model Checking," in *Proc. of the 6th Conf. on Formal Methods in Software Engineering (FormaliSE 2018)*. ACM, 2018, p. 74–83.

[79] L. Ramos, G. Divino, G. Lopes, B. de França, L. Montecchi, and E. Colombini, "The RoCS Framework to Support the Development of Autonomous Robots," *Journal of Software Engineering Research and Development*, vol. 7, pp. 10:1–10:14, 2019.

[80] D. Weyns and M. U. Iftikhar, "Model-Based Simulation at Runtime for Self-Adaptive Systems," in *Proc. 15th Int. Conf. on Autonomic Computing (ICAC 2016)*, 2016, pp. 364–373.

[81] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1039–1069, 2018.

[82] B. H. C. Cheng, R. J. Clark, J. E. Fleck, M. A. Langford, and P. K. McKinley, "AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System," in *Proc. 23rd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2020)*. ACM, 2020.

[83] Qinbao Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 69–82, 2006.

[84] D. Rodriguez, R. Ruiz, J. C. Riquelme, and R. Harrison, "A Study of Subgroup Discovery Approaches For Defect Prediction," *Information and Software Technology*, vol. 55, no. 10, pp. 1810–1822, 2013.

[85] C. Smith, E. Denney, and G. Pai, "Hazard Contribution Modes of Machine Learning Components," OSTI, Tech. Rep., 2020, (AAAI Workshop: SafeAI 2020). [Online]. Available: https://www.osti.gov/biblio/1606667

[86] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems," in *Proc. 17th IEEE Int. Requirements Engineering Conf. (RE 2009)*. IEEE Computer Society, 2009, pp. 79–88.