Goal-Oriented Patterns for UML-Based Modeling of Embedded Systems Requirements

Heather J. Goldsby\textsuperscript{1}, Sascha Konrad\textsuperscript{2}, Betty H.C. Cheng\textsuperscript{1†}

\textsuperscript{1}Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA
\textsuperscript{2}Siemens Corporate Research, Inc., 755 College Road East, Princeton, NJ 08540, USA
E-mail: hjg@cse.msu.edu,sascha.konrad@siemens.com,chengb@cse.msu.edu

Abstract

Embedded systems are used for critical applications that must adhere to safety constraints. Developers of these systems face three key challenges when attempting to apply existing requirements analysis approaches: (1) declaratively specifying and modeling functional requirements, non-functional requirements, and constraints; (2) operationally modeling the required behavior; and (3) analyzing the requirements models of the behavior for adherence to the constraints. To address these challenges, this paper introduces COBRA patterns, which provide complementary UML and goal model templates to be instantiated in tandem to create models that capture system requirements and their constraints. Structural consistency is established through the construction of the goal and UML diagrams. Behavioral consistency is achieved by analyzing the UML models for adherence to the constraints specified in the goal model.

1 Introduction

Embedded systems are frequently used for critical applications that must operate reliably while adhering to safety constraints. There is an increasing interest in applying model-driven development (MDD) to embedded systems, where models are successively refined from requirements to design and eventually to code. Using MDD, undetected errors in the requirements model can be propagated to code. As such, analyzing the requirements models for adherence to safety constraints is particularly important. Several approaches [16, 28, 30, 35] assist developers in using model checkers to formally analyze models for adherence to constraints. However, developers face three key challenges when attempting to apply these existing requirements analysis approaches: (1) declaratively specifying and modeling functional requirements, non-functional requirements, and constraints in an integrated fashion; (2) operationally modeling the required behavior; and (3) analyzing the requirements models of the behavior for adherence to the constraints. To address these challenges, this paper introduces COBRA patterns, which provide complementary UML and goal model templates to be instantiated in tandem to create models that capture multiple views of system requirements and their constraints. The goal model template captures non-functional requirements and declaratively specifies functional requirements and constraints. The UML model template operationally specifies behavior that satisfies the requirements and provides a path to leverage MDD techniques. Consistency is established between the goal and UML models. First, structural consistency is achieved through construction by relating elements of the goal model to elements of the UML structural diagram. Second, behavioral consistency is achieved by formally analyzing the UML model for adherence to the constraints specified by the goal model.

Several notable approaches provide patterns that relate goal models to UML design models [9, 13, 38]. The general idea behind these patterns is to leverage the complementary nature of goal and UML models to obtain a more complete system specification. Gross and Yu proposed patterns that relate non-functional goal models, specified with the NFR framework [3], to design patterns [13]. Wang et al. similarly proposed patterns that address security patterns at the design level [38]. Fletcher and Cleland-Huang proposed softgoal traceability patterns [9], which enhance the patterns defined by Gross and Yu with bidirectional traceability relationships that enable elements of the UML class diagram to be generated from the goal model. However, none of these patterns address the functional requirements of the system and thus are not able to assist the developer in formally determining if the behavior specified by the UML
model satisfies the functional requirements of the embedded system.

In contrast, COBRA (Constraints and OBjects for Requirements Analysis) patterns relate goal models to UML models at the requirements level. COBRA patterns provide three key benefits. First, the instantiated goal model template declaratively specifies the functional and non-functional requirements of the embedded system and refines the requirements into constraints. Second, the instantiated UML model template operationally specifies behavior that satisfies the requirements. Specifically, the structure of the system is captured using UML class diagrams, and behavior is captured using UML state diagrams. These diagrams can be used as part of existing MDD approaches, some of which involve automatic code generation. Third, structural and behavioral consistency is established between the resulting goal and UML models. We tailored a variation of the softgoal traceability pattern approach [9] for requirements-level models to provide structural consistency, which establishes that the goal model and the UML class diagram are consistent. Behavioral consistency establishes that the goal model and the UML behavioral diagrams are consistent. It is achieved by formally analyzing the UML models for adherence to the constraints specified by the goal model.

To capture the essential components of an embedded system, the COBRA patterns model the requirements associated with sensors, actuators, controllers, and user interfaces. To facilitate usability, we use a template similar in style to that used by Gamma et al. [10]. The template has been modified to apply to address early and late requirements engineering. Specifically, it addresses early requirements engineering by identifying when to apply the pattern, how to apply the pattern, and the goals for the pattern. The template addresses late requirements engineering by identifying the structure and behavior of the relevant system components for the pattern, and the outcome of applying the pattern.

We illustrate the COBRA patterns by applying them to the modeling and analysis of an electronically controlled steering system obtained from industry. The remainder of the paper is organized as follows. Section 2 reviews goal modeling notation and two tools used to establish behavioral consistency. Section 3 describes the COBRA patterns and provides an illustrative example. Section 4 describes structural and behavioral consistency. Section 5 applies the COBRA patterns to the modeling of an electronically controlled steering system. Section 6 overviews related work. Finally, in Section 7 we present conclusions and discuss future work.

2 Modeling and Analysis

This section overviews the goal modeling notation used to specify the goal model templates. Also, we briefly describe two previously developed tools, Hydra [30] and SPIDER [21, 23], that are used to check behavioral consistency.

2.1 Goal Modeling Notation

We use the Non-Functional Requirements (NFR) [3] approach to specify the goal model templates. In general, goals are system objectives and goal models specify goals and the relationships among goals. Goal models enable developers to evaluate alternative solutions and to document the rationale behind requirements, that is, goal models describe why a requirement exists [37]. Our COBRA patterns contain four types of goals with their respective icons in parentheses:

- **Softgoal**: (cloud) a non-functional objective whose achievement cannot (always) be formally evaluated, e.g., reusability and quality.
- **Softgoal operationalization**: (shaded cloud) a softgoal that describes a development technique or artifact that contributes to a softgoal [9, 13]. For the purposes of this paper, softgoal operationalizations are used to relate softgoals to the late requirements engineering artifacts.
- **Functional goal**: (roundtangle) a functional objective of the system that can be achieved, e.g., detect failure.
- **Constraint goal**: (shaded roundtangle) a new type of goal that we introduce to describe a functional objective of the system that can be verified through formal analysis.

Additionally, there are two types of goal relationships. First, a contribution relationship (depicted as a line with the word “helps” or “hurts”) connects a goal to a softgoal and indicates if the element helps or hurts, respectively, the realization of the softgoal. Second, a refinement relationship (depicted as a line with the word AND or OR) elaborates how the goal is achieved. A goal is AND-refined if only one of the sub-goals must be achieved for the goal to be achieved. A goal is OR-refined if only one of the sub-goals must be achieved for the goal to be achieved. Within the COBRA patterns, softgoal operationalizations contribute to softgoals. Additionally, softgoal operationalizations are refined by functional goals, which are, in turn, refined by constraint goals.

2.2 Analyzing Behavior

After instantiating a COBRA pattern, we use the Spin model checker [15] to assess behavioral consistency. The model checker verifies that the UML model adheres to the constraints specified by the goal model. We use two previously developed tools, Hydra [30] and SPIDER [21, 23], to facilitate the process. We use Hydra to translate the UML model to a formal representation. For this work, we use SPIDER to translate the constraints specified by the goal
model to an LTL representation that can be analyzed by the Spin model checker.

**Hydra.** To enable UML diagrams to be formally analyzed, McUmber and Cheng [30] previously developed a metamodel-based formalization framework that maps a given UML model into a formal specification language, e.g., Promela, the specification language for the model checker Spin [15]. Hydra automates this mapping process [30]. The UML-to-Promela formalization approach was developed to capture semantics for embedded systems behavior. Objects are mapped to processes in Spin (processes) that exchange messages via channels. Nested and concurrent states are also formalized as processes. Additional details on the underlying formalization framework can be found in [30].

**Spider.** SPIDER [21, 23] (Specification Pattern Instantiation and Derivation Environmment) is a process with corresponding tool support that enables a developer to use a natural language-based grammar to specify properties of UML models. SPIDER can map the natural language representations of properties to formal specification languages, such as Linear Temporal Logic (LTL) [29] and computational tree logic (CTL) [4] formulae.

3 COBRA Patterns

COBRA patterns assist developers in constructing and analyzing complementary UML and goal models of the requirements for embedded systems. Table 1 is a catalog of the COBRA patterns identified thus far. For each pattern, the table describes its classification (structural or behavioral), a brief description of the pattern, the primary functional goal of the pattern, and how some of the key non-functional requirements of embedded systems, e.g., performance, modifiability, affordability, and reusability, are affected by the pattern. We briefly describe some of the columns in greater detail. Patterns classified as structural contain UML templates that are focused on identifying objects, abstracting objects into classes, and capturing relationships between classes; whereas, behavioral patterns contain UML templates that specify the behavior of objects/classes by defining the essential behavior of reactive objects. Within the non-functional requirements columns, plus and minus signs indicate whether a given pattern helps (+) or hurts (-) a given non-functional requirement; if the field is blank, then the pattern does not have a significant impact on the softgoal. For our current COBRA patterns, there is generally a tradeoff between the non-functional requirements (i.e., softgoals) in that a pattern that helps achieve performance generally increases the cost of the system and vice-versa. Modifiability and reusability are similarly disjoint. In the following, we overview the COBRA pattern template and illustrate the template fields on the frequently used Computing Component pattern.

To construct the COBRA patterns, we leveraged and extended our previously developed object analysis patterns [19, 24]. Whereas design patterns [10] guide developers in the construction of design models, the object analysis patterns guide developers in the creation of conceptual UML models of the functional requirements of embedded systems during the (requirements) analysis phase preceding the design phase [6]. We have used the object analysis patterns to model a number of automotive applications, including a diesel filter system, adaptive cruise control application, and an electronically controlled steering system. Additionally, feedback from our industrial collaborators has been incorporated into the COBRA patterns. Specifically, the COBRA patterns combine elements of the object analysis patterns with a new goal model template, introduce additional behavioral templates, and address non-functional requirements. These new elements provide a declarative method of specifying requirements, relate the requirements to higher-level functional and non-functional goals, refine the requirements into precisely specified constraints, and provide additional assistance for UML modeling.

3.1 COBRA Template

Figure 1 depicts the template used to present the COBRA patterns and describes each of the template fields. We have augmented the template used by Gamma et al. for design patterns [10] to reflect our emphasis on requirements engineering. Specifically, the Implementation and Sample Code fields have been removed and the Goal Model Template, Behavior, and Applicable Design Patterns fields have been added. Additionally, the pattern template has been divided into four sections. The Pattern Name and Classification section identifies the pattern and classifies it as structural or behavioral. The Early and Late Requirements Engineering sections each comprise fields suited for the respective phase in the software development life cycle. Specifically, the Early Requirements Engineering section includes a goal model template. The Late Requirements Engineering section includes UML structural (class diagram) and behavioral (state and sequence diagram) templates. The Pattern Meta-data section identifies related COBRA patterns, design patterns, and uses of the pattern.

3.2 Example COBRA Pattern

We illustrate the fields of the COBRA pattern template in the context of the Computing Component pattern. Computing components collect information about the environment using sensors, select an action based on the current environmental conditions, and affect the environment using actuators. The complete set of patterns and further details about the COBRA patterns can be found in the extended technical report [12]. (Due to space constraints, the following
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Classification</th>
<th>Description</th>
<th>Primary Functional Goal</th>
<th>NFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Sensor</td>
<td>Structural</td>
<td>Specify various kinds of active sensors and their relationship to the computing component(s)</td>
<td>Monitor environment; broadcast information (push)</td>
<td>Performance: +, Modifiability: -, Affordability: +, Reusability: +</td>
</tr>
<tr>
<td>Passive Sensor</td>
<td>Structural</td>
<td>Specify various kinds of passive sensors and their relationship to the computing component(s)</td>
<td>Monitor environment; requires explicit request information (pull)</td>
<td>Performance: -, Modifiability: +, Affordability: +, Reusability: -</td>
</tr>
<tr>
<td>Actuator</td>
<td>Structural</td>
<td>Specify various kinds of actuators and their relationship to the computing component(s)</td>
<td>Influence environment by setting an actuator value</td>
<td>Performance: -</td>
</tr>
<tr>
<td>Control</td>
<td>Structural</td>
<td>Describes how to model the controls of a user interface, in order to acquire input from the user</td>
<td>Receive information from user</td>
<td>Performance: +</td>
</tr>
<tr>
<td>Controller Decompose</td>
<td>Structural</td>
<td>A global view on the relationships among the components in a system</td>
<td>Decompose system into components</td>
<td>Performance: -</td>
</tr>
<tr>
<td>Indicator</td>
<td>Structural</td>
<td>Describes how to model the indicators of a user interface, in order to signal the current system state to the user</td>
<td>Provide information to user</td>
<td>Performance: +</td>
</tr>
<tr>
<td>Communication Link</td>
<td>Behavioral</td>
<td>Specify how the system should interact with external entities, e.g., diagnostic devices</td>
<td>Interact with external device, e.g., for fault diagnostics</td>
<td>Performance: -, Modifiability: +, Affordability: +, Reusability: -</td>
</tr>
<tr>
<td>Computing Component</td>
<td>Behavioral</td>
<td>Specify various operating modes for the computing components of an embedded system</td>
<td>Distribute computational tasks</td>
<td>Performance: +</td>
</tr>
<tr>
<td>Corrector</td>
<td>Behavioral</td>
<td>Describes how fault correction capabilities can be included in the system</td>
<td>Correct faults</td>
<td>Performance: -</td>
</tr>
<tr>
<td>Detector</td>
<td>Behavioral</td>
<td>Describes how fault detection capabilities can be included in the system</td>
<td>Detect faults</td>
<td>Performance: -</td>
</tr>
<tr>
<td>Fault-Handler</td>
<td>Behavioral</td>
<td>Describes how to model global and local fault handlers to provide a higher level of abstraction</td>
<td>Centralized handling of faults</td>
<td>Performance: -</td>
</tr>
</tbody>
</table>

Table 1. COBRA Pattern Evaluation Table

1. Pattern Name and Classification:
   A descriptive handle for the pattern and the purpose of the pattern.
2. Early Requirements Engineering:
   - Intent: Describes the problems that the pattern addresses.
   - Motivation: Describes the objectives that motivate the use of the pattern.
   - Applicability: Describes the conditions in which the pattern may be applied.
   - Goal Model Template: Depicts generic softgoals, functional goals, softgoal operationalizations, and constraint goals for the pattern.
3. Late Requirements Engineering:
   - Structure: Represents classes and their relationships in UML class diagrams.
   - Behavior: Represents the behavior of the relevant objects in UML state and sequence diagrams.
   - Participants: Enumerates the classes/objects that are included in the pattern and their responsibilities.
   - Collaborations: Describes how classes/objects interact and their roles.
   - Consequences: Identifies the outcome of applying a pattern.
4. Pattern Meta-data:
   - Applicable Design Patterns: Suggests design patterns that could refine the pattern.
   - Also Known As: Lists alternative names for the COBRA pattern.
   - Known Uses: Examples of the pattern found in models of embedded systems.
   - Related COBRA Patterns: Lists and compares related COBRA patterns to this pattern.

Figure 1. COBRA pattern template

contains only the representative information for each field of the pattern. For clarity, some explanatory text, which is denoted in italics, for the contents of the selected fields is included.)

1. Pattern Name and Classification
   Computing Component: Behavioral Pattern
2. Early Requirements Engineering
   Intent:
   Specify computing components for a real-time embedded system.
   Motivation:
   This pattern is concerned with the computing components (often also referred to as controllers) of an embedded system. Computing components carry out computations and interact with the environment through the sensor and actuator interfaces. In a distributed embedded system, in contrast to a centralized embedded system, several computing components exist that interact through message passing. Computing components commonly have several possible modes of operation. For example, in a distributed flight control system of an airplane, none of the computing components should shut down completely in response to a fault. In the case of a sufficiently severe fault at a specific computing component, this component should enter a partial shutdown mode and offers basic functionality needed to safely operate the plane. Generally, depending on the severity of the fault, a computing component may change its current operational mode, such as performing the partial shutdown upon detecting a component failure.
Goal model template:
Figure 2 depicts the goal model template for the Computing Component pattern. We use letters enclosed in circles to label the diagram elements and thus facilitate this description. The Computing Component pattern hurts (A) Affordability because hardware and software redundancies may be required for some operational modes, but helps (B) Reusability because several computing components may be constructed using variants of the same model. The softgoal operationalizations, i.e., goals (C) - (H), constrain the structure of the UML model template instantiation. This relationship is described in greater detail in Section 4. The functional goals, e.g., (I) Initialize components, (J) Change operational mode, and (K) Achieve operational goal, specify the functional requirements that should be satisfied by the pattern. The functional goals are, in turn, AND-refined by constraint goals, e.g., (L), each of which describe a specific, analyzable property that instantiations of the UML model template for the Computing Component pattern should satisfy. These constraint goals are specified using structured natural language, which enables a developer to use SPIDER [21, 23] to translate the natural language constraint goal into a formal representation.

Note: Instantiating a goal model template has two steps. First, each goal in the template is customized by replacing the generic underlined text (e.g., element (D) in Figure 2) with information about the specific embedded system under development. The template can be further customized by adding additional softgoals relevant to the system under development. These softgoals can refine the existing softgoals or introduce new non-functional requirements that are affected by the COBRA pattern. Second, the instantiated goal model template is incorporated into the overall goal model for the embedded system by establishing a contribution or refinement relationship between the softgoal naming the pattern instance (e.g., (D) Computing component name) and a softgoal in the system goal model.

3. Late Requirements Engineering
Structure:
The UML class diagram of the Computing Component pattern is shown in Figure 3. In a distributed system, a Computing Component communicates with other Computing Components through message passing, while in a centralized embedded system only one such component exists. In addition, a Computing Component may interact with the User Interface in order to provide information to the user about the current operational state, as well as receive input from the user. Note: A developer instantiates this diagram by constructing concrete classes that inherit from the abstract classes depicted in the diagram.

Behavior:
The following operational modes are provided for Computing Components [6]. Due to space constraints, the state diagram template for the Computing Component class has been elided. However, it can be found in the extended technical report [12]. Within the template, each operational mode is represented as a state.

- **Power off**: State preceding activation of a Computing Component.
- **Initialize**: In this state, the Computing Component initializes. For example, the Computing Component checks the operational states of all components.
- **Normal behavior**: State when no faults have occurred and the Computing Component is functioning normally.
- **Manual/external control**: In this state, the Computing Component is controlled by an external entity, such as a diagnostic device.
- **Production stop**: This state is useful, for example, when a human enters a hazardous area. The Computing Component should be able to complete its current task and secure the environment, but it should shutdown as soon as possible.
- **Protection stop**: Ceases operation immediately, but does not turn off power. This state is appropriate, for example, when a Computing Component needs to be stopped, but a device should continue to operate to avoid hazardous situations. For example, a cooling device should remain working even in case of a system malfunction.
- **Partial shutdown**: The Computing Component offers basic functionality; for example, Computing Component in medical devices may remain in a monitoring state.
- **Hold**: No functionality is provided in this state, but safety actions are taken; for example, a rocket self-destructs in the case of abnormal functions. There is no outgoing transition from this state; a Computing Component can only be reactivated by a complete restart.

Furthermore, an emergency stop can be performed by the system. This stop state is not modeled as a separate state because an emergency stop takes the system to the Power off state immediately. Mechanical backup devices may be required then to assure a safe functioning of the system when the power is off. For example, an ABS system still provides standard mechanical breaking abilities after the system has been powered off.

Note: Not all operational modes are needed in every Computing Component of a distributed system. For example, controllers in ABS systems generally do not possess partial shutdown states, because the system constraints require that an inactive system should not affect the basic functionality of the brakes. Therefore, an emergency stop, where all controllers cut power, is sufficient. A developer instantiates the state diagram templates by providing additional detail within the hierarchical states and modifying the transitions to describe the operations and attributes of elements of the instantiated class diagram.

Participants: 
AbstractComputingComponent, AbstractUserInterface

Collaborations:
- A Computing Component performs computations and interacts with the environment of the system.
- The User Interface is used to signal the user of the system the current operational mode of the system. This information is important so that the user is aware of a system that is working in safe mode. In addition, a user provides input to the system through the User Interface, e.g., setting the temperature of an air conditioning unit, a value that is then used...
Figure 2. Goal model template for the Computing Component pattern

Consequences:
1. Required safety operational modes should be implemented in the ComputingComponents.
2. Hardware and software redundancies may be required for some operational modes, thereby increasing system cost.

4. Pattern Meta-data
Applicable Design Patterns: Strategy pattern [10], Double Checked Locking pattern [32], Acceptor-Connector pattern [32]
Also Known As: To be determined.
Known Uses: To be determined.
Related COBRA Patterns: Detector, Corrector, Fault Handler, Indicator

4 Consistency Analysis

In this section, we discuss establishing structural and behavioral consistency between the goal and UML models.

4.1 Structural Consistency by Construction

Structural consistency identifies the classes/objects whose behavior is constrained by the application of the pattern. It is established during the construction of the goal and UML models. Specifically, structural consistency creates an explicit relationship between the softgoal operationalizations of an instantiated goal model template and elements of the corresponding instantiated UML model template. There are four types of softgoal operationalizations
and each provides increasingly concrete information about the UML class diagram template instantiation. We illustrate these types of softgoal operationalizations with reference to the generic elements defined by the Computing Component goal model template depicted in Figure 2:

1. At an abstract level, the name of the pattern is identified, e.g., (C) Computing component pattern.
2. Context is provided by naming the pattern instance, e.g., (D) Computing component name.
3. Softgoal operationalizations identify the constituent components of the UML class diagram template. For example, (E) Computing component abstract class and (F) User interface abstract class identify the abstract classes defined by the UML class diagram template for this pattern (depicted in Figure 3).
4. Softgoal operationalizations identify the classes that must be used to instantiate the UML class diagram template. For example, (G) Concrete computing component and (H) Concrete user interface name the concrete classes constructed to instantiate the UML class diagram template for a specific computing component.

Syntactic consistency is established when the UML class diagram is constructed. Specifically, if the UML class diagram is constructed to contain the abstract and concrete classes identified by the instantiation of the goal model template, then structural consistency is established. Structural consistency provides developers with guidance for maintaining both models. If a constraint goal within an instantiated goal pattern template is changed, then the structural consistency relationships highlight the UML class elements whose state diagrams will potentially need to be modified to meet the constraint. If a state diagram for a UML class is changed, then the structural consistency relationships highlight the goals that may need to be modified.

4.2 Behavioral Consistency by Analysis

Behavioral consistency is established through formal analysis. Specifically, the UML model resulting from COBRA pattern instantiations is analyzed for adherence to the constraint goals. The analysis has three steps. First, using Hydra [30], the UML model is translated to a formal representation amenable to analysis using the Spin model checker [15]. Second, using SPIDER [21, 23] the constraint goal is translated to a formal representation by instantiating all the placeholders with expressions that use application-specific variables derived from the UML model. Third, the formalized UML model is analyzed for adherence to the formally stated constraint goal. Any errors that are detected need to be corrected prior to applying another pattern. To determine the source of an error, the violation trace provided by the model checker may be inspected, or visualized in the original UML diagrams using Theseus [11], which is a UML visualization tool that enables developers to visualize model checking analysis results in terms of UML model elements. If the violation trace represents a valid behavior of the system, then the property specified by the constraint goal is erroneous and needs to be modified. In cases where an erroneous behavior of the system is detected, the UML system model needs to be corrected prior to applying another pattern to add more details to the model.

5 Example Application

In this section, we use the COBRA patterns to model a portion of an electronically controlled steering system (ECSS) obtained from industry. We briefly overview the ECSS description and discuss structural and behavioral consistency between the models.

5.1 ECSS Case Study

The ECSS is intended to supplement the benefits provided by traditional hydraulic power steering. This all-electric and engine-independent system eliminates the traditional hydraulic system’s power steering pump, hoses, and hydraulic fluid, as well as the drive belt and pulley on the engine. Instead, the ECSS uses an electric-motor power assist mechanism to provide responsive power steering. The ECSS provides variable assistance with turning the wheels based on the current speed of the car and the amount of torque (turning force) applied to the steering wheel. This adjustment provides for a much safer ride, as small steering wheel movements at high speeds will not cause the car to swerve.

To model the ECSS, the developer identifies the primary goal of the system, which is Provide responsive power steering. Next, the developer refines the primary goal until there is a relationship between each child goal and a COBRA pattern. Table 2 enumerates the COBRA pattern instantiations used to model the ECSS. Specifically, it describes the order of the pattern instantiations1, the goal of each of the instantiations, and the name of the COBRA pattern applied. In the first two instantiations, the Passive Sensor pattern is applied to model the goals and key structural elements and their behavior of the sensors in the ECSS. Subsequently, the Actuator pattern is applied to capture the motor providing assistance to the driver in turning the steering wheel. In the fourth pattern instantiation, the processing capabilities of the system are modeled using the Computing Component pattern that determines the assistance that needs to be provided to the driver based on the sensor values. Finally, in the last two pattern instantiations, fault detection and corrections provided to the driver based on the sensor values. Finally, in the last two pattern instantiations, fault detection and corrections

1In general, we first model sensors, then actuators, then computing components, and lastly fault handling components.
tion is modeled using the Detector and Corrector patterns, respectively.

<table>
<thead>
<tr>
<th>No.</th>
<th>Instantiation Description</th>
<th>COBRA Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Determine torque applied to wheel</td>
<td>Passive Sensor</td>
</tr>
<tr>
<td>2</td>
<td>Determine car speed</td>
<td>Passive Sensor</td>
</tr>
<tr>
<td>3</td>
<td>Provide power assistance to turn wheel</td>
<td>Actuator</td>
</tr>
<tr>
<td>4</td>
<td>Compute necessary power assistance</td>
<td>Computing Component</td>
</tr>
<tr>
<td>5</td>
<td>Detect sensor and actuator errors or faults</td>
<td>Detector</td>
</tr>
<tr>
<td>6</td>
<td>Remediate sensor and actuator errors</td>
<td>Corrector</td>
</tr>
</tbody>
</table>

### Table 2. ECSS Pattern Instantiations

#### 5.2 Structural Consistency

Figure 4 depicts elided portions of the goal model and UML class diagram for the ECSS. We use letters enclosed in circles to label the goal model elements. Specifically, numbered labels, e.g., (I), correspond to a pattern instantiation identified in Table 2 and primed labels, e.g., (D'), are the instantiated version of a goal, e.g., (D), depicted in Figure 2. Each pattern instantiation identified in Table 2 modified both the goal model and UML model. Specifically, the goal model depicts several softgoals that were constructed to seed the goal model, i.e., softgoals (0) - (6). Softgoal (0) Provide responsive power steering describes the primary objective of the ECSS and softgoals (I) - (6) describe a pattern instantiation. Each of these softgoals was refined by instantiating the goal model template of a COBRA pattern. However, only the partial refinement of (4) Compute necessary power assistance and (I) Determine torque applied to wheel are depicted; the other refinements have been elided due to space constraints.

All diagram elements corresponding to a given pattern instantiation are denoted with the same line style (e.g., dotted, dashed). For example, (I) Determine torque applied to wheel applies the Passive Sensor pattern. The softgoal operationalizations of its goal model template identifies classes Abstract PassiveSensor, Abstract PassiveRealSensor, TorqueSensor, AbstractComputingComponent, and AComputingComponent. In the case where a given class diagram element is involved in more than one pattern instantiation, all relevant lines are depicted. Note that classes Abstract ComputingComponent and AComputingComponent have been shaded black, since all of the applied COBRA pattern instances identify them.

#### 5.3 Behavioral Consistency

Behavioral consistency analyzes the UML state diagrams for adherence to the constraint goals. Figure 5 depicts the state diagram for the TorqueSensor, one of the components in the ECSS. This diagram was created for pattern instantiation (I) Determine torque applied to wheel by instantiating the UML model template for the Passive Sensor COBRA pattern.

Assessing behavioral consistency has three steps. First, the UML model is formalized using Hydra. Second, the constraint goals are formalized using SPIDER. For example, the Passive Sensor goal model template contains the constraint goal Globally, it is never the case that a sensor value exceeds its maximum. As depicted in Figure 4, this constraint goal has been customized for the torque sensor of the ECSS to be (X') Globally, it is never the case that a torque sensor value exceeds 10 ft-lbs.. Using SPIDER, the text of this customized constraint goal can be instantiated using UML model-specific elements to be Globally, it is never the case that TorqueSensor.value > 10. SPIDER can then automatically translate this constraint to a formal representation. Third, the formalized UML model is analyzed for adherence to this formally stated constraint goal. If the analysis does not succeed, either or both the goal and UML models should be refined.

### 6 Related Work

Numerous patterns for architecture and design have been proposed. Specifically, others have identified design patterns [10], software architecture patterns [33], database access patterns [18], fault-tolerant telecommunication system patterns [1], design patterns tailored to distributed real-time embedded systems [34], design patterns for avionics control systems [27], real-time design patterns [7, 8], and security patterns [25]. However, in contrast to the COBRA patterns, these patterns focus on architecture and design, rather than requirements and analysis. They are closer to implementation and are generally concerned with tasks, active objects, scheduling, packaging generated object artifacts, the distribution of run-time components, and detailing the definition of interfaces [7]. These patterns do not include a complementary goal model.

A number of approaches relate goal models to UML models [2, 9, 13, 14, 17, 26, 31, 36, 38]. Some approaches provide patterns that relate goal models to UML design models [9, 13, 38]. Differing from our approach, none of these patterns address the functional requirements of the system and thus are not able to assist the developer in determining if the behavior specified by the UML model satisfies the functional requirements of the embedded system. Others have developed UML profiles to represent goal models in UML [14, 31]. Notably, Mylopoulos et al. [31] extended UML to represent Tropos and Heaven and Finkelstein [14] extended UML to represent KAOS [5]. In contrast to the COBRA patterns, these approaches do not address inter-model consistency. Additionally, Jürjens [17] developed a
goal model and a UMLsec model for security-related concerns, but did not address structural or behavioral consistency.

In summary, none of the aforementioned approaches combines the use of goal models to capture correctness properties, the use of UML models to operationally specify how to satisfy those properties, and establish structural and behavioral consistency between the goal and UML models.

7 Conclusion

This paper described COBRA patterns, which assist developers in constructing goal and UML models of an embedded system. The goal model declaratively specifies requirements, which are refined into constraints. The UML model operationally specifies behavior that satisfies the requirements. Structural consistency establishes that the goal model and UML class diagram are consistent. Behavioral consistency is achieved by analyzing the UML model for adherence to the constraint goals specified by the goal model.

Numerous directions for future work are possible. The COBRA patterns could be extended to include real-time information, where we will leverage real-time extended versions of our property templates [22] and UML formalization framework [20]. In addition, our approach could be applied to domains other than embedded systems. This extension could lead to new COBRA patterns, as well as promote the generalization of already existing patterns. Lastly, the development of a browser for the COBRA patterns is in progress.
References


[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


