Automatic Derivation of Utility Functions for Monitoring Software Requirements[*]

Andres J. Ramirez and Betty H.C. Cheng

Michigan State University
Department of Computer Science and Engineering
3115 Engineering Building
East Lansing, MI 48824
{ramir105,chengb}@cse.msu.edu

Abstract. Utility functions can be used to monitor requirements of a dynamically adaptive system (DAS). More specifically, a utility function maps monitoring information to a scalar value proportional to how well a requirement is satisfied. Utility functions may be manually elicited by requirements engineers, or indirectly inferred through statistical regression techniques. This paper presents a goal-based requirements model-driven approach for automatically deriving state-, metric-, and fuzzy logic-based utility functions for RELAXed goal models. State- and fuzzy logic-based utility functions are responsible for detecting requirements violations, and metric-based utility functions are used to detect conditions conducive to a requirements violation. We demonstrate the proposed approach by applying it to the goal model of an intelligent vehicle system (IVS) and use the derived utility functions to monitor the IVS under different environmental conditions at run time.

1 Introduction

A dynamically adaptive system (DAS) monitors itself and its execution environment to assess how well it satisfies requirements at run time. This monitoring information enables a DAS to detect both requirements violations, as well as conditions conducive to their occurrence [7,8,16]. Utility functions have been successfully applied for self-assessment purposes in DASs [3,9,15,17]. Within the context of a DAS, a utility function maps monitoring data to a scalar value, typically within the ranges of zero and one, that is proportional to how well the DAS satisfies its requirements at run time. This paper presents a goal-based model-driven approach for automatically deriving utility functions during the requirements engineering phase. The set of derived utility functions enable a

[*] This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, DBI-0939454, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Army, Ford, or other research sponsors.

DAS to monitor the satisfaction of requirements at run time, as well as identify potential sources of obstacles that may impede a goal's satisfaction.

Utility functions provide a light-weight technique for associating the actions taken by a decision-making process with a DAS's high-level goals, concerns, and requirements [9,17]. As such, utility functions can be used to monitor both functional and non-functional requirements of a DAS. Utility functions for monitoring the functional behavior of a DAS are often derived manually by requirements engineers with the aid of domain experts. In contrast, utility functions for monitoring the non-functional requirements (e.g., performance) of a DAS may be indirectly inferred at run time through statistical regression-based techniques [1,9]. These *performance-based* utility functions often generate a single, application-level utility value representative of the overall system's performance. Deviations from this utility value suggests an anomalous behavior that may require an adaptation. While these approaches facilitate the derivation of performance-based utility functions, they tend to postpone their integration until deployment when real execution data becomes available to drive the regression process.

This paper presents Athena, an approach that leverages goal-based models to facilitate the automatic derivation of utility functions at the requirements level. Currently, Athena supports the automatic derivation of state-, metric-, and fuzzy logic-based utility functions for KAOS models [5,12] that include RELAXed goals [2,18]. State-based utility functions assess whether a DAS satisfies functional invariant goals. Metric-based utility functions, on the other hand, detect conditions conducive to a requirements violation, ideally enabling a DAS to mitigate such conditions before an invariant goal becomes violated. Lastly, fuzzy logic-based utility functions compute the satisfaction of non-invariant goals that have been RELAXed in order to explicitly account for the effects of environmental uncertainty. Derived utility functions enable a DAS not only to monitor requirements at run time, but also identify candidate sets of system and environmental agents that may be responsible for a requirements violation.

Athena accepts as input a goal model and a mapping between environmental conditions and the monitoring elements responsible for observing them, and generates utility functions to be used for requirements monitoring. To generate a utility function, Athena uses these mappings to identify observable conditions of the system-to-be and its execution environment specified in a goal's definition. Next, Athena maps keywords in a goal's definition to different types of utility function templates. For each invariant goal, Athena generates a state-based utility function that returns true or false depending on the satisfaction of the goal. For a non-invariant goal, Athena generates a metric-based utility function to measure the degree to which some observable condition in the goal's definition is minimized or maximized with regards to a given threshold. Lastly, for a RELAXed goal, Athena generates a fuzzy logic-based utility function by mapping RELAX operators to their corresponding fuzzy logic-based mathematical functions [18]. At run time, derived utility functions accept monitoring data from the environmental agents (i.e., sensors) in order to detect requirements violations and conditions conducive to their occurrence.

We demonstrate Athena by applying it to a goal model we constructed to capture the objectives, constraints, and requirements of an Intelligent Vehicle System (IVS) application that must perform adaptive cruise control and lane keeping while avoiding collisions with other vehicles on the road. Based on this goal model, Athena generated utility functions to assess how well the IVS satisfies its requirements at run time. Lastly, we implemented the set of derived utility functions within a prototype of the IVS in the Webots simulation platform [14] to enable the monitoring of requirements during simulation runs.The remainder of this paper is organized as follows. In Section 2 we present background material on goal-oriented requirements modeling and the RELAX language. Next, in Section 3, we introduce the IVS application domain and use it to present the proposed approach. Section 4 presents our case study. We then provide an overview of related work in Section 5. Lastly, Section 6 discusses Athena, summarizes main findings, and presents future directions.

2 Background

This section presents background material on goal-based requirements modeling, and the RELAX requirements specification language.

2.1 Goal-Based Requirements Modeling

From the perspective of a stakeholder, a goal specifies the objectives that the system-to-be and its execution environment must satisfy at run time [12]. While the system-to-be must always satisfy invariant goals, it may temporarily allow the dissatisfaction of non-invariant goals. In general, goals may be classified across two orthogonal dimensions. A goal may be classified either as functional or non-functional depending on whether it specifies *what* services the system-to-be must provide or whether it constrains *how* such services must be provided, respectively. In addition, a goal may also be classified either as a hard or soft goal. The satisfaction of a *hard* goal can be determined in a crisp manner, usually through state-based predicates. In contrast, a *soft* goal may be measured, to some degree, through user-defined metrics, though their ultimate satisfaction may not be precisely determined due to subjective, and potentially conflicting, preferences by various stakeholders. Since the satisfaction of a soft goal cannot be absolutely determined, it is often said that a soft goal is *satisficed* [4]. Jureta *et al.* [11] extended these concepts of achieving a hard goal and satisficing a soft goal with the notion of *excelling*, where a goal may be constantly improved upon some measurable dimension (e.g., minimize vehicle acceleration rate).

A key objective in goal-based analysis is to systematically decompose high-level goals into finer-grained goals. To this end, goals are graphically represented in an acyclic directed graph where a goal may be decomposed into subgoals through AND/OR refinements. While a goal that has been AND-decomposed may only be satisfied if all of its subgoals are satisfied, a goal that has been OR-decomposed is satisfied if at least one of its subgoals is satisfied. This goal decomposition process terminates once each goal is assigned to a single system or environmental agent. Whereas a goal under the assignment of an agent in the system-to-be is a requirement, a goal under the assignment of an environmental agent is an *expectation* of the environment [12]. Darimont and van Lamsweerde [6] developed a set of goal refinement patterns to guide requirements engineers through the process of decomposing higher-level goals into finer-grained goals. Each refinement pattern is proven correct, thereby enabling a requirements engineer to instantiate them and leverage the underlying theoretical framework without having to prove their correctness again.

2.2 RELAX Specification Language

RELAX [2,18] is a requirements specification language for identifying, evaluating, and mitigating sources of environmental uncertainty in a DAS. RELAX focuses on declaratively specifying the sources and impacts of uncertainty at the shared boundary between the system-to-be and its execution environment [10]. This information is organized into ENV, MON, and REL elements. In particular, ENV specifies environmental properties that may or may not be directly observable by a DAS; MON specifies the elements that make up the DAS's monitoring infrastructure; and REL defines how to compute the values of ENV properties from MON elements. The semantics of RELAX operators have been defined in terms of fuzzy logic to constrain the extent to which a non-invariant requirement may become temporarily unsatisfied [18]. For example, the RELAXed goal "*Achieve [VehicleSpeed AS CLOSE AS POSSIBLE TO DesiredSpeed]*" specifies that while the value of *VehicleSpeed* should approximate the *DesiredSpeed* threshold value, minor deviations between the two values, as specified by a corresponding fuzzy logic operator, are tolerable at run time.

Previously, Cheng *et al.* [2] presented an approach for applying the RELAX process to non-invariant goals in a KAOS model [5,12] where uncertainty may cause a goal to become temporarily unsatisfied. To apply their approach, a requirements engineer informally specifies ENV, MON, and REL elements. This paper automates the derivation of utility functions for KAOS models with RELAXed goals where the definition of ENV, MON, and REL are specified more formally and amenable to automated processing.

3 Athena Approach

This section presents a goal-based, requirements model-driven approach for automatically deriving utility functions. First, we introduce the intelligent vehicle system (IVS) application domain and present a goal model that captures its requirements, including several goals that have been RELAXed when environmental uncertainty is an issue. We then use the IVS goal model as an example to present and describe the steps of Athena in detail.

3.1 Intelligent Vehicle System

Intelligent transportation systems (ITS) will provide safe and efficient transportation of passengers across roadways. Within the ITS domain, an intelligent vehicle system (IVS) provides autonomous vehicle control through a combination of adaptive cruise control (ACC), lane keeping, and collision avoidance features. As Figure 1 illustrates, the ACC module is responsible for maintaining a *SafeDistance* between the IVS and obstacles in front of the IVS, such as Lead Vehicle. Specifically, the ACC module commands the vehicle's engine to maintain a *SafeSpeed* and keep the IVS within the *CoastingZone*. The lane keeping module, on the other hand, detects roadway markings and keeps the IVS within the center of the driving lane. Lastly, the collision avoidance module uses cameras and distance sensors to detect obstacles and adjust the vehicle's engine and steering mechanisms in response.

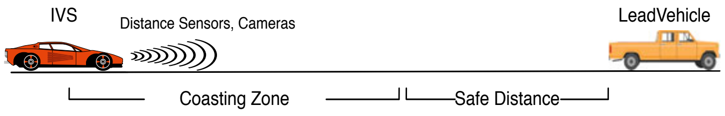


Fig. 1. Intelligent Vehicle System

The Webots simulation platform [14] provides a generic implementation of an IVS capable of cruise control and lane keeping. This generic IVS model comprises a GPS unit for computing the vehicle's velocity, a camera for detecting roadway markings, and an accelerometer to compute acceleration and deceleration rates. For this study, we extended the basic Webots IVS implementation with a monitoring infrastructure that supports ACC, lane keeping, and collision avoidance. The extended IVS also includes a compass and a gyroscope to compute changes in vehicle heading and velocity, three additional cameras to detect roadway markings and obstacles, and ten laser- and sonar-based distance sensors that measure the distance between the IVS and nearby obstacles. For the remainder of this paper, IVS refers to the extended IVS implementation.

Figure 2 shows an elided RELAX goal model for the IVS application. This model captures goals for computing the current speed of the IVS, as well as its distance to nearby obstacles. The IVS can use either a GPS unit or wheel sensors to compute its velocity, and either cameras or distance sensors to compute its distance to nearby obstacles. These alternative refinements enable the IVS to change how it senses its environment at run time. Several non-invariant goals were RELAXed in this goal model, where the RELAX operators are in uppercase, to explicitly account for the effects of uncertainty in achieving a goal. For instance, goal (C) was RELAXed since the IVS can tolerate minor differences between *VehicleSpeed* and *DesiredSpeed*. However, goal (D) was not RELAXed as deviations between *VehicleSpeed* and *SafeSpeed* may cause a collision. Lastly, to capture the interactions between system and environmental agents (denoted by a stick figure in an agent hexagon), we applied the *unmonitorability* refinement pattern [6] to goals (I,L,M) (J,N,O), and (K,P,Q). This refinement pattern was applied because system agents alone were not capable of monitoring the conditions formulated in goals (I,J,K). Instead, by applying this refinement, system

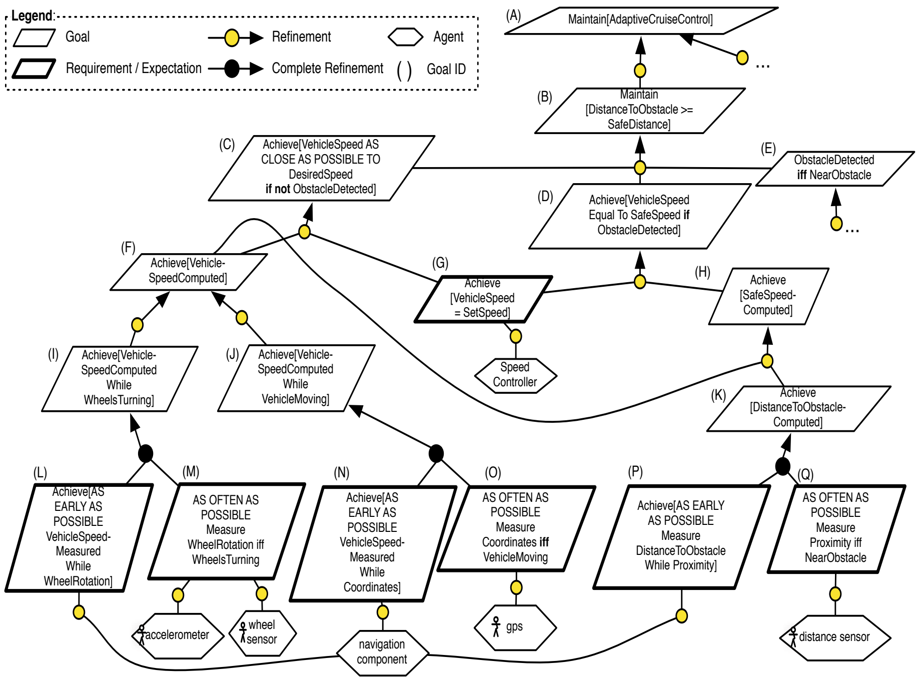

Legend:
Goal
Requirement / Expectation
Refinement
Complete Refinement
Agent
( ) Goal ID
(A) Maintain[AdaptiveCruiseControl]
(B) Maintain [DistanceToObstacle >= SafeDistance]
(C) Achieve[VehicleSpeed AS CLOSE AS POSSIBLE TO DesiredSpeed if not ObstacleDetected]
(D) Achieve[VehicleSpeed Equal To SafeSpeed if ObstacleDetected]
(E) ObstacleDetected iff NearObstacle
(F) Achieve[Vehicle-SpeedComputed]
(G) Achieve [VehicleSpeed = SetSpeed]
(H) Achieve [SafeSpeed-Computed]
(I) Achieve[Vehicle-SpeedComputed While WheelsTurning]
(J) Achieve[Vehicle-SpeedComputed While VehicleMoving]
(K) Achieve [DistanceToObstacle-Computed]
(L) Achieve[AS EARLY AS POSSIBLE VehicleSpeed-Measured While WheelRotation]
(M) AS OFTEN AS POSSIBLE Measure WheelRotation iff WheelsTurning
(N) Achieve[AS EARLY AS POSSIBLE VehicleSpeed-Measured While Coordinates]
(O) AS OFTEN AS POSSIBLE Measure Coordinates iff VehicleMoving
(P) Achieve[AS EARLY AS POSSIBLE Measure DistanceToObstacle While Proximity]
(Q) AS OFTEN AS POSSIBLE Measure Proximity iff NearObstacle
Speed Controller
accelerometer
wheel sensor
navigation component
gps
distance sensor

Fig. 2. Goal model for adaptive cruise control in IVS, including RELAXed goals

agents process the environmental conditions measured by an environmental agent. Due to space constraints we only show the ACC goal model and not its lane keeping counterpart.

3.2 Description of Athena Approach

To use Athena, a requirements engineer must first follow the RELAX goal modeling approach [2] and construct a goal model, specify invariant goals, and ENV, MON, and REL elements for each requirement. ENV specifies environmental conditions observable by the DAS; MON specifies environmental agents (i.e., sensors) that make up the DAS's monitoring infrastructure; and REL specifies how to compute the values of ENV properties from MON elements. Table 1 presents a subset of ENV, MON, and REL elements for the IVS application. For example, row (1) specifies that the IVS obtains the value of the *WheelRotation* ENV property directly from its *WheelSensor* MON element. In contrast, row (2) specifies that the IVS is unable to directly compute the value of the ENV property *VehicleSpeed*. Instead, the IVS computes the value of this property from the wheel's dimensions and rotation rate, as specified in the REL relationship for that row.

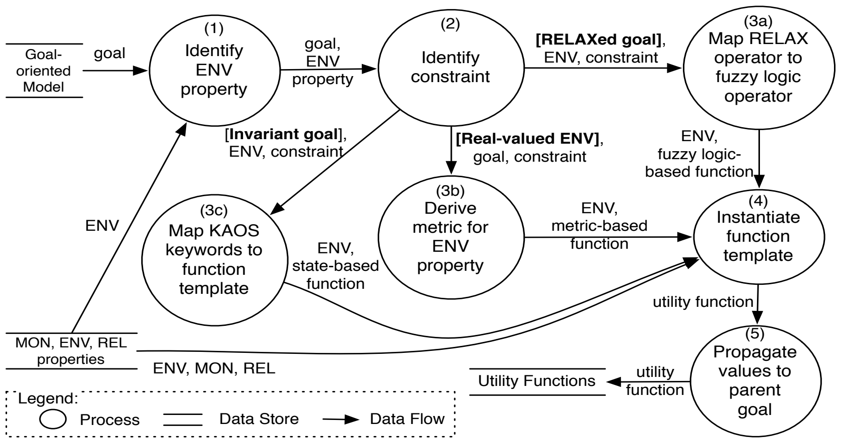
Athena accepts as input a goal model and set of ENV, MON, and REL elements, and produces as output a set of state-, metric-, and fuzzy logic-based utility functions by instantiating functions templates based on the goal's type. As with

Table 1. Table with ENV, MON, and REL elements for IVS application

Row	Goal	ENV	MON	REL
1	M	WheelRotation	WheelSensor	WheelSensor.value
2	L	VehicleSpeed		VehicleSpeed = IVS.wheel_diameter * 3.1415 * WheelRotation
3	O	Coordinates	GPS	Coordinates = GPS.value
4	N	VehicleSpeed		VehicleSpeed = (NavigationComponent.prev_pos - Coordinates) / GPS.time_unit
5	Q	Proximity	DistanceSensor	Proximity = DistanceSensor.value
6	P	DistanceToObstacle		DistanceToObstacle = Proximity * DistanceSensor.max_range
7	H	SafeSpeed	WheelSensor, GPS, DistanceSensor	SafeSpeed = VehicleSpeed - 0.1 * VehicleSpeed * (1.0 - SafeDistance / DistanceToObstacle)
8	E	ObstacleDetected	DistanceSensor	ObstacleDetected = Proximity > 0.95
9	B	SafeDistance	WheelSensor, GPS	SafeDistance = 2.5 * VehicleSpeed * 1000 / 3600

traditional requirements monitoring approaches, Athena generates state-based functions to monitor functional invariant goals since their *satisfaction* can be absolutely determined. In contrast, Athena generates metric- and fuzzy logic-based utility functions to monitor the *satisficement* [4] of non-invariant goals whose satisfaction may not be precisely determined, ideally enabling a DAS to detect and mitigate conditions that would otherwise lead to the violation of an invariant goal. Once implemented within a DAS, these utility functions compute utility values at run time based on available monitoring data, thereby enabling a DAS to monitor requirements, detect conditions conducive to a requirements violation, and facilitate the identification of potential goal obstructions.

The data flow diagram in Figure 3 illustrates the bottom-up approach that Athena applies to *automatically* generate utility functions starting at the leaf goals and progressing towards the root goal. We now describe each of the key steps that Athena automatically applies:



Goal-oriented Model
goal
(1) Identify ENV property
goal, ENV property
(2) Identify constraint
[RELAXed goal], ENV, constraint
(3a) Map RELAX operator to fuzzy logic operator
[Invariant goal], ENV, constraint
[Real-valued ENV], goal, constraint
ENV, fuzzy logic-based function
ENV
(3c) Map KAOS keywords to function template
(3b) Derive metric for ENV property
ENV, metric-based function
(4) Instantiate function template
ENV, state-based function
utility function
MON, ENV, REL properties
ENV, MON, REL
(5) Propagate values to parent goal
Utility Functions
utility function
Legend:
Process
Data Store
Data Flow

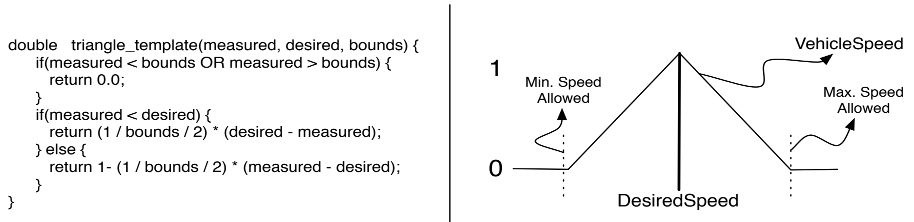
Fig. 3. Data flow diagram describing the approach

(1) Identify ENV Property. ENV specifies observable conditions of the execution environment, and can thus be observed by the DAS through its monitoring infrastructure. Athena matches text elements in a goal's specification with the set of ENV properties (see Table 1). For example, goal (B) refers to the ENV property *DistanceToObstacle* specified in row 6 of Table 1. Not all goals, however, refer to an ENV property. For instance, goal (F) does not refer to an ENV property (i.e., *VehicleSpeedComputed* is not specified in Table 1). If a goal does not refer to an ENV property, then Athena proceeds to step (5).

(2) Identify Constraints on the Goal. Constraints are often logical conditions or thresholds that can be evaluated in a crisp fashion (i.e., true or false). A goal may specify either an *absolute* constraint (i.e., a fixed threshold), or a *relative* constraint that specifies a relationship between properties whose value may change, such as an ENV property. For instance, goal (B) specifies a relative constraint/threshold, *SafeDistance*, whose value depends upon the IVS's current speed that is observable and controllable by the system. If a goal does not specify a constraint or threshold, then Athena proceeds to step (5).

(3a) Map RELAX Operator to Fuzzy Logic-Based Utility Function. RELAX defines a set of operators to constrain how a non-invariant goal may become temporarily unsatisfied due to environmental uncertainty [2,18]. Each operator is associated with a fuzzy logic-based function that evaluates the degree to which a non-invariant goal is satisfied. Athena generates a fuzzy logic-based utility function by mapping a RELAX operator to its corresponding fuzzy logic operator. For instance, Figure 4(A) presents a triangle-shaped function template for the RELAX operator *MeasuredQuantity AS CLOSE AS POSSIBLE TO DesiredQuantity.* This utility function returns a value between 0 and 1 proportional to how much *MeasuredQuantity* approaches *DesiredQuantity.*

Continuing with this example, Figure 4(B) illustrates how this fuzzy logic-based utility function template was applied to goal (C). In particular, the measured ENV property, *VehicleSpeed*, is mapped to a triangular shape that is centered at the ENV property's constraint (i.e., *DesiredSpeed*). This RELAXed goal specifies that the IVS tolerates temporary deviations between *VehicleSpeed* and *DesiredSpeed* within the minimum and maximum bounds allowed.


```
double   triangle_template(measured, desired, bounds) {
    if(measured < bounds OR measured > bounds) {
      return 0.0;
    }
    if(measured < desired) {
      return (1 / bounds / 2) * (desired - measured);
    } else {
      return 1- (1 / bounds / 2) * (measured - desired);
    }
}
```

(A) Fuzzy Logic Utility Function Template

(B) Fuzzy Logic Utility Function Instance

Fig. 4. Example function template for RELAX operator

(3b) Derive a Metric for a Real-Valued ENV Property. Athena generates a metric-based utility function for measuring the *satisficement* of invariant and non-invariant goals. While it is possible to determine whether an invariant goal is *satisfied*, Athena *also* evaluates the degree to which an invariant goal is satisficed or *excelled* [11], thereby enabling a DAS to detect and mitigate conditions conducive to a requirements violation. To this end, Athena leverages a goal's fitness criterion, which is an annotation often associated with soft goals to quantify the extent to which a goal should be met [12]. Specifically, Athena maps keywords in this annotation (i.e., minimize/maximize some condition) to a function template that either minimizes or maximizes the divergence between an ENV property and its constraint or threshold. For instance, the following function template measures the degree to which an ENV property approaches a given constraint, $\mathrm{Val}_{constraint}$:

$$\mathrm{UT}_{\mathrm{minimize}} = 1 - \min\left\{ \frac{|\mathrm{Val}_{ENV} - \mathrm{Val}_{Constraint}|}{\mathrm{Val}_{ENV}}, 1 \right\} \tag{1}$$

As such, Athena generates a utility function that measures the degree to which the IVS minimizes the difference between *VehicleSpeed* and *SafeSpeed*. Function template (1) can be instantiated as follows:

$$\mathrm{UT}_{\mathrm{SafeSpeed}} = 1 - \min\left\{ \frac{|\mathrm{VehicleSpeed} - \mathrm{SafeSpeed}|}{\mathrm{VehicleSpeed}}, 1 \right\} \tag{2}$$

This utility function produces a utility value inversely proportional to the difference between *VehicleSpeed* and *SafeSpeed*. A sharp drop in the utility values produced by this utility function may suggest the IVS is exceeding its *SafeSpeed* constraint, which may lead to a collision with an obstacle.

(3c) Derive State-Based Function for an Invariant Goal. An invariant goal describes a functionality that the system-to-be must *always* provide. To specify an invariant goal, a requirements engineer uses a set of KAOS [5,12] keywords (i.e., *Maintain*, *Achieve*, *Avoid*) that can be mapped to precise semantics in temporal state-based logic [12]. Athena maps these keywords to a state-based utility function template that returns true or false depending on whether the constraint is satisfied. For instance, Figure 5(A) presents a state-based utility function template that is used to monitor the satisfaction of *Maintain* goals, where *ENV* refers to the environmental condition identified in step (1), *Op* refers to a logical operator (i.e., <, =, etc.), and *Constraint* refers to the goal's constraint identified in step (2). This template uses a *satisfied* guard to preserve the semantics of a *Maintain* goal and thus returns true only if the constraint has always been satisfied. As an example, Figure 5(B) shows how this template was instantiated for goal (B), where the utility function returns true as long as the IVS has never crossed the *SafeDistance* threshold.

```
boolean   maintain_template(ENV, Op, Constraint) {
    if(satisfied) {
      return (satisfied = Op(Env, Constrain));
    }
    return false;
}
```

(A) general state-based utility function

```
boolean   maintain_template(DistanceToObstacle, <=, SafeDistance) {
    if(satisfied) {
      return (satisfied = DistanceToObstacle <= SafeDistance);
    }
    return false;
}
```

(B) instance of state-based utility function

Fig. 5. Example state-based function template for *Maintain* goals

(4) Instantiate Function Template. Athena leverages the set of ENV properties, MON elements, and REL relationships (See Table 1) to express each utility function solely in terms of MON elements. These MON elements provide the monitoring information that each utility function needs to assess the satisfaction of a goal at run time. For example, Athena replaces the ENV property term *VehicleSpeed* in the utility functions derived for goal (D) with the following expression:

$$\text{VehicleSpeed} = \text{IVS.wheel\_diameter} * 3.1415 * \text{WheelSensor.value} \tag{3}$$

This expression, shown in row (2) of Table 1, specifies that the value of *VehicleSpeed* can be computed based on the wheel's geometry and its rotation rate, which is measured by *WheelSensor*.

(5) Propagate Utility Values to Parent Goals. Athena propagates the utility values associated with a goal (if any) to its parent goal in order to detect conditions conducive to a requirements violation. To a parent goal, this propagated utility value measures how well its subgoals are satisficed. The utility values of multiple subgoals are combined in different ways depending on the type of goal refinement applied. For an AND-decomposition, Athena computes the product of each utility value reported by the subgoals in the refinement. For instance, if the utility value associated with goals (L) and (M) are 0.8 and 1.0, respectively, then from the perspective of goal (I), its subgoals are satisficed to a degree of 0.8. In contrast, for an OR-decomposition, Athena selects the *maximum* value of each utility value produced by the subgoals in the OR-refinement. For example, if the utility value associated with goals (I) and (J) are 0.8 and 0.9, respectively, then from the perspective of goal (F), its subgoals are satisficed to a degree of 0.9. These semantics capture the notion that to satisfy a goal that has been AND-decomposed all subgoals must be satisfied, whereas to satisfy a goal that has been OR-decomposed, at least one subgoal must be satisfied.

(6) Repeat Steps (1) through (5) Until the Root Goal is Reached.

4 Case Study

This case study presents two different scenarios, each implemented in the Webots simulation platform [14], to illustrate how the set of derived utility functions enable a DAS to perform self-assessment in response to changing system and environmental conditions. The following scenarios involve a single IVS placed 400 meters behind a Lead Vehicle in the same lane. During each simulation, both vehicles accelerate in order to achieve their desired velocities. While the IVS sets its desired speed to 60 km/h, the Lead Vehicle sets its desired speed to 40 km/h. This speed differential makes it necessary for the ACC module in the IVS to readjust its speed to prevent a collision with the Lead Vehicle.

4.1 No Requirements Violations

In this scenario, the utility functions derived by Athena enable the IVS to satisfy its requirements by mitigating conditions conducive to a requirements violation, such as a collision, via dynamic adaptive behavior. Figure 6 shows an excerpt of the values produced by four different utility functions during this simulation, as they relate to the high-level goal (B) in Figure 2. As this plot illustrates, in order to satisfice RELAXed goal (C), the IVS gradually increases its *VehicleSpeed* until it is equivalent to *DesiredSpeed* (at approximately time step 400). The IVS maintains its *DesiredSpeed* until its distance sensors detect the Lead Vehicle, at approximately time step 840 in Figure 6(A). At this point, the metric-based utility function measuring the satisficement of goal (B) reports lower values as the distance between the IVS and the Lead Vehicle decreases. Simultaneously, the IVS switches from satisficing goal (C) to satisfying goal (D). Figure 6(A) illustrates this transition as the utility function that measures the satisficement of goal (D) drops to 0 (time step 860) and then progressively increases to 1 as the IVS achieves its *SafeSpeed*. Lastly, the IVS continues to maintain its *SafeSpeed* until the end of the simulation, at which point the IVS has not violated any invariant goals, as shown by Figure 6(B).

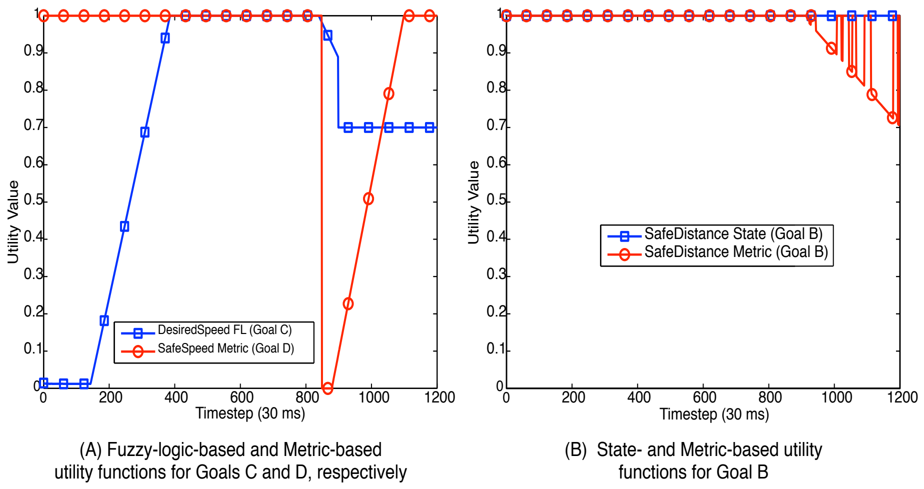


Fig. 6. Plot of recorded utility functions for adaptive cruise control in IVS application

4.2 Sensor Noise Leads to Requirements Violation

In this scenario, the utility functions derived by Athena enable the IVS to detect the violation of an invariant requirement and diagnose potential causes for such a violation. To produce such a requirement violation, we introduced intermittent noise in the forward-bearing sensors of the IVS, which are responsible for detecting and computing the distance between the IVS and the Lead Vehicle. Due to the severe levels of noise applied to these sensors, the IVS is unable to accurately measure the distance to the Lead Vehicle and thus violates several requirements.

Figure 7 shows an excerpt of the values produced by utility functions during this simulation as they relate to the high-level goal (B) in Figure 2. Initially, the

IVS satisfies all invariant goals and satisfices its RELAXed goal (C) by achieving its *DesiredSpeed*, at timestep 450 in Figure 7(A). Shortly thereafter, the IVS distance sensors detect the Lead Vehicle. Due to environmental uncertainty, however, the computed value of *DistanceToObstacle* is unreliable. In particular, forward-bearing distance sensors intermittently report noisy data that suggests no obstacle is present. As a result, the IVS begins to alternate between satisficing goal (C), when distance sensors do not report an obstacle, and satisfying goal (D), when distance sensors report an obstacle. Alternating between these two goals impedes the IVS from successfully achieving its *SafeSpeed* objective.

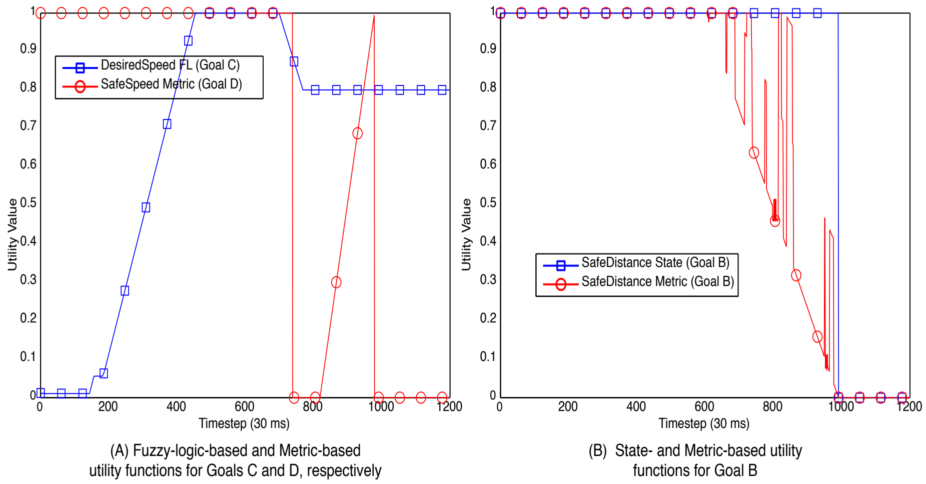


Fig. 7. Plot of recorded utility functions for adaptive cruise control in IVS application

Given the failure to accurately compute the value of *DistanceToObstacle*, the IVS crossed the *SafeDistance* threshold (at approximately time step 1000) and violated the invariant goal (B). The state-based utility function for this invariant goal successfully detected this requirement violation, as can be observed in Figure 7(B), where the *SafeDistance State* utility curve dropped from 1 to 0 for the remainder of the experiment. As the *SafeDistance Metric* utility curve shows, the metric-based utility function for this invariant goal suggested an imminent violation of goal (B) by gradually reporting values closer to 0 as the IVS approached the *SafeDistance* threshold. By leveraging the entire set of utility functions, the DAS is able to partially detect a set of agents involved in the goal's violation. In particular, plotted utility curves indicate goals (B) and (D) were violated during the simulation. Moreover, by examining the utility values produced by (D)'s subgoals, the DAS is able to further pinpoint that goals (H), (K), and (P) were not satisfied multiple times between time steps 700 and 1000, thus suggesting the *NavigationComponent* and *DistanceSensor* agents as root causes for the invariant goal's violation. This information enables requirements engineers to either revise the interactions between the *NavigationComponent* and the *DistanceSensors*, or alternatively, add new refinements to mitigate the obstacle that caused this requirement violation.

5 Related Work

This section overviews related work in specifying partial satisfaction of goals, requirements monitoring, and the use of utility functions for self-assessment in a DAS.

5.1 Partial Satisfaction of Goals

Letier and van Lamsweerde [13] introduced a probabilistic framework for specifying and analyzing the partial satisfaction of goals. Their approach leveraged probability theory to model how requirements may become obstructed, as well as how the obstruction of such goals impact the satisfaction of other goals in the goal-oriented model. In addition, Letier and van Lamsweerde also presented various heuristics to identify probabilistic functions that measure the likelihood of goals becoming unsatisfied. While similar in objective, Athena uses utility functions to measure requirements satisfaction instead of probability theory. As a result, a requirements engineer can apply Athena without requiring data from which goal satisfaction probabilities may be derived.

5.2 Requirements Monitoring

Requirements monitoring focuses on detecting and mitigating both requirements violations and conditions conducive to a requirements violation. Feather, Fickas, and Robinson [7,8,16] developed frameworks for run-time monitoring of software requirements that support the instrumentation, diagnosis, and reconfiguration of the system. To leverage these frameworks, a requirements engineer must first model the system's requirements through a goal-based modeling language, such as KAOS [5], and identify assumptions and constraints that could become violated. At run time, a requirements monitoring framework observes traces of the executing system, logs violations of assumptions and constraints, and then reconciles the system with its goals. Athena shares similar objectives as these requirements monitoring frameworks. However, Athena supports the automatic derivation of utility functions from goal models. Furthermore, Athena supports state-, metric-, and fuzzy logic-based utility functions for measuring the satisfaction and satisficement of goals at run time.

5.3 Utility Functions for Self-adaptive Systems

Utility functions have been applied for self-assessment purposes in DASs. For instance, Walsh *et al.* [17] used utility functions to map monitoring data to a scalar value representative of how well the system was executing, akin to the concept of a *health* value. In this manner, utility functions provide not only an objective and quantitative basis for automated decision-making, but also facilitate the mapping of those decisions to higher-level goals, requirements and concerns. Similarly, utility functions have been applied within a DAS to guide the selection of self-optimizing strategies. For instance, Garlan *et al.* [3] applied utility functions to evaluate and select among different reconfiguration strategies depending on how each satisfied architectural and performance-based constraints. Even though utility functions provide numerous benefits for decision-making within a DAS, these are usually elicited either from domain experts or application users [9].

Statistical regression techniques enable a DAS to infer utility values that capture their overall performance at run time [1,9]. For instance, Valetto *et al.* [9] proposed a statistical correlation-based approach for generating, at run time, a single application-level utility value that measured the most salient properties of that system. These approaches automate the task of deriving utility functions, but their success depends on the quality of monitoring data gathered from the executing DAS. Specifically, regressed utility functions may inadvertently miss the detection of anomalous behaviors if this behavioral data is incomplete or contains undesirable behaviors. Athena does not suffer from such drawbacks as it derives utility functions directly from a goal model. Athena could, however, leverage these techniques to further refine derived utility functions at run time.

6 Conclusions

This paper presented Athena, a goal-based requirements model-driven approach for automatically deriving utility functions from a KAOS or RELAX goal model. In particular, Athena leverages the information contained in a KAOS goal model, as well as its RELAXed goals, corresponding fuzzy logic-based constraints, and MON, ENV, and REL elements to derive utility functions that can be used at run time to monitor the requirements of a DAS. As such, the primary benefit of Athena is that it leverages artifacts already produced by a requirements engineer that applied either a KAOS or RELAX goal modeling approach, thereby enabling a requirements engineer to focus on other aspects of the design rather than on manually deriving utility functions via ad-hoc manual approaches.

These utility functions enable a DAS to assess requirements satisfaction and satisficement at run time. In particular, state- and fuzzy logic-based utility functions enable a DAS to determine whether an invariant or a RELAXed goal has been violated, respectively. Metric-based utility functions enable a DAS to detect conditions conducive to a requirements violation, thereby facilitating the mitigation of such conditions before a goal violation occurs. Experimental results show that utility functions generated by Athena are not only successful at requirements monitoring, but also at identifying a candidate set of agents responsible for the violation of a goal.

Future directions for this work include applying evolutionary computation techniques in order to optimize the set of utility functions generated by Athena. In addition, we are also exploring the use of generated utility functions for adaptive requirements monitoring [15].

References

1. Chajewska, U., Koller, D., Ormoneit, D.: Learning an agent's utility function by observing behavior. In: Proceedings of the Eighteenth International Conference on Machine Learning, ICML, pp. 35–42. Morgan Kaufmann Publishers Inc., San Francisco (2001)
2. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 468–483. Springer, Heidelberg (2009)
3. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems, pp. 2–8. ACM, Shanghai (2006)
4. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Dordrecht (2000)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
6. Darimont, R., van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. SIGSOFT Software Engineering Notes 21(6), 179–190 (1996)
7. Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: IWSSD 1998: Proceedings of the 8th International Workshop on Software Specification and Design, pp. 50–59. IEEE Computer Society, Washington, DC (1998)
8. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: RE 1995: Proceedings of the Second IEEE International Symposium on Requirements Engineering, pp. 140–147. IEEE Computer Society, Washington, DC (1995)
9. de Grandis, P., Valetto, G.: Elicitation and utilization of application-level utility functions. In: The Proceedings of the Sixth International Conference on Autonomic Computing (ICAC 2009), pp. 107–116. ACM, Barcelona (2009)
10. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: Proceedings of the 17th International Conference on Software Engineering, ICSE, pp. 15–24. ACM, Seattle (1995)
11. Jureta, I.J., Faulkner, S., Schobbens, P.Y.: Achieving, satisficing, and excelling. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 286–295. Springer, Heidelberg (2007)
12. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
13. Letier, E., van Lamsweerde, A.: Reasoning about partial goal satisfaction for requirements and design engineering. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 53–62. ACM, Newport Beach (2004)
14. Michel, O.: Webots: Professional mobile robot simulation. Journal of Advanced Robotics Systems 1(1), 39–42 (2004)
15. Ramirez, A.J., Cheng, B.H.C.: Adaptive monitoring of software requirements. In: Proceedings of the 2010 Workshop on Requirements at Run Time, RE@RunTime, pp. 41–50. IEEE Computer Society, Sydney (2010)

16. Robinson, W.N.: Monitoring software requirements using instrumented code. In: HICSS 2002: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, pp. 276–285. IEEE Computer Society, Hawaii (2002)
17. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. In: Proceedings of the First IEEE International Conference on Autonomic Computing, pp. 70–77. IEEE Computer Society, New York (2004)
18. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In: The Proceedings of the 17th International Requirements Engineering Conference (RE 2009), pp. 79–88. IEEE Computer Society, Atlanta (2009)