Using Models at Run Time to Detect Incomplete and Inconsistent Requirements

Byron DeVries
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, 48823, USA
Email: devri117@cse.msu.edu

Betty H. C. Cheng
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, 48823, USA
Email: chengb@cse.msu.edu

***Abstract*—The validity of run-time monitoring of system goals and requirements depends on both the completeness of the requirements, as well as the correctness of the environmental assumptions. Often specifications are built with an idealized view of the environment that leads to incomplete and inconsistent requirements related to non-idealized behavior. Worse yet, requirements may be measured as satisfied at run time despite an incomplete or inconsistent decomposition of requirements due to violated environmental assumptions. While methods exist to detect incomplete requirements at design time, environmental assumptions may be invalidated in unexpected run-time environments causing undetected incomplete decompositions. This paper introduces *Lykus*, an approach for using models at run time to detect incomplete and inconsistent requirements decompositions at run time. We illustrate our approach by applying *Lykus* to a requirements model of an adaptive cruise control system from our industrial collaborators. *Lykus* is able to automatically detect instances of incomplete and inconsistent requirements decompositions at run time.**

I. Introduction

While run-time requirement monitors are intended to measure and report the satisfaction of the requirements in a software system, they are only effective if the requirements are complete and consistent. Problematically, unexpected environmental scenarios that may lead to incomplete or inconsistent requirements in cyber-physical systems (CPS) may also allow run-time monitors to erroneously assess requirements satisfaction. For example, if a requirements decomposition is incomplete, then the decomposed requirement would be satisfied even if the missing requirement(s) were unsatisfied. This paper presents *Lykus*,[1] an approach to automatically detect incomplete requirements coverage at run time.

Detecting incomplete requirements is still an active research area [1], [2], [3], [4], [5], [6], which becomes more complicated when design-time environmental assumptions are found to be invalid at run time. For example, a requirement for a vehicle may be to accelerate. In an idealized system, applying the throttle (e.g., pressing the gas pedal) would be sufficient. However, the assumption that spinning the wheels faster due to an increased throttle affects vehicle speed may be found to be invalid. Given an environmental scenario with low traction (e.g., ice) or where the vehicle's wheels do not touch the ground (e.g., the vehicle is flipped over) would invalidate the earlier assumption when the vehicle did not accelerate. Formal design-time methods to decompose goals and requirements with guaranteed completeness exist [7], though they are limited to only specific formal decomposition rules. Design-time methods exist that are not limited to formal decomposition patterns [6], [8], but make assumptions about the environment that may be shown to be invalid at run time. Problematically, run-time monitoring of incompletely decomposed requirements may indicate satisfaction when the missing requirement(s) would be unsatisfied (e.g., the throttle is increased indicating satisfaction, but the vehicle does not accelerate). Currently, no methods exist to detect incomplete and inconsistent requirements decompositions at run time in order to ensure relevant requirement satisfaction assessment.

This paper describes *Lykus*, the extension of a design time model-driven technique [6] to detect incomplete and inconsistent requirements decompositions at run time. *Lykus* adapts run-time monitors to ensure relevant assessment in the context of the physical environment and requirements model. While requirements in a hierarchical decomposition may be assessed directly or based on the satisfaction of the decomposed child requirements, neither method is sufficient to assess satisfaction alone in the presence of an incomplete or inconsistent decomposition. However, by using a combination of both assessments, it is possible to detect an incomplete or inconsistent decomposition and calculate the correct requirement satisfaction. For example, given a parent requirement to accelerate a vehicle, it may appear that the requirement is satisfied when all of the decomposed child requirements (e.g., increase throttle) are satisfied. If the vehicle *does not* accelerate, then there is an incomplete requirement decomposition and additional requirements that are unrepresented are unsatisfied. Similarly, if the vehicle *does* accelerate but the decomposed requirements are not satisfied, then the parent requirement to accelerate is not being satisfied in the manner specified by the child requirements. Instead, it could be that the vehicle is rolling down a hill rather than accelerating via throttle. In both cases, the intent of the acceleration requirement is not met, therefore the requirement is unsatisfied. *Lykus* identifies incomplete and inconsistent decompositions at run time to dynamically modify requirement satisfaction monitors in order to

[1] *Lykus* is the mortal son of *Ares*, who sacrificed strangers to his father.

provide accurate assessments of the requirements unsatisfied. *Lykus* also identifies violated environmental assumptions when incomplete or inconsistent decompositions are identified at run time.

Lykus uses utility functions [9] to analyze individual requirements within the system specification. Rather than return the raw assessment values generated by the utility functions, *Lykus* identifies incomplete requirements (i.e., additional requirements are necessary but not specified) and inconsistent requirements (i.e., the system is not constrained in the manner defined by the requirements) at run time by comparing the utility function values for parent and child requirements. In the case of incomplete requirements, the parent utility function value is modified to be 'unsatisfied,' as there exists at least one requirement that should have been decomposed (but was not) that is 'unsatisfied.' Similarly, in the case of an inconsistent satisfied requirement, the parent utility function value is also modified to be unsatisfied since the satisfaction did not take place according to the constraints imposed by the decomposed requirements. *Lykus* explicitly uses the decompositions from within a hierarchically decomposed goal model, typically a design-time model, to analyze the decomposition completeness and consistency at run time.

The contributions of this paper are as follows:

- We introduce a run-time approach to automatically detect incomplete and inconsistent requirement decompositions in hierarchical requirements models.
- We adapt the utility function results in the case of incomplete and inconsistent decompositions to correctly assess satisfaction in relation to the physical environment, as opposed to an environmental model.
- We identify the environmental assumptions that are shown to be invalid and are the contributing factor to the incomplete or inconsistent decomposition.
- We present a prototype implementation of the *Lykus* runtime analysis and requirement assessment approach.
- We demonstrate the applicability of *Lykus* on an adaptive cruise control (ACC) system implemented on a rover vehicle.

The remainder of this paper is organized into the following sections. Section II provides an overview of background information. Section III details the approach. Section IV describes an example application, and Section V details related work. Finally, Section VI discusses the conclusions and avenues of future work.

II. Background

This section covers background information on hierarchical requirements modeling, utility functions, and the Adaptive Cruise Control (ACC) system used in this paper.

A. Hierarchical Requirements Modeling

Hierarchical requirements are applicable to multiple requirement frameworks including *i** [10], KAOS goal modeling [11], and hierarchical requirements modeling [12]. Intrinsic to all hierarchical requirements modeling approaches, high-level requirements are defined by a collection of decomposed lower-level requirements that are necessary for the satisfaction of the high-level requirement. Decomposition may occur over numerous levels, and it terminates when a set criteria is met.

The portions of KAOS goal modeling we employ realizes Goal-Oriented Requirements Engineering (GORE) by defining the decomposition of parent goals as a graph. Either AND- or OR-decompositions may be employed, where a parent requirement is satisfied if all of its AND-decomposed requirements are satisfied or if any of its OR-decomposed requirements are satisfied [11]. KAOS goal model decomposition terminates when a decomposed goal can be satisfied by an agent. In cases where the agent measures the environment, the decomposed goal is an *environmental expectation*. In cases where the agent performs an action as part of the system-to-be, the decomposed goal is a *system requirement*.

The models defined in this paper for use in the examples apply portions of the KAOS goal modeling notation, but we do not use the KAOS formal decomposition patterns or other formal KAOS modeling methods. As *Lykus* is generally applicable to any hierarchical goal and requirement modeling framework, we use the terms goals and requirements interchangeably throughout this paper. Practically, the role of goals, requirements, and expectations are generally differentiated, however we treat them identically during analysis. In this paper, goal and requirement model labels are in **`bold courier`** font, while variable names, goal and requirement text, and emphasis are indicated by *italics*.

1) Complete Decomposition: Parent requirements are completely decomposed if the satisfaction of the aggregate decomposed child requirements ensures that the parent requirement is satisfied. That is, a requirement is incompletely decomposed if the decomposed requirements are satisfied yet the parent requirement remains unsatisfied. In that case, at least an additional requirement is necessary to satisfy the parent requirement. More formally, the satisfaction of the decomposed requirements (i.e., $R_1, R_2, R_3, R_{...}, R_n$), along with any domain properties and assumptions (i.e., Dom) ensure that the parent requirement (i.e., R) is satisfied as illustrated in Equation (1) [11].

$$\{R_1, R_2, R_3, R_{...}, R_n, Dom\} \vDash R. \tag{1}$$

2) Consistent Decomposition: Decomposed requirements are one method of constraining the parent requirement to a single desirable solution, amongst potentially numerous other possible decompositions. For example, if a parent requirement is to stop a vehicle, then two alternative solutions would be to apply the brakes or remove the engine. Clearly one solution is undesirable. Consistent decomposition implies that the parent requirement satisfaction is only achieved when the desired solution is achieved. That is, the solution (i.e., decomposed requirements, $R_1, R_2, R_3, R_{...}, R_n$, and domain properties, Dom) also prevent undesired solutions of the par-

ent requirement (i.e., R). Formally this is defined in Equation (2).

$$R \vDash \{R_1, R_2, R_3, R_{...}, R_n, Dom\}. \quad (2)$$

B. Utility Functions

The satisfaction of requirements has been assessed by run-time monitors [13], [14] implemented as utility functions [15]. While typically a Boolean expression, *satisficement* may be used to represent a degree of satisfaction [16], similar to the concept of *satisficing* [17] but without the assumption of sufficiency (i.e., the amount of satisfaction may not be sufficient for the system). The utility functions used in this paper are generated from three sources used to capture environmental properties (**ENV**, **MON**, **REL**) by Athena [9], and are defined as follows:

- **ENV** defines environmental properties related to the satisfaction of the goal that may or may not be directly observable (e.g., the expected speed of a vehicle at a future time),
- **MON** defines monitors (e.g., agents and sensors) in the system that are able to monitor specific values (e.g., a speed sensor that measures the speed of a vehicle at the current time), and
- **REL** represents relationships between the monitors and environmental properties that relate to the satisfaction or satisficement of goals (e.g., relating expected future vehicle speed and current vehicle speed to measure the satisficement of a requirement to increase speed).

The **REL** properties are used by Athena to compute the degree of satisfaction (i.e., satisficement) of a requirement, and returns the value as either state-, metric-, or fuzzy-logic based values. Domain properties (e.g., a throttle increase will spin the wheels faster causing acceleration) used by Athena [9] to generate utility functions are present in the environmental properties (**ENV**, **MON**, and **REL**) specified by the system designer [18] based on their knowledge of the environment and domain.

For example, the utility function created for Goal **C.1**, where the speed in the future ($Speed_{t+1}$) must be *greater* than the current speed ($Speed_t$) in order to achieve a faster speed, results in the utility function shown in Figure 1.

```
bool sat_of_c1() {
  // Assess REL satisfaction
  if(speed_t < speed_t_plus_1) {
    // Return satisfied
    return 1.0;
  } else {
    // Return unsatisfied
    return 0.0;
  }
}
```

Fig. 1: Utility Function Listing of **C.1**

C. Adaptive Cruise Control Systems

Figure 2 details a hierarchical goal and requirements model of an Adaptive Cruise Control (ACC) system that uses distance sensors to ensure a safe following distance from the car ahead by adjusting vehicle speed while simultaneously maintaining as close to the desired speed as possible. Abbreviations *M* and *A* are used for *Maintain* and *Achieve*, respectively. Leaf nodes (i.e., agents) that are numbered in the requirements model are provided in TABLE I. The ACC model defined here has been previously shown to be complete [6], given the assumed environmental conditions.

TABLE I: Agents used in Goal Model

#	Agent (Sensor / Actuator)
1	Cruise Switch Sensor
2	Cruise Active Sensor
3	Cruise Active Switch
4	Throttle Pedal Sensor
5	Throttle Actuator
6	Brake Pedal Sensor
7	Brake Actuator
8	Speed Sensor 1
9	Speed Sensor 2
10	Distance Sensor 1
11	Distance Sensor 2

The ACC model in Figure 2 is defined by four primary components: cruise control modes (i.e., goals with a prefix of '**A.**'), speed increase (i.e., goals decomposed from **C.1**), speed decrease (i.e., goals decomposed from **B.2**), and maintain speed (i.e., goals decomposed from **D.1**). The speed is increased or decreased to maximize speed up to the desired speed while maintaining a safe distance from the target car (i.e., the car immediately in front). In cases where the safe distance is violated, the speed is decreased regardless of the desired speed. In cases that both the desired speed and safe distance are met, the speed is maintained.

Goal **C.1**, which is used throughout as an example, is decomposed into three children: **C.2**, **C.3**, and **C.12**. The Goal **C.1** ('*Achieve(Faster Speed)*') increases the speed by increasing the throttle via Goal **C.3** ('*Achieve(Increase Throttle)*'). The throttle is increased while ensuring braking is minimized via Expectation **C.12** when the *Speed* is less than the *Desired Speed* (Expectation **C.4**) and the *Distance* measured is greater than the defined *Safe Distance* (Expectation **C.9**).

The utility functions for the requirements in Figure 2 are derived from the **ENV**, **MON**, and **REL** properties in TABLE II, where, for brevity, only values related to the ongoing example presented in this paper are presented. TABLE III defines the variable value ranges and units.

III. APPROACH

An overview of *Lykus* is presented in a Data Flow Diagram (DFD) in Figure 3. Processing elements are represented by circles. Persistent data is represented by parallel horizontal bars. The labeled arrows indicate data flows, and boxes represent external entities. *Lykus* makes use of Athena [9] to generate the utility functions that are used as run-time

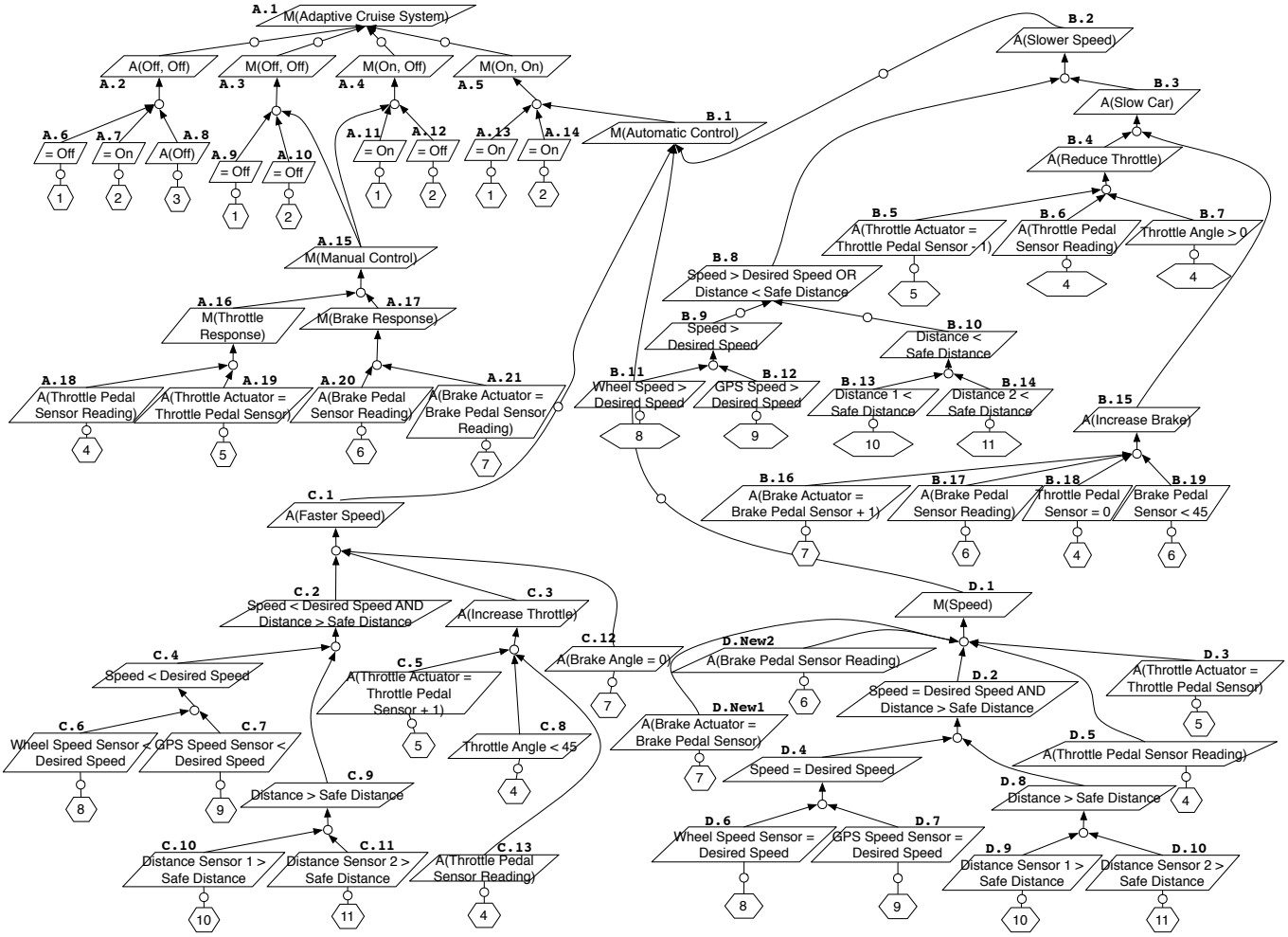

Fig. 2: Adaptive Cruise Control Goal Model

TABLE II: **ENV**, **MON**, and **REL** Properties

	ENV	**MON**	**REL**
C.1	$Speed_t, Speed_{t+1}$		$Speed_t < Speed_{t+1}$
C.2	$Speed_t, Distance$	*Desired Speed*, *Safe Distance*	$Speed_t <$ *Desired Speed* $\wedge$ *Distance* $>$ *Safe Distance*
C.3		*Throttle Actuator*, *Throttle Pedal Sensor*	*Throttle Actuator* $>$ *Throttle Pedal Sensor*
C.12		*Brake Actuator*	*Brake Actuator* $==$ *MIN*
$Speed_t$		*Speed Sensor 1*, *Speed Sensor 2*	*Speed Sensor 1* $\vee$ *Speed Sensor 2*
$Speed_t$		*Throttle Pedal Sensor*, *Brake Pedal Sensor*	$\max($*MIN*, *Throttle Pedal Sensor* $-$ *Brake Pedal Sensor*$)$
$Speed_{t+1}$		*Throttle Actuator*, *Brake Actuator*	$\max($*MIN*, *Throttle Actuator* $-$ *Brake Actuator*$)$
Distance		*Distance Sensor 1*, *Distance Sensor 2*	*Distance Sensor 1* $\vee$ *Distance Sensor 2*

monitors to detect incomplete and inconsistent decompositions of parent requirements in the goal model (e.g., Figure 2) using the properties in TABLE II. Optionally, *Ares* can be used to detect incomplete requirements decompositions that can be identified at design-time [6]. The utility functions and goal model are then used by *Lykus* to generate logical expressions that represent the decompositions within the hierarchically decomposed goal model to detect both incomplete and inconsistent decompositions. These logical expressions are used by *Lykus* to generate executable monitoring code that detects incomplete and inconsistent decompositions and accurately report parent satisfaction. The executable monitoring code generated by *Lykus* is used at run time to monitor the system to detect counterexamples and accurately report satisfaction throughout the system's execution.

Lykus is applicable to systems that interact with their environment, where the environment is anything that is outside of the system-to-be. The environment may be other systems, rather than a physical environment. Additionally, the system must be able to sense its expected impact on the environment to allow utility functions to measure the satisfaction of individual requirements directly, or indirectly using relationships between multiple sensors. Finally, requirements must be defined hierarchically to detect decompositional counterexamples.

TABLE III: Units and Scaling for Variables in TABLE II

Variable	Min	Max	Unit
$Speed_t$	0.0	100.0	MPH
$Speed_{t+1}$	0.0	100.0	MPH
Distance	0.0	50.0	Inches
Desired Speed	0.0	100.0	MPH
Safe Distance	0.0	50.0	Inches
Throttle Actuator	0.0	100.0	%
Throttle Pedal Sensor	0.0	100.0	%
Brake Actuator	0.0	100.0	%
Brake Pedal Sensor	0.0	100.0	%
Distance Sensor 1	0.0	50.0	Inches
Distance Sensor 2	0.0	50.0	Inches
Speed Sensor 1	0.0	100.0	MPH
Speed Sensor 2	0.0	100.0	MPH
Cruise Switch Sensor	*Off*	*On*	Boolean
Cruise Active Sensor	*Off*	*On*	Boolean
Cruise Active Switch	*Off*	*On*	Boolean

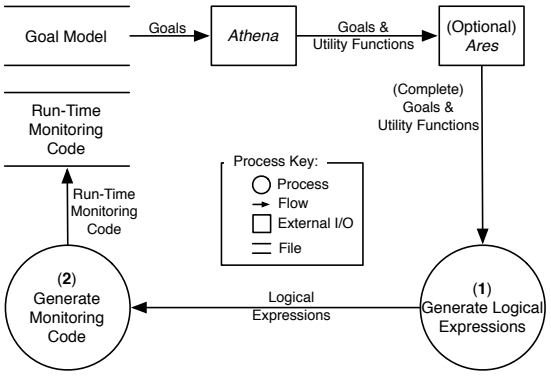

Fig. 3: *Lykus* Data Flow Diagram

A. DFD Step 1: Generate Logical Expression

The input to Step 1 is a goal model that *may* be analyzed at design-time for incomplete requirement decompositions using *Ares* [6] and utility functions generated by Athena [9]. *Lykus* outputs a set of requirements monitors and additional logic that detects incomplete and inconsistent requirements decomposition and provides the status of the parent requirement satisfaction.

Unlike design-time solutions that use utility functions for analysis across an entire range of possible scenarios [6], *Lykus* applies the utility value functions at run time for a single scenario that the system is currently experiencing. Based on the results of the comparisons of the realized utility value functions, analysis is performed each time the utility functions are calculated at run time. Counterexamples are identified, and subsequently adapted immediately as needed on the run-time requirement assessments.

TABLE IV lists the satisfaction of both the parent requirement and the set of decomposed child requirements along with the updated parent requirement satisfaction status to reflect the run-time evaluation of the parent in actual environmental conditions. For each decomposed requirement, two additional checks are performed, one for incomplete decomposition and one for inconsistent composition. If the requirement is neither incomplete nor inconsistent, then the requirement's utility function is unmodified. The possible parent and decomposed child requirement satisfaction results listed in TABLE IV are discussed in the following subsections. If the parent's and decomposed child requirements' satisfactions do not match, then the parent must be unsatisfied due to either incomplete or inconsistent decomposition. TABLE IV identifies the possible combinations of parent and decomposed child requirements satisfaction, along with the updated parent satisfaction based on run-time satisfaction measures.

TABLE IV: Satisfaction Cases

Row	Utility Function		Updated Parent Requirement
	Parent Requirement	Child Requirements	
1	Unsatisfied	Unsatisfied	Unsatisfied
2	Unsatisfied	Satisfied	Unsatisfied
3	Satisfied	Unsatisfied	Unsatisfied
4	Satisfied	Satisfied	Satisfied

1) Standard Unsatisfied Requirements: In the case where both the parent and the child requirements are unsatisfied (i.e., Row 1 in TABLE IV), the satisfaction of the parent does not change and the parent requirement remains unsatisfied at run time. Since both the utility function representing the satisfaction of the parent requirement and the combined satisfaction of child requirements are both unsatisfied, then the result is neither incomplete nor inconsistent. That is, there is not a known missing requirement due to an unsatisfied parent requirement nor is there an inappropriately satisfied parent requirement due to a solution not specified by the decomposed child requirements. When a parent requirement is unsatisfied, we would normally expect the decomposed child requirements to be unsatisfied.

2) Unsatisfied Due to Incompleteness: In the case where the parent is unsatisfied, yet the decomposed child requirements are satisfied (i.e., Row 2 in TABLE IV), the parent requirement should remain unsatisfied due to an incomplete decomposition. While the aggregate child requirements indicate that the parent requirement should be satisfied, they only indicate satisfaction due to a missing requirement that would alter the satisfaction of the aggregate set.

3) Unsatisfied Due to Method Used: In the case where the parent requirement is satisfied, yet the decomposed child requirements are unsatisfied (i.e., Row 3 in TABLE IV), the parent status must be modified to indicate that it is unsatisfied. While the parent requirement utility function indicates that the parent requirement is satisfied, the method of satisfaction is not constrained to the solution provided by the decomposed requirements.

4) Standard Satisfied Requirement: In the case where the parent requirement and the decomposed child requirements are both satisfied (i.e., Row 4 in TABLE IV), then the decomposition is neither incomplete nor inconsistent. Regardless of the method of assessing the requirement (i.e., directly or via the requirement's decomposed child requirements) the assessment results in satisfaction for both the parent and children.

B. DFD Step 2: Generate Monitoring Code

The logic for comparing parent requirement and aggregate child requirements satisfactions is generated for each decom-

posed parent requirement. Neither direct measurement of each requirement (e.g., measuring parent requirements for satisfaction) nor measuring a requirement's aggregate decomposed requirements (e.g., measuring the set of child requirements for satisfaction) is sufficient to assess the satisfaction of the parent requirement in all cases. An implementation of TABLE IV to calculate the updated satisfaction of requirement **C.1** at run time is given in Figure 4. This approach differs from the *Ares* approach, since here we detect at run time if a single specific environmental and system scenario related to the current state of the system includes incomplete or inconsistent requirements decomposition. The requirements, contents of the

```
bool updated_sat_of_c1() {
  // Assess Parent Satisfaction
  bool parent_sat = sat_of_c1();
  // Assess Aggregate Child Satisfaction
  bool child_sat = sat_of_c2() &&
      sat_of_c3() && sat_of_c12();

  if(parent_sat == child_sat)
    return parent_sat;
  else {
    // Save variables and requirement
    record_counterexample(''C.1'');

    // Unsat if incomplete / incorrect
    return false;
  }
}
```

Fig. 4: Updated Utility Function Listing of **C.1**

variables, and (when included) environmental assumptions are recorded upon the detection of an incomplete or inconsistent requirements decomposition. Similar functions that update parent requirement status, as measured from the original utility functions, are provided for each of the parent requirements in the goal model in order to detect incomplete and inconsistent decompositions and the associated variables at run time.

C. Execution Time & Deployment

For each requirements decomposition, the computational effort of *Lykus* grows linearly with respect to the number of decomposed requirements, assuming a constant maximum size of each utility function. Instead of calculating if a requirements decomposition is incomplete or inconsistent in any scenario, *Lykus* calculates if a requirement decomposition is incomplete or inconsistent in only the current scenario (e.g., the agents periodically read from sensors to support the calculation of the utility functions). This is similar to checking a known NP problem, SAT, for a single set of true or false assignments rather than solving which true or false assignments satisfy the Boolean expression.

More formally, identifying incomplete or inconsistent requirements decompositions exists in NP-Complete. That is, identifying an incomplete or inconsistent requirements decompositions is computationally expensive but verifying a specific incomplete or inconsistent requirements decomposition can be done very quickly. *Lykus* leverages the latter property and verifies the decompositional completeness and consistency with respect to only the current environmental scenario.

Given that utility functions are widely used as run-time monitors [9] (e.g., "sat_of_$*$()" in Fig 3), the additional cost *Lykus* incurs is used to calculate a single Boolean expression based on the concrete satisfaction of the utility functions (e.g., calculation of "parent_sat == child_sat" in Fig. 3) for each decomposition.

Lykus calculates if each decomposition is incomplete or inconsistent for the current scenario as measured by the utility functions and their respective monitor properties (i.e., agents and sensors). The utility functions are updated as new sensor results are available (as often as 20 times a second). When the utility function values are updated, the decompositions that include the requirements with updated utility function values are checked for incompleteness. That is, the decompositions of the requirements specification are evaluated and re-evaluated for incompleteness throughout the execution of the system the generated run-time monitoring code is deployed in.

D. Limitations

The accuracy of *Lykus* is only as good as the accuracy of the **ENV**, **MON**, and **REL** properties used to define the utility functions. Additionally, counterexamples may be present in a scenario that occurs only between sensor readings. In such cases, the values calculated by the utility functions never represent an incompleteness or inconsistency and, therefore, no counterexample is detected.

IV. EXAMPLES

This section covers examples of an incomplete requirement decomposition and an inconsistent requirement decomposition.

A. Experimental Setup

The updated utility functions and detection logic are executed during the operation of a small autonomous car. The car comprises a 16 MHz ATmega328 microcontroller, and two speed controllers driving 4 motors turning 4 wheels. Distance measurements are provided by an ultrasonic sensor, speed measurements are provided by a cumulative accelerometer, and user input is provided by infrared remote control.

The small autonomous car was placed into two different environmental scenarios intended to elicit the detection of both incomplete and inconsistent decompositions.

B. Incomplete Requirement Decomposition

In order to demonstrate an incomplete requirement decomposition for a parent requirement, the aggregate child requirements must be satisfied while the parent requirement itself is unsatisfied. Using goal **C.1** as an example, the car must be placed in a position where the utility function is invalid. Specifically, the speed cannot increase over some

TABLE V: Incomplete Decomposition Values

Row	Variable or Goal	Value
1	Brake Actuator	*MIN*
2	Brake Pedal Sensor	*MIN*
3	Desired Speed	*MAX*
4	Distance	*MAX*
5	Distance Sensor 1	*MAX*
6	Distance Sensor 2	*MAX*
7	Safe Distance	6 Inches
8	Speed Sensor 1	*MIN*
9	Speed Sensor 2	*MIN*
10	Speed_t	*MIN*
11	Speed_{t+1}	*MIN*
12	Throttle Actuator	80%
13	Throttle Pedal Sensor	70%
14	Goal `C.1`	Unsatisfied
15	Goal `C.2`	Satisfied
16	Goal `C.3`	Satisfied
17	Goal `C.12`	Satisfied

given time despite an increase in throttle with no braking. An environmental scenario outside of the idealized environmental model (i.e., as defined in TABLE II for rows $Speed_t$, $Speed_{t+1}$, and *Distance*) may elicit previously undetected incomplete requirement decompositions.

In this example, we physically flip the vehicle onto its back, allowing none of the wheels to touch the ground. Since our idealized environmental model never considered the possibility of rolling the vehicle over, the standard method of accelerating does not apply. The updated utility function code generated by *Lykus* detects an incomplete decomposition from `C.1` at run time and records the incomplete decomposition, the system and environmental variables, and violated environmental assumptions.

Importantly, in order to maintain the set speed, the cruise control system *attempts* to accelerate by increasing the throttle (via goal `C.3`) while the brake is not applied (via goal `C.12`); there is a safe distance to any upcoming obstacle and the current speed is less than the desired speed (via expectation `C.2`). Despite these decomposed child goals being satisfied, the speed (both at the current time and in the future) are 0. Due to the detected incomplete decomposition, parent goal `C.1` is reported as *unsatisfied* based on run-time monitored conditions. The variables recorded for the incompleteness are shown in TABLE V and are limited to a resolution of 10% for percentage based values and one tenth (i.e., 0.1) for all other values due to the truncation of sensor and actuator resolution to minimize oscillation and measurement error.

Given the list of environmental assumptions defined in TABLE II, the values of the counterexample variables can be used to identify violated environmental assumptions at run time. TABLE VI includes the list of environmental assumptions and indicates if they are valid or not. It is important to note that if the optional design-time completeness detection was not performed by *Ares*, there may be no environmental assumptions documented. In this case, TABLE VI shows that rows 2 and 3 both include violated environmental assumptions related to the calculation of speed from *only* the brake and throttle, ignoring the possibility of outside influences (e.g., rollovers).

TABLE VI: Environmental Assumptions: Incompleteness

Row	Environmental Assumption	Valid
1	$Speed_t = Speed\ Sensor\ 1 \vee Speed_t = Speed\ Sensor\ 2$	True
2	$Speed_t = \max(MIN,$ *Throttle Pedal Sensor* $-$ *Brake Pedal Sensor*$)$	False
3	$Speed_{t+1} = \max(MIN,$ *Throttle Actuator* $-$ *Brake Actuator*$)$	False
4	$Distance = Distance\ Sensor\ 1 \vee$ $Distance = Distance\ Sensor\ 2$	True

TABLE VII: Inconsistent Decomposition Values

Row	Variable or Goal	Value
1	Brake Actuator	*MIN*
2	Brake Pedal Sensor	*MIN*
3	Desired Speed	2 MPH
4	Distance	*MAX*
5	Distance Sensor 1	*MAX*
6	Distance Sensor 2	*MAX*
7	Safe Distance	6 Inches
8	Speed Sensor 1	1.2 MPH
9	Speed Sensor 2	1.2 MPH
10	Speed_t	1.2 MPH
11	Speed_{t+1}	2 MPH
12	Throttle Actuator	20%
13	Throttle Pedal Sensor	20%
14	Goal `C.1`	Satisfied
15	Goal `C.2`	Unsatisfied
16	Goal `C.3`	Unsatisfied
17	Goal `C.12`	Satisfied

Lykus is able to generate code to use utility functions to detect incomplete requirements decompositions at run time to ensure run-time relevant requirements assessment.

C. Inconsistent Requirement Decomposition

In order to demonstrate an inconsistent requirements decomposition, the speed must increase despite not increasing the throttle. We achieve this case by allowing the vehicle to drive off a 'cliff,' represented by the edge of a desk. The car accelerates through its fall despite not increasing the throttle. Just as with the incomplete requirements decomposition, the updated utility function code generated by *Lykus* detects an inconsistent decomposition from `C.1` and records the inconsistent decomposition, the system and environmental variables, and violated environmental assumptions.

In this case, the ACC system accelerates without increasing the throttle (via goal `C.3`) while the brake is not applied (via goal `C.12`). While there is a safe distance to any upcoming obstacle the current speed is not less than the desired speed (via expectation `C.2`). Despite not satisfying these decomposed child goals, the speed increases due to the precipitous drop. Due to the detected inconsistent decomposition, goal `C.1` is reported as *unsatisfied*, as it is not satisfied in the method required by the decomposed child requirements. The variables recorded for the inconsistency are shown in TABLE VII and are limited to a resolution of 10% for percentage based values and one tenth (i.e., 0.1) for all other values due to the truncation of sensor and actuator resolution to minimize oscillation and measurement error.

Given the list of environmental assumptions previously identified in TABLE II, TABLE VIII lists the environmental assumptions and indicates their run-time validity. Similar to the incompleteness counterexample, rows 2 and 3 both include violated environmental assumptions related to the calculation of speed from *only* the brake and throttle, ignoring the possibility of outside influences (e.g., drastic changes in terrain).

TABLE VIII: Environmental Assumptions: Inconsistency

Row	Environmental Assumption	Valid
1	$Speed_t = Speed\ Sensor\ 1 \vee Speed_t = Speed\ Sensor\ 2$	True
2	$Speed_t = \max(MIN,$ $Throttle\ Pedal\ Sensor - Brake\ Pedal\ Sensor)$	False
3	$Speed_{t+1} = \max(MIN,$ $Throttle\ Actuator - Brake\ Actuator)$	False
4	$Distance = Distance\ Sensor\ 1 \vee$ $Distance = Distance\ Sensor\ 2$	True

Lykus is able to generate code to use utility functions to detect inconsistent requirements decompositions at run time to ensure run-time relevant requirements assessment.

D. Threats to Validity

Since *Lykus* is most realistically validated at run time, rather than using a simulation or static analysis, the validation is limited to the finite set of scenarios that occur. We have ensured that both inconsistent and incomplete requirements decompositions can be identified, however additional inconsistent and incomplete requirements decompositions may exist within the requirements model. The validation of the detection classification, however, is done by specific cases (i.e., TABLE IV) within the description of the approach. Additionally, methods that could manage sensor uncertainty (e.g., RELAXed goals and requirements [19]) are not included in the examples of incomplete and inconsistent requirements.

V. RELATED WORK

This section overviews related work, including requirements completeness and run-time monitoring of requirements. Unlike proposals to use run-time specific models to support run-time analysis [20], we use goal models intended for design time use for run-time analysis.

A. Requirements Completeness

Multiple methods have been developed to identify incomplete requirements decomposition. We overview these strategies and compare them to *Lykus*. Obstacles to requirements completeness have been generated using search-based techniques [1], however the counterexamples must be manually reviewed for applicability. Formal methods of guaranteeing complete requirements exist for behavioral state-based systems [21], formally described requirements using theorem provers [11], and low-level functional details [21]. Additionally, decomposition with formally-proven completeness properties also exist, where a system defined by repeated application of the formal decomposition patterns guarantees completeness. Problematically, formal methods are intrinsically heavyweight solutions that often require expertise with theorem proving [11] or limit possible solutions to the formally defined patterns.

Tools also exists that identify single [6] or multiple [8] counterexamples in hierarchical requirements decompositions at design time without restrictions on decomposition patterns or heavyweight formal descriptions and analysis.

Lykus differs by acknowledging that invalid environmental assumptions made at design time may leave incomplete decompositions intact until they occur at run time. *Lykus* detects these incomplete decompositions at run time rather than at design-time and provides more accurate utility function assessment in the presence of incomplete decompositions.

B. Run-Time Monitors

Several frameworks exist to monitor requirements at run time [13], [14], [22] that are intended to support instrumentation, diagnosis, and reconfiguration operations within the system. The utility functions whose values we adapt in *Lykus*, as generated by Athena [9], provide the same support with the benefit of automatic generation from a set of environmental properties. *Lykus* differs from existing run-time monitors by detecting incomplete and inconsistent requirements at run time and adapting the results of existing run-time monitors [9] to provide more accurate results.

VI. CONCLUSIONS

In this paper, we have presented *Lykus*, a run-time approach for detecting incomplete and inconsistent requirements decomposition and using that information to identify invalid environmental assumptions and unsatisfied requirements. *Lykus* reports the issues detected with requirements, as well as the invalid environmental assumptions, while automatically updating the results of requirements monitoring.

We demonstrate *Lykus* on an adaptive cruise control system implemented on a robotic vehicle and developed in collaboration with our industrial collaborators. We show that *Lykus* is able to detect incomplete and inconsistent requirements at run time due to invalid environmental assumptions. Specifically, we show that while existing tools identify no incomplete requirements, incomplete decompositions may still exist due to incorrect assumptions that are detectable using run-time monitors.

In the future, we plan to apply *Lykus* to additional examples, including requirements models with RELAXed requirements [19] that make use of fuzzy logic in specifying requirements. Additionally, we plan to investigate how *Lykus* can be used to trigger additional mitigations at run time due to unsatisfied requirements that would not be detected with other methods.

ACKNOWLEDGMENT

This work has been supported in part by NSF grants CNS-1305358 and DBI-0939454, Air Force Research Lab grant, Ford Motor Company, and General Motors Research. 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, Air Force Research Lab, Ford, GM, or other research sponsors.

REFERENCES

[1] D. Alrajeh, J. Kramer, A. v. Lamsweerde, A. Russo, and S. Uchitel, "Generating obstacle conditions for requirements completeness," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 705–715.

[2] B. H. C. Cheng and J. M. Atlee, "Research directions in requirements engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 285–303.

[3] A. Ferrari, F. dell'Orletta, G. O. Spagnolo, and S. Gnesi, "Measuring and improving the completeness of natural language requirements," in *Requirements Engineering: Foundation for Software Quality*. Springer, 2014, pp. 23–38.

[4] I. Menzel, M. Mueller, A. Gross, and J. Doerr, "An experimental comparison regarding the completeness of functional requirements specifications," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010, pp. 15–24.

[5] M. M. Zenun and G. Loureiro, "A framework for dependability and completeness in requirements engineering," in *Latin American Symposium on Dependable Computing*, 2013, pp. 1–4.

[6] B. DeVries and B. H. Cheng, "Automatic detection of incomplete requirements via symbolic analysis," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 385–395.

[7] R. Darimont and A. Van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6. ACM, 1996, pp. 179–190.

[8] B. DeVries and B. H. Cheng, "Automatic detection of incomplete requirements using symbolic analysis and evolutionary computation," in *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*. ACM, 2017, pp. 49–64.

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

[10] E. S. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," in *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*. IEEE, 1997, pp. 226–235.

[20] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos, "Runtime goal models: Keynote," in *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. IEEE, 2013, pp. 1–11.

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

[12] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal verification of avionics software products," in *International Symposium on Formal Methods*. Springer, 2009, pp. 532–546.

[13] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior," in *Proceedings of the 9th international workshop on Software specification and design*. IEEE Computer Society, 1998, p. 50.

[14] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," in *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*. IEEE, 1995, pp. 140–147.

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

[16] J. Whittle, P. Sawyer, N. Bencomo, and B. H. C. Cheng, "A language for self-adaptive system requirements," in *Service-Oriented Computing: Consequences for Engineering Requirements, 2008. SOCCER'08. International Workshop on*. IEEE, 2008, pp. 24–29.

[17] H. A. Simon, "Rational choice and the structure of the environment." *Psychological review*, vol. 63, no. 2, p. 129, 1956.

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

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

[21] M. P. Heimdahl and N. G. Leveson, "Completeness and consistency in hierarchical state-based requirements," *Software Engineering, IEEE Transactions on*, vol. 22, no. 6, pp. 363–377, 1996.

[22] W. N. Robinson, "Monitoring software requirements using instrumented code," in *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*. IEEE, 2002, pp. 3967–3976.