A Requirements Patterns-Driven Approach to Specify Systems and Check Properties*

Sascha Konrad, Laura A. Campbell, Betty H. C. Cheng**, and Min Deng

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
{konradsa,campb222,chengb,dengmin1}@cse.msu.edu

Abstract. In order to use object-oriented development techniques and UML for embedded systems, we previously developed a framework, *Hydra*, for adding formal semantics to a collection of UML diagrams that enable the automated derivation of formal language specifications for those diagrams. Recently, we have also identified a number of *requirements patterns* for embedded systems that includes sample UML structural and behavioral diagrams for modeling requirements and high-level design for embedded systems. This paper describes a requirements patterns-driven approach for developing UML diagrams for embedded systems, where each pattern has a constraints section to specify safety and other invariant properties. We describe how the diagrams, via specifications generated from *Hydra*, can be automatically analyzed for adherence to these formally specified constraints using the SPIN model checker. In addition, this paper describes how this approach has been applied to an air filter system to reduce the amount of soot emitted from diesel truck exhaust.

1 Introduction

It is well-known that requirements modeling and analysis is one of the most difficult tasks in the software development process [20], but this problem is greatly exacerbated for embedded systems given the hardware constraints and the potentially complex control logic. Previously, we developed modeling/visualization and formalization frameworks and tools to facilitate the rigorous development of embedded systems. Specifically, we have tools to support the graphical modeling of requirements (MINERVA [3]), the translation of these models into formal specifications (Hydra [4, 15]) that can then be analyzed using the SPIN simulator and model checker [9], and the visualization of errors captured in terms of the original graphical models (MINERVA). Recently, we identified a number of *requirements*

* This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CC-9984726, and CC-9901017, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in cooperation with Siemens Automotive and Detroit Diesel Corporation.

** Please contact this author for all correspondences.

patterns for use in the development of requirements and high-level design for embedded systems [14]. We constructed a requirements pattern template, much in the spirit of the template used by Gamma *et al.* [7] for design patterns. This paper describes how a requirements-patterns-driven development process, using Hydra, MINERVA and SPIN, can be used to model and analyze requirements for an automotive application from industry, an air particulate filter system to reduce the amount of soot emitted from diesel truck exhaust.

Given the safety-critical nature of many embedded systems, methods for modeling and developing embedded systems and rigorously verifying behavior before committing to code are increasingly important. Currently, much of the embedded systems industry use *ad hoc* development approaches [5]. The embedded systems community appears, however, to be interested in exploring how object-oriented modeling, specifically the *Unified Modeling Language* (UML) [2], can be used for embedded systems [5]. Our requirements patterns use the UML to model structural and behavioral information, using class diagrams, and sequence and state diagrams, respectively. This information can be used to guide the construction of UML models for embedded systems. Our modeling tool, MINERVA, and formal specification generation tool, Hydra, enable developers to model their systems and check the models for adherence to critical properties. In addition, our visualization utilities in MINERVA depict errors detected by the analysis tools in terms of the original diagrams, thereby greatly accelerating the development and refinement process. The alternative for evaluating the UML models is to use visual inspection. A few tools that support UML diagrams enable developers to perform high-level syntactic analysis [19], or generate formal specifications, such as SDL [21], animate state diagrams and sequence diagrams [11], or generate code for simulation [8, 18]. But none of these utilities provide the collective capabilities captured by our approach and tools, nor do they attempt to reuse organized information such as that captured by our requirements patterns.

We applied the requirements patterns to several embedded systems to determine their utility. The requirements pattern template includes motivation, consequences, high-level goals, context information, constraints, and diagrams depicting templates for structure and behavior. The **Constraints** field of the template includes formal specifications of properties that should be satisfied in the context of using a given pattern [12]. The constraints are described in prose and specified in LTL (Linear Temporal Logic) according to *specification patterns* developed by Dwyer *et al.* [6]. We found that requirements patterns enable novices, guided by the structure and behavior diagrams in the templates, to quickly construct models of their systems. Also, the requirements patterns prompt developers to consider aspects of a system that might otherwise be overlooked until much later in the development process, such as fault tolerance and safety considerations.

Requirements patterns can provide both guidance to novices of embedded systems development for determining the key elements of many embedded systems, and examples of how to model these elements with a commonly accepted diagramming notation, UML. With the formalization capability, we are able to

validate (using simulation) the behavior of the requirements as captured by the state diagrams [3] within the structural context imposed by the class diagrams. Furthermore, constraints from the requirements patterns can guide novices in constructing formal properties to check against their UML models. The result is that developers can accelerate the initial development of requirements models through the use of the requirements patterns, and then using our formalization work and tools, they have a mechanism to rigorously check the requirements using simulation and model checking techniques.

The remainder of this paper is organized as follows. Section 2 briefly describes our modeling/visualization and formalization frameworks and the tools instantiating them, and overviews requirements patterns. Section 3 illustrates how requirements patterns can be incorporated into our iterative modeling and analysis process for creating, analyzing, and refining UML diagrams. Section 4 contains an overview of the Diesel Filter System (a system of particulate filters to reduce soot in diesel truck exhaust) and describes the preliminary results of developing and analyzing the UML diagrams. Section 5 gives concluding remarks and discusses future investigations.

2 Background

This section overviews several technologies contributing to the project, including MINERVA and Hydra that support UML, used to give high-level descriptions of systems, and requirements patterns that describe information typically used for describing requirements of embedded systems.

2.1 Modeling/Visualization and Formalization Frameworks

We have developed a general framework [4, 15, 17] for formalizing a subset of UML diagrams in terms of different formal languages [15–17]. The formal (target) language chosen should reflect and support the intended semantics for a given domain. This formalization framework enables the construction of a set of rules for transforming UML models into specifications in a formal language [4]. The resulting specifications derived from UML diagrams enable either execution through simulation or analysis through model checking, using existing tools. The mapping process from UML to a target language has been automated in a tool called Hydra [15].

To complement Hydra's formalization framework for automatic generation of formal specifications, we developed a modeling and visualization framework [3] to support a number of tasks necessary to model and analyze UML diagrams. These tasks have been automated in a tool called MINERVA [3] that includes the following capabilities: graphical construction of syntactically correct UML diagrams[1]; and visualization of consistency-checking results, simulation traces,

[1] We use Honeywell's domain model editor toolkit, DoME [10], to build the graphical editor for UML diagrams.

and paths of execution that lead to errors, all in terms of UML diagrams. For this paper, we focus on the embedded systems domain, instantiating MINERVA and Hydra with formalization rules for Promela to be used with the SPIN simulator and model checker [9].

2.2 Requirements Patterns

We developed a template to describe requirements patterns [14] by modifying the original design pattern template [7] to address the needs of requirements engineering. Modifications relevant to this paper include extending the original design pattern template with **Constraints** and **Behavior** sections that contain specification-pattern-based [6] representations of properties of interest, and sequence and state diagrams that illustrate sample behavior, respectively. Thus far, our constraints have included representations of two of Dwyer *et al.*'s [6] most popular general specification pattern categories, *universality/absence* (to capture invariant properties) and *response* (to capture cause/effect relationships in system behavior). Requirements pattern constraints provide a template for instantiating properties specific to a modeled system in terms of the UML diagrams describing the system. See [14] for further details on the requirements pattern template.

We have identified several patterns to describe requirements for the main elements of an embedded system. Figure 1 gives a list of the requirements patterns that have been identified to date with a brief description of each. The complete set of requirements patterns and their full descriptions are given elsewhere [14].

Actuator-Sensor:	How to specify various kinds of sensors and actuators and their relationships to a controller in an embedded system.
Controller Decompose:	How to decompose an embedded system into different components according to their responsibilities.
Monitor-Actuator:	How to increase safety and reliability by monitoring actuator behavior for errors.
Fault Handler:	How to integrate a fault handler into an embedded system.
Channel:	How to arrange communication between two components.
Watchdog:	How to monitor a device or system conditions and initiate corrective action(s) if a violation is found.
Examiner:	How to monitor a device and store occurring errors.
User Interface:	How to specify a user interface that is extensible and reusable.
Mask:	How to reduce the burden placed on the computing component when many sensors and actuators are present, whose values need to be sorted or filtered into single values for the computing component.
Moderator:	How to provide an interface to support decoupling of complex subsystems.

Fig. 1. Current list of requirements patterns for embedded systems

3 Modeling and Analysis Process

Figure 2 overviews our approach, illustrating how requirements patterns can be combined with the iterative modeling and analysis process supported by MINERVA and Hydra [3, 15] (here instantiated with the model checker SPIN [9]). The user begins by selecting appropriate requirements patterns based on the requirements of the system. Using the structural and behavioral diagrams in the requirements patterns as a guide, the user constructs UML class and state diagrams in MINERVA's graphical editors (Figure 2, part A). Hydra performs consistency checks (Figure 2, part B), and MINERVA visualizes structural consistency-checking results (dash-dotted arc in Figure 2, part F). (We omit discussion of these capabilities in this paper; see [3] for details.) Hydra then generates formal specifications from textual representations of UML diagrams (Figure 2, part C); these formal specifications can be used to validate the UML diagrams through simulation using SPIN (Figure 2, part D). In addition, the user may instantiate (as LTL claims) properties from the **Constraints** section of those requirements patterns used to guide the modeling of the system (Figure 2, part E). These LTL claims, defined in terms of attributes and states of the UML model, can then be checked against the UML diagrams (Figure 2, part D) through model checking using SPIN. Finally, MINERVA visualizes behavior simulation and counterexample traces (solid arc, Figure 2, part F) via state diagram animation, generation/animation of *collaboration diagrams* (which depict the paths of communication, or *links*, between objects that exchange messages), and generation of sequence diagrams, thus facilitating the debugging and refinement of the original UML diagrams.

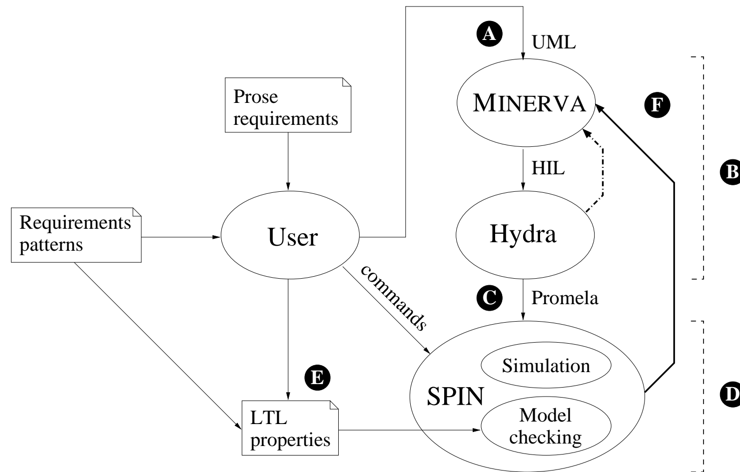


Fig. 2. Overview of our approach

4 Modeling and Analyzing a Diesel Filter System

This section describes how we applied our modeling and analysis process to an automotive application obtained from one of our industrial partners, Detroit Diesel. Specifically, we depict an embedded system controlling a self-cleaning particulate filter that reduces the amount of pollutants emitted from the exhaust of diesel trucks. We illustrate how requirements patterns can be used to guide the creation of a system model and how MINERVA and Hydra enable simulation and model checking with SPIN.

4.1 Application Overview

An effective way to reduce particulate combustion aerosols, or soot, from diesel truck exhaust is to use particulate filters placed in a canister and inserted into the exhaust gas path. A filter comprises several tubes, with each tube consisting of ceramic fibers wound around a metallic cylindrical grid. Exhaust gas flows through the filters, out of the canister, and into the exhaust pipe. To enable the exhaust gas to flow freely through the filters, they must be cleaned periodically. Therefore, the grid wires can be electrified, causing them to heat up and burn off trapped particulates. The Diesel Filter System (DFS) is an embedded system that initiates a cleaning cycle when the differential pressure across the filter canister, as measured in *Pascals* (Pa), is within an acceptable range. The grid heating sequence will not begin if too few engine revolutions have occurred since the last time the cleaning cycle was completed, or the current engine revolutions per minute (RPMs) are too low.

4.2 Requirements Patterns for the Diesel Filter System

We present four requirements patterns that we identified to be appropriate for this system based on the DFS requirements [22]: *Actuator-Sensor*, *Fault Handler*, *Watchdog* and *User Interface* Patterns. Figure 3 illustrates how the information in the **Structure** section of the *Actuator-Sensor*, *Fault Handler*, *Watchdog*, and *User Interface* Patterns can be used to guide the creation of a preliminary UML class diagram for the DFS. The ComputingComponent, shown in bold in Figure 3, plays a role in all four patterns.

Actuator-Sensor **Pattern:** The *Actuator-Sensor* Pattern, denoted by dashed boxes and lines, shows how abstract sensor and actuator classes are used to give a common interface to the concrete sensors (CurrentMirror's, PressureSensor) and actuators (DriverDisplay, HeaterRegulator's) in the DFS.

Fault Handler **Pattern:** The FaultHandler, illustrated with a dash-dotted box and lines, controls the ComputingComponent to initiate safety actions when errors occur. It also controls the UserInterface, warning the user that errors have occurred.

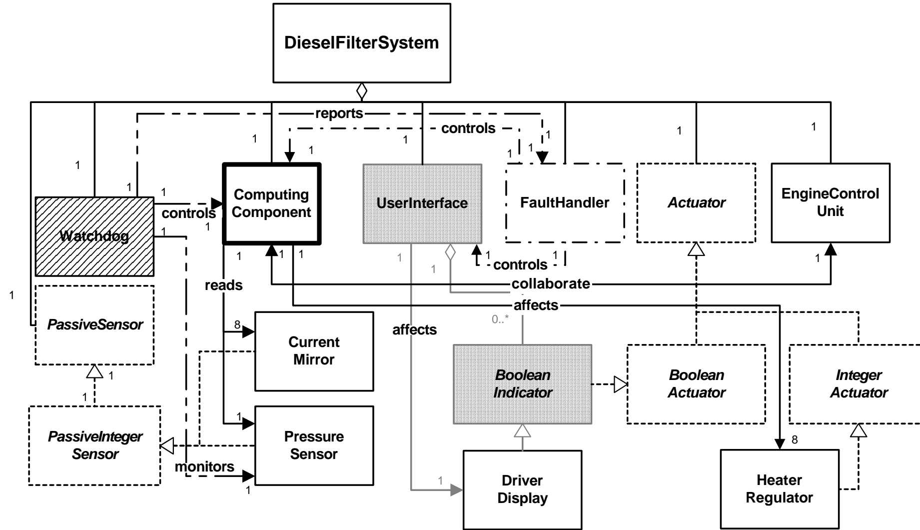


Fig. 3. Requirements-Pattern-Guided UML class diagram of the Diesel Filter System

***Watchdog* Pattern:** The Watchdog, denoted by a striped box and long-short-short dashed lines, monitors the PressureSensor. If it detects a violation of the maximum pressure value, then it notifies the FaultHandler of the error and initiates an emergency shutdown in the ComputingComponent.

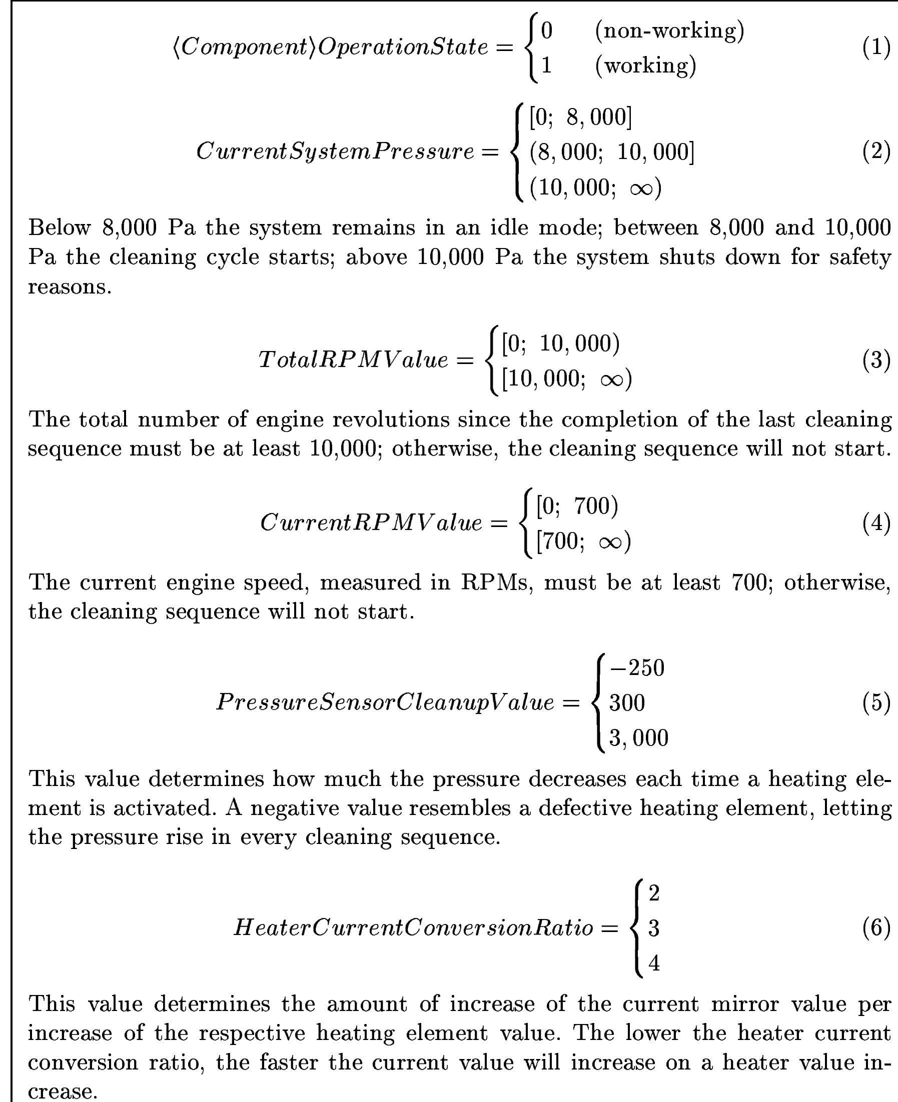
***User Interface* Pattern:** The *User Interface* Pattern is represented by the shaded boxes and lines. The UserInterface controls only one boolean indicator, the DriverDisplay, which represents a simple warning device such as an indicator light.

4.3 Abstraction and Equivalence Classes

Abstraction can significantly reduce the state space needed to perform model checking; we use two techniques. First, we model only those portions of the system that are relevant to our focused analysis. In this study, we are interested in specifying and analyzing the DFS cleaning cycle. We model only those components relevant to this analysis. Additionally, we also abstract the number of heater regulators and their corresponding current mirrors from eight in the actual system down to two in our model.

Second, we determine *equivalence classes* for the possible values of system conditions. Generally, the operational status of a component is represented as *non-working* (false) or *working* (true), as shown in Expression (1) in Figure 4. We model the operational status of the PressureSensor, HeaterRegulator1, and HeaterRegulator2. Ranges for other monitored values (*e.g.*, current system pressure, number of revolutions of the engine since the last cleaning cycle, current engine speed) can be determined from the requirements, as shown in Expressions (2), (3), and (4) in Figure 4 (∞ represents the target language-dependent

upper bound). We also introduce physical abstraction values for modeling purposes (Figure 4, Expressions (5) and (6)). These values represent the interaction between components due to existing physical relationships (*e.g.*, how much the current pressure decreases after every successful cleaning cycle in the DFS).



$$\langle Component\rangle OperationState = \begin{cases} 0 & \text{(non-working)} \\ 1 & \text{(working)} \end{cases} \tag{1}$$

$$CurrentSystemPressure = \begin{cases} [0;\ 8,000] \\ (8,000;\ 10,000] \\ (10,000;\ \infty) \end{cases} \tag{2}$$

Below 8,000 Pa the system remains in an idle mode; between 8,000 and 10,000 Pa the cleaning cycle starts; above 10,000 Pa the system shuts down for safety reasons.

$$TotalRPMValue = \begin{cases} [0;\ 10,000) \\ [10,000;\ \infty) \end{cases} \tag{3}$$

The total number of engine revolutions since the completion of the last cleaning sequence must be at least 10,000; otherwise, the cleaning sequence will not start.

$$CurrentRPMValue = \begin{cases} [0;\ 700) \\ [700;\ \infty) \end{cases} \tag{4}$$

The current engine speed, measured in RPMs, must be at least 700; otherwise, the cleaning sequence will not start.

$$PressureSensorCleanupValue = \begin{cases} -250 \\ 300 \\ 3,000 \end{cases} \tag{5}$$

This value determines how much the pressure decreases each time a heating element is activated. A negative value resembles a defective heating element, letting the pressure rise in every cleaning sequence.

$$HeaterCurrentConversionRatio = \begin{cases} 2 \\ 3 \\ 4 \end{cases} \tag{6}$$

This value determines the amount of increase of the current mirror value per increase of the respective heating element value. The lower the heater current conversion ratio, the faster the current value will increase on a heater value increase.

Fig. 4. Equivalence classes for system conditions

4.4 UML Modeling for the Diesel Filter System

Based on Figure 3 and our abstractions, we created UML object and state diagrams to model the DFS. Figure 5 overviews the UML object diagram for the DFS (attributes and methods have been elided). Components retain their shading/line characteristics from Figure 3. The ComputingComponent, the core of the system, reads values from the sensors PressureSensor, CurrentMirror1, and CurrentMirror2, and the EngineControlUnit. It also sets the values of the actuators HeaterRegulator1 and HeaterRegulator2. The PressureSensor senses the current pressure. The EngineControlUnit models an interface to the engine controller to check the current engine speed (RPMs) and the total number of revolutions since the last cleaning cycle. Each CurrentMirror senses the amount of electrical current flowing through its respective HeaterRegulator. The FaultHandler processes error messages received and takes appropriate actions (defined in the FaultHandler state diagram). The Watchdog monitors the PressureSensor, notifying the FaultHandler and shutting down the ComputingComponent if the pressure exceeds 10,000 Pa. The UserInterface controls the DriverDisplay, which represents a simple warning light. Additionally, our approach incorporates two special classes, an Environment class that defines the equivalence classes for system conditions of the environment as depicted in Figure 4, and a _SYSTEMCLASS_ class that represents the aggregation of the main components of the system and non-deterministically selects values for the system conditions.

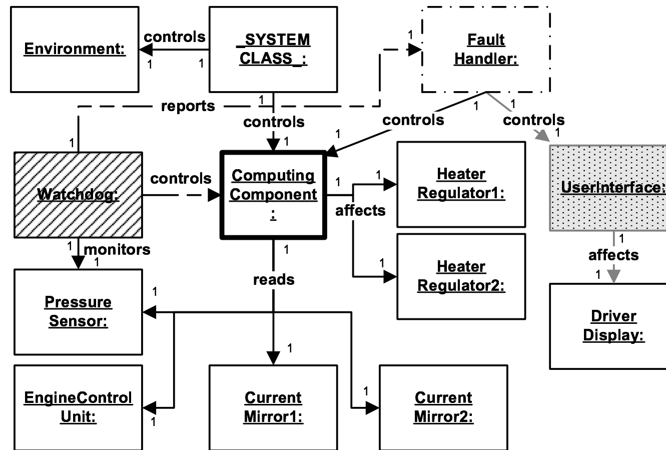


Fig. 5. UML object diagram of the abstracted Diesel Filter System

In our approach, each component has its own state diagram; however, due to space constraints, we show only the (elided) state diagram of the ComputingComponent, the central component of the DFS, in Figure 6. The structure of this state diagram follows that of the state diagram given in the **Behavior** section

of the *Fault Handler* pattern [14]. Specifically, it has the state *PowerOff* and the composite states *Initialize* and *NormalBehavior* (elided in Figure 6). Furthermore, the three states *GetPressure1*, *GetPressure2*, and *Idle* represent the Idle mode of the DFS where the system continuously queries the PressureSensor and initiates a cleaning cycle if the pressure is found to exceed 8,000 Pa. (The dashed and bolded transitions and the italicized elements will be described in the next section as part of the analysis process.)

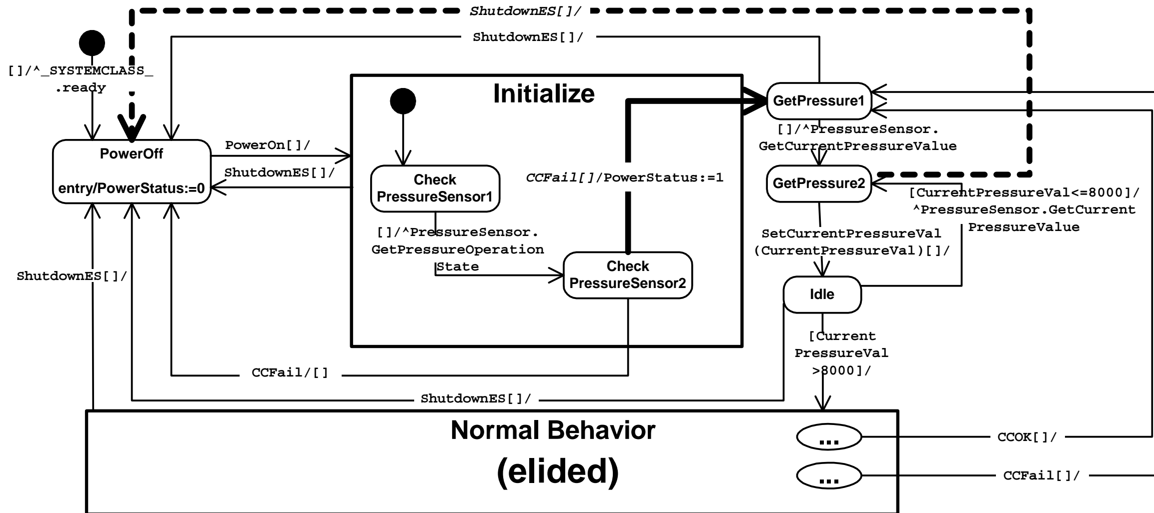


Fig. 6. UML state diagram of the ComputingComponent (elided)

The DFS performs three main steps. First, on system activation, the DFS enters an *Initialization* phase. If the initialization is performed successfully, then the system enters an *Idle* phase. While in the *Idle* phase, the system continually checks the current system pressure. If a failure occurs during the initialization, then the system shuts down.

Second, if the differential pressure in the filter container exceeds 8,000 Pa, then the cleaning cycle is started. At the beginning of the cleaning cycle, the system waits for the total number of revolutions since the last cleaning cycle and the current RPMs to pass their thresholds of 10,000 and 700, respectively. In a cleaning sequence, each operational heater element is ramped up to burn off trapped particulates and ramped down afterwards. During the ramp-up process of each heater element, the system monitors the current on the corresponding current mirror to detect excess conditions and accordingly ramps down the heating element.

Third, after the completion of the cleaning cycle the DFS returns to the *Idle* phase, waiting for either the pressure to exceed 8,000 Pa again or a system

shutdown message to arrive.

4.5 Analysis Using Requirements and Specification Patterns

After we use MINERVA to construct UML diagrams of the system, we use Hydra to generate an executable specification of the system in terms of Promela (not shown due to space constraints; see [13]). Briefly, objects are captured as `proctypes` that communicate via `channels` using queueing semantics [4, 15]. In this section, we examine three requirements for the DFS. In each case, we give the prose requirement, the relevant requirements pattern(s) in bold italics, the relevant specification-pattern-based constraint(s) from the **Constraints** section of each requirements pattern, the instantiated constraints checked against the generated Promela specification, and the analysis results (including visualizations).

Requirement 1: It should never be the case that the PressureSensor is non-operational and yet the system power is on.

Fault Handler *Constraint:* If system initialization fails, then the system should remain in a powered-off state.

```
[] ( !( ``Initialization failed'' && ``System power is on'' ) )
```

Instantiated Constraint: In our DFS model, system initialization has been abstracted to checking the operational status of the PressureSensor, represented by the attribute `PressureOperationState`. Its possible values are *zero* (not working) and *one* (working). The power to the ComputingComponent, the core of the system, is represented by the attribute `PowerStatus`. Its possible values are *zero* (off) and *one* (on).

```
[] ( !( ( PressureSensor.PressureOperationState==0 )
          && ( ComputingComponent.PowerStatus==1 ) ) )
```

Analysis Results: SPIN detected a counterexample. MINERVA generated a sequence diagram[2] from the counterexample in terms of the UML objects in our model. However, the sequence diagram (not shown) indicated only that the ComputingComponent began its initialization phase by querying the PressureSensor operational status, and received a *CCFail* message indicating that the PressureSensor was not working. Without more information, it is difficult to determine the cause of the error. MINERVA can also animate the UML state diagrams based on trace data from the SPIN counterexample. State diagram animation of the entire model revealed that the ComputingComponent unexpectedly reached (and became deadlocked in) state *GetPressure2*. Figure 7 shows, in human-readable form generated by MINERVA, only those animation steps pertaining to the ComputingComponent state diagram (Figure 6).

[2] MINERVA's sequence diagram illustrates the order in which messages were sent between UML objects during a particular trace. In contrast, SPIN generates Message Sequence Charts (similar to sequence diagrams) that depict message communication at the level of Promela processes, rather than UML objects.

1. Object “ComputingComponent” transitions from state “Initial” to state “PowerOff” on event “modelstart”
2. Object “ComputingComponent” transitions from state “PowerOff” to state “Initialize” on event “PowerOn”
3. Object “ComputingComponent” transitions from state “Initial” to state “CheckPressureSensor1” on event “modelstart”
4. Object “ComputingComponent” transitions from state “CheckPressureSensor1” to state “CheckPressureSensor2”
5. Object “ComputingComponent” transitions from state “CheckPressureSensor2” to state “GetPressure1” on event “CCFail”
6. Object “ComputingComponent” transitions from state “GetPressure1” to state “GetPressure2”

Fig. 7. Animation trace of the ComputingComponent state diagram (Requirement 1)

Corrective Actions: We determined that the problem was unintentional non-determinism in the ComputingComponent state diagram. The italicized event “CCFail” on the bold transition in Figure 6 unintentionally creates a non-deterministic situation when the ComputingComponent leaves state *CheckPressureSensor2* on a *CCFail* event. We changed the italicized event to the correct event, “CCOK”, in the state diagram and regenerated the specification. SPIN verified the claim after 2,616,630 transitions.

Requirement 2: If the Watchdog detects a violation, then the system should turn off.

Watchdog *Constraint:* If a violation of the system requirements is found, then the Watchdog should start the corresponding *recovery action* appropriate to the system being modeled (*e.g.*, begin error recovery, reset the device, shut down).

```
[] ( ‘‘Violation’’ -> <> ‘‘Start recovery action’’ )
```

Instantiated Constraint: The attribute `Violation` represents whether or not the Watchdog has detected a violation. Its possible values are *zero* (no violation has been detected) and *one* (a violation has been detected). The DFS powers off when a violation occurs.

```
[] ( ( Watchdog.Violation==1 ) ->
       <> ( ComputingComponent.PowerStatus==0 ) )
```

Analysis Results: SPIN detected a counterexample, from which MINERVA generated the (elided[3]) sequence diagram shown in Figure 8. In this diagram, the

[3] This sequence diagram has been elided in the following manner: (1) all interactions between the _SYSTEMCLASS_ or Environment classes and the other components of the system have been elided, including the initial *PowerOn* message sent to the ComputingComponent; and (2) the lifelines of all objects not participating in the message exchange depicted by the sequence diagram have been elided.

ComputingComponent queries the operational status of the PressureSensor and receives a *CCOK* message, indicating that the PressureSensor is working. It then requests the current system pressure. However, the PressureSensor notifies the Watchdog that the current system pressure exceeds 10,000 Pa. The Watchdog then sends a *ShutdownES* message to the ComputingComponent and an error code to the FaultHandler, indicating that a violation has been detected. The FaultHandler notifies the UserInterface, which then activates the DriverDisplay. Again, the sequence diagram does not provide enough information to determine the cause of the error. (The expected behavior of the ComputingComponent upon receiving a *ShutdownES* message is to power off.) State diagram animation of the entire model reveals that the ComputingComponent becomes deadlocked in state *GetPressure2* rather than returning to state *PowerOff*.

Corrective Actions: We determined that the problem was a missing transition in the ComputingComponent state diagram that unintentionally created a deadlock in the state *GetPressure2*. We added a transition from state *GetPressure2* to state *PowerOff* to handle event *ShutdownES* (depicted by the dashed transition in Figure 6) and regenerated the specification. SPIN verified the claim after 3,028,470 transitions.

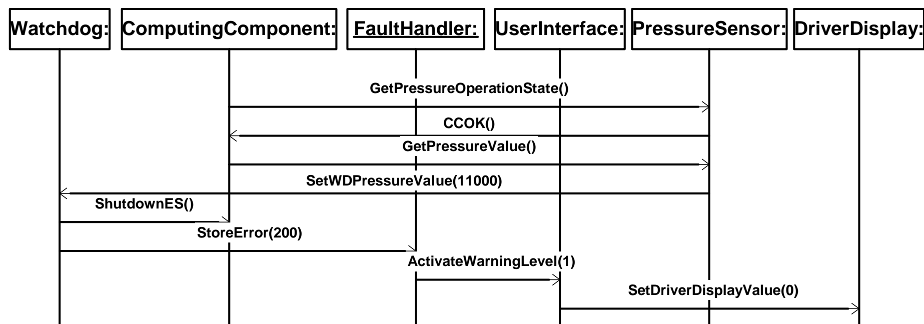


Fig. 8. Elided UML sequence diagram (Requirement 2)

Requirement 3: If the Watchdog detects a violation, then a warning light turns on. Constraints from several requirements patterns combine to specify this requirement. Upon detecting a violation, the Watchdog interacts with the FaultHandler. Upon receiving an error message, the FaultHandler interacts with the UserInterface. Finally, upon notification from the FaultHandler, the UserInterface takes appropriate action, in this case turning on a warning light. The three constraints are described below:

Watchdog *Constraint:* When a violation is found, a message containing the appropriate error code should be sent to the FaultHandler (indicated by the keyword **sent**).

```
[] ( ‘‘Violation’’ ->
     <> ( sent( FaultHandler.StoreError(ErrorMessage) ) ) )
```

Instantiated Constraint: We model only one type of violation, sending the error code "200" to the FaultHandler.

```
[] ( ( Watchdog.Violation==1 ) ->
       <> ( sent( FaultHandler.StoreError(200) ) ) )
```

Analysis Results: SPIN verified this claim after 2,362,780 transitions.

Fault Handler *Constraint:* When an error message is sent to the FaultHandler, it should activate the appropriate user interface warning level.

```
[] ( sent( FaultHandler.StoreError(Error) ) ->
     <> ‘‘Activate appropriate user interface warning level’’ )
```

Instantiated Constraint: We model only one type of error in our system, using the code "200". The possible values of the UserInterface attribute `WarningLevel` are *zero* (no warning) and *one* (warning).

```
[] ( sent( FaultHandler.StoreError(200) ) ->
     <> ( sent( UserInterface.ActivateWarningLevel(1) ) ) )
```

Analysis Results: SPIN verified this claim after 4,283,420 transitions.

User Interface *Constraint:* Upon receiving a warning, activate the appropriate indicator devices, such as turning on an alarm or a warning light.

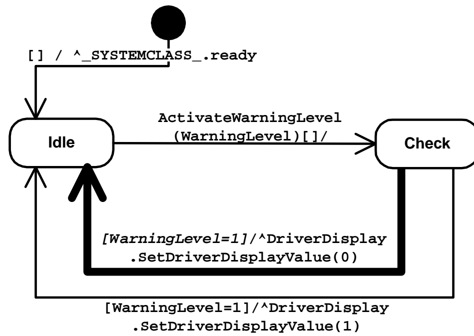
```
[] ( ( sent( UserInterface.ActivateWarningLevel(WarningLevel) ) ) ->
       <> ( ‘‘Activate appropriate indicators’’ ) )
```

Instantiated Constraint: In the modeled system, a light in the actuator DriverDisplay is represented by the attribute `DriverDisplayValue`. The possible values for this attribute are *zero* (the light is off) and *one* (the light is on).

```
[] ( ( sent( UserInterface.ActivateWarningLevel(1) ) ) ->
       <> ( DriverDisplay.DriverDisplayValue==1 ) )
```

Analysis Results: SPIN detected a counterexample, and MINERVA generated a sequence diagram (not shown). The messages of interest can also be seen as the last two messages in Figure 8. Although the UserInterface receives the message *ActivateWarningLevel(1)*, indicating a warning, it sends the message *SetDriverDisplayValue(0)* to the DriverDisplay, turning off the light. State diagram animation revealed that the problem was an erroneous guard on a transition in the UserInterface state diagram. The italicized guard on the bold transition in Figure 9(a) unintentionally creates non-determinism in transitioning from the *Check* to the *Idle* state, and erroneously allows the warning light to be turned off when it should instead indicate a warning to the user. Figure 9(b) shows, in human-readable form generated by MINERVA, only those animation steps pertaining to the UserInterface state diagram. The animation itself highlights in color the transition shown in bold in Figure 9(a), distinguishing which one of the transitions was taken.

Corrective Actions: We corrected the guard to compare the warning level to zero and regenerated the specification. SPIN verified the claim after 4,283,420 transitions.



(a) State Diagram

1. Object "UserInterface" transitions from state "Initial" to state "Idle" on event "modelstart"
2. Object "UserInterface" transitions from state "Idle" to state "Check" on event "ActivateWarningLevel(WarningLevel)"
3. Object "UserInterface" transitions from state "Check" to state "Idle" on condition "WarningLevel=1"

(b) Transition Trace

Fig. 9. Animation trace of UserInterface state diagram (Requirement 3)

5 Conclusions

Preliminary feedback from industrial collaborators indicate that requirements patterns can be an effective mechanism for describing requirements of embedded systems. Furthermore, by adding specification-pattern-based properties to the **Constraints** field of the requirements pattern template, developers have some guidance as to what kinds of properties can be checked for a given system when a particular pattern is applied. The specification patterns used in conjunction with requirements patterns enable even novice developers to easily formulate claims to check the system for specific constraints. Then using our formalization work and tool suite, developers have a mechanism to rigorously check the requirements using simulation and model checking techniques.

Several major directions for future research are motivated by the validation results obtained thus far. First, we are continuing to identify additional requirements patterns. In particular, we are investigating patterns that address timing issues for embedded systems. Accordingly, our formalization framework is being extended to handle models that contain timing information. Second, additional specification patterns are being sought that capture typical or common constraints applicable to embedded systems. Finally, in order to keep the model checking portion of the analysis tractable, we are exploring numerous abstraction techniques, similar to those used by Heitmeyer *et al.* [1] and others, to reduce the scope of a given analysis scenario.

References

1. Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37–68, January 1999.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addision-Wesley, 1999.
3. Laura A. Campbell, Betty H. C. Cheng, William E. McUmber, and R. E. K. Stirewalt. Automatically detecting and visualizing errors in UML diagrams. *Requirements Engineering Journal*, 7(4):264–287, 2002.
4. Betty H. C. Cheng, Laura A. Campbell, Min Deng, and R.E.K. Stirewalt. Enabling validation of UML formalizations. Technical Report MSU-CSE-02-25, Department of Computer Science, Michigan State University, East Lansing, Michigan, September 2002.
5. Bruce Powell Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison–Wesley, 1999.
6. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings 2nd Workshop on Formal Methods in Software Engineering*, pages 7–16, Clearwater Beach, FL, March 1998.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. Wai Ming Ho, Jean-Marc Jezequel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. of IEEE International Conference on Automated Software Engineering*, Cocoa Beach, FL, October 1999.
9. Gerald J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
10. Honeywell. URL: www.htc.honeywell.com/dome.
11. I-logix. Rhapsody. URL: www.ilogix.com.
12. Sascha Konrad, Laura A. Campbell, and Betty H. C. Cheng. Adding formal specifications to requirements patterns. In *Proceedings of the Requirements for High Assurance Systems Workshop (RHAS02) as part of the IEEE Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002.
13. Sascha Konrad, Laura A. Campbell, Betty H. C. Cheng, and Min Deng. A requirements pattern-driven approach to specify systems and check properties. Technical Report MSU-CSE-02-28, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2002.
14. Sascha Konrad and Betty H. C. Cheng. Requirements patterns for embedded systems. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002.
15. William E. McUmber and Betty H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, May 2001.
16. William E. McUmber and Betty H.C. Cheng. UML-based analysis of embedded systems using a mapping to VHDL. In *Proceedings of IEEE High Assurance Software Engineering (HASE99)*, Washington, DC, November 1999.
17. William Eugene McUmber. *A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development*. PhD thesis, Michigan State University, August 2000.

18. Jorg Niere and Albert Zundorf. Using FUJABA for the development of production control systems. In *Applications of Graph Transformations with Industrial Relevance AGTIVE*, pages 181–191. Springer Verlag, 1999. Volume 1779, Lecture Notes in Computer Science.
19. Rational. Rational Rose. URL: www.rational.com.
20. Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.
21. Telelogic. ObjectGEODE. URL: www.telelogic.com.
22. Anthony Torre. Project specifications for diesel filter system, 2000. www.cse.msu.edu/~cse470/F2000/cheng/Projects/F00-Cheng/filter/Description/air-filter.html.