Requirements Patterns for Embedded Systems

Sascha Konrad and Betty H.C. Cheng *
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
Email: {konradsa,chengb}@cse.msu.edu

Abstract

In software engineering, design patterns propose solution skeletons for common design problems. The solution skeleton is described in such a way that the design can be used for other projects, where each application tailors the design to specific project constraints. This paper describes research into investigating how a similar approach to reuse can be applied to requirements specifications, which we term requirements patterns. Specifically, we explore how object-oriented modeling notations, such as the Unified Modeling Language (UML), can be used to represent common requirements patterns. Structural and behavioral information are captured as part of a requirements pattern. In order to maximize reuse, we focus on requirements patterns for embedded systems. This paper also describes case studies that illustrate how we have applied these general patterns to multiple embedded systems applications from the automotive industry.

1. Introduction

In recent years, many research and development efforts in software engineering have focused on the identification and use of design patterns. Given the detailed descriptions of commonly used design patterns captured by Gamma *et al.* [6], the software engineering community is becoming more aware of other types of patterns applicable to other parts of the software development process. Fowler [5] identified high-level analysis patterns that might be used to represent conceptual models of business processes, such as abstractions from accounting, trading, and organizational relationships. Geyer-Schulz and Hahsler [7] add more structure to their descriptions of analysis patterns and focus on the domain of cooperative work and collaborative applications. Gross and Yu [8] discuss the relationship between non-functional requirements and design patterns, and Robertson [16] discusses the use of event/use case modeling to identify, define, and access requirements process patterns. Sutcliffe *et al.* [20] describe how scenarios of use cases can be investigated to identify generic requirements for different application classes. Others have attempted to identify software architecture patterns [18], database access patterns [11], fault-tolerant telecommunication system patterns [1], patterns for distributed systems [19], design patterns for avionics control systems [14], etc.

This paper describes research into how an approach similar to design patterns can be applied to requirements specifications, which we term *requirements patterns*. Specifically, we explore how object-oriented modeling notations, such as the Unified Modeling Language (UML), can be used to represent common requirements patterns. Structural and behavioral information are captured as part of a requirements pattern. We use a slight variation of the patterns template developed by Gamma *et al.* [6], which we tailored to contain information more specific to requirements engineering.

In order to maximize reuse, we focus on requirements patterns for embedded systems. We note that *design patterns* have also been identified specifically for embedded systems [4], but they are focused on language-specific or communication-related issues. In addition, *architectural patterns* have a similarity to our requirements patterns in that both use diagrams to denote structural patterns; however, architectural patterns depict information more specific to detailed design and implementation. In contrast, requirements patterns can be used to capture specific functional or non-functional requirements of a system that can be refined with more design and implementation details, potentially involving the use of architectural and design patterns. Furthermore, we note that the integral role that the physical elements of an embedded system play dictates the need to

*Please contact B. Cheng for all correspondences.

specify the distinction between the hardware and software elements in the requirements patterns, including what functionality will be handled respectively [3]. In other contexts or domains, such a level of functional decomposition might be considered being architectural or design-level specific. As part of the validation effort, this paper also describes case studies that illustrate how we have generalized these patterns to apply to multiple automotive embedded systems applications.

The remainder of this paper is organized as follows: Section 2 gives a template for describing a pattern and overviews the patterns found so far. Section 3 gives detailed descriptions of two commonly used requirements patterns (the complete description of patterns may be found elsewhere [12]). Section 4 describes how we applied the patterns to two different applications from the automotive domain. Finally, conclusions and future work are discussed in Section 5.

2. Describing Requirements Patterns

This section gives the template used to describe requirements patterns, enumerates the list of patterns, and presents a set of criteria for organizing and classifying the patterns.

2.1. Requirements Pattern Template

In contrast to other informal presentation styles for patterns, this paper uses a template similar in style to that used by Gamma *et al.* [6] in order to facilitate its understanding and application. The template uses problem frames [9] to describe the problem and its context, and UML diagrams are used to give structural and behavioral information. In the spirit of Ryan's work [17] on the use of natural language for requirements engineering, we use natural language to supplement diagrams in order to describe important aspects of the patterns as a mean for facilitating the understanding the requirements from different viewpoints.

The original design pattern template has been modified in several aspects to address the needs of requirements engineering. Specifically it has been extended with "Constraints", "Behavior" and "Design Patterns". The sections "Implementation" and "Sample Code" have been removed because they were too specific to software design and implementation. The "Behavior" section contains sequence and state diagrams that illustrate sample behavior. In contrast, the "Collaborations" section contains a textual description of the general behavior that is possible among collaborators within a pattern.

Pattern Name and Classification: The pattern name consists of a description of the pattern; the classifi cation provides the purpose of the pattern.

Intent: A brief description of the problems that the pattern addresses.

Motivation: A description of sample goals and objectives of a system that motivate the use of the pattern Problem frame diagrams are used to provide context for the problem specifi ed by the requirement of the pattern. Furthermore, use-cases and use-case-diagrams describe the goals of the pattern application. (These diagrams are not included in the examples due to space constraints.)

Constraints: Restrictions that are applied to the system. For example, constraints can elucidate the system's special hardware constraints and timing restrictions. Any environmental, domain, or implementation-imposed constraints should be included here; these may be functional or non-functional by nature. Safety should also be viewed in this section due to its importance in embedded systems.

Applicability: Provides a description concerning the conditions in which the pattern may be applied.

Structure: A representation of the classes and their relationships depicted in terms of UML class diagrams.

Behavior: Provides an illustrative representation of scenarios for class and object interaction. Also gives a description of the behavior of the pattern by using UML state and sequence diagrams.

Participants: Itemizes the classes/objects that are included in the requirements pattern and their responsibilities.

Collaborations: Describes how objects and classes interact, as well as their roles for carrying out various responsibilities.

Consequences: How are the objectives supported by a given pattern? Using the pattern, what are the trade-offs and outcomes?

Design Patterns: Applicable design patterns that can be used to refi ne the requirements patterns.

Also Known As: Lists alternative names for the Requirements Pattern.

Related Requirements Patterns: What requirements patterns are related to this one? What are the advantages and shortcomings of this pattern compared to different ones and which pattern should be used?

2.2. Requirements Patterns Catalogue Overview

Below is an enumeration of the requirements patterns that have been identified from analyzing several embedded systems.

- **Controller Decompose:** How to decompose an embedded system into different components according to their responsibilities.
- **Actuator-Sensor Pattern:** How to specify various kinds of sensors and actuators in an embedded system.
- **Watchdog Pattern:** How to monitor a device and initiate corrective action if it does not behave properly.

- **Examiner Pattern:** How to monitor a device and store occurring errors.
- **Fault Handler Pattern:** How to integrate fault handling functionality into an embedded system.
- **Mask Pattern:** 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 Pattern:** How to provide an interface to support decoupling of complex subsystems.
- **User Interface Pattern:** How to specify a user interface that is extensible and reusable.
- **Channel Pattern:** How to arrange communication between two components.
- **Monitor-Actuator Pattern:** How to increase safety and reliability by monitoring actuator behavior for errors.

3. Example Requirements Patterns

The following is an abbreviated description of two commonly used requirements patterns found in automotive embedded systems development. The names of requirements patterns are denoted in *italics*, and the elements of a requirements pattern are named in a san serif font, including class and state names.

***Actuator-Sensor*: Structural Pattern**

Intent:
How to specify various kinds of sensors and actuators in an embedded system.

Motivation:
Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to the control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The *Actuator-Sensor* Pattern is using a *pull* mechanism (explicit request for information) for PassiveSensors and a *push* mechanism (broadcast of information) for the ActiveSensors. Additional information about *pull* and *push* mechanism can be found elsewhere [13].

Figure 1 shows the problem frame diagram for the *Actuator-Sensor* Pattern. This pattern can be used to build the *Input-/Output-Modeler*, meaning this *Input-/Output Modeler* translates the data of the actuators and sensors from the environment into a model for which the system was designed. For example, a value of a temperature sensor is transformed into a five-digit integer number or a transformation from Celsius to Fahrenheit. Explicitly, the diagram shows the requirement *ComputingComponent - Model* in a dashed circle, from which arrows point to the *Input-/Output Model* and the *ComputingComponent* domain, meaning that the requirement constrains both of those domains because data can flow in both directions. The *ComputingComponent* receives data from the sensors and sends data to the actuators. The machine domain *Input-/Output Modeler* is the machine that has to be built for the *ComputingComponent* to access data in the *Input-/Output Model* domain. The latter one is a designed domain; it has to be designed by the developer of the system who is free to define the data structures.

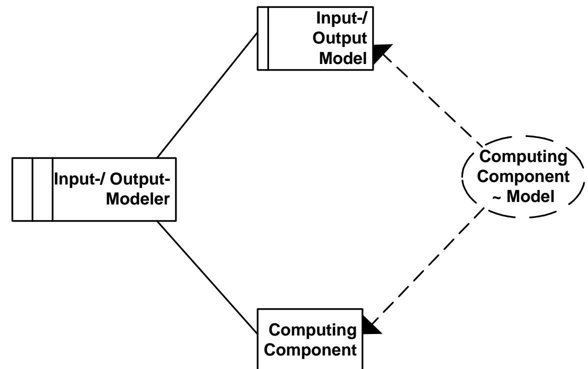

Figure 1. Problem Frame Diagram of the *Actuator-Sensor* Pattern

Constraints:

- Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.
- Each active sensor must have capabilities to broadcast update messages when its value changes.
- Each active sensor should send a *life tick*, a status message issued within a specified time frame, to detect malfunctions.
- Each actuator must have some method to invoke the appropriate response determined by the ComputingComponent.
- Each sensor and actuator should have a function implemented to check its own operation state.
- Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

Applicability:
Actuator-Sensor pattern is applicable

- in all embedded systems, especially when many actuators and sensors are present.

Structure:
A UML class diagram for the *Actuator-Sensor* Pattern can be found in Figure 2. Actuator, PassiveSensor and ActiveSensor are abstract classes and denoted in italics. There are four different types of sensors and actuators in this pattern. The boolean, integer and real classes represent the most common types of sensors and actuators. The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device. Nonetheless, these devices should

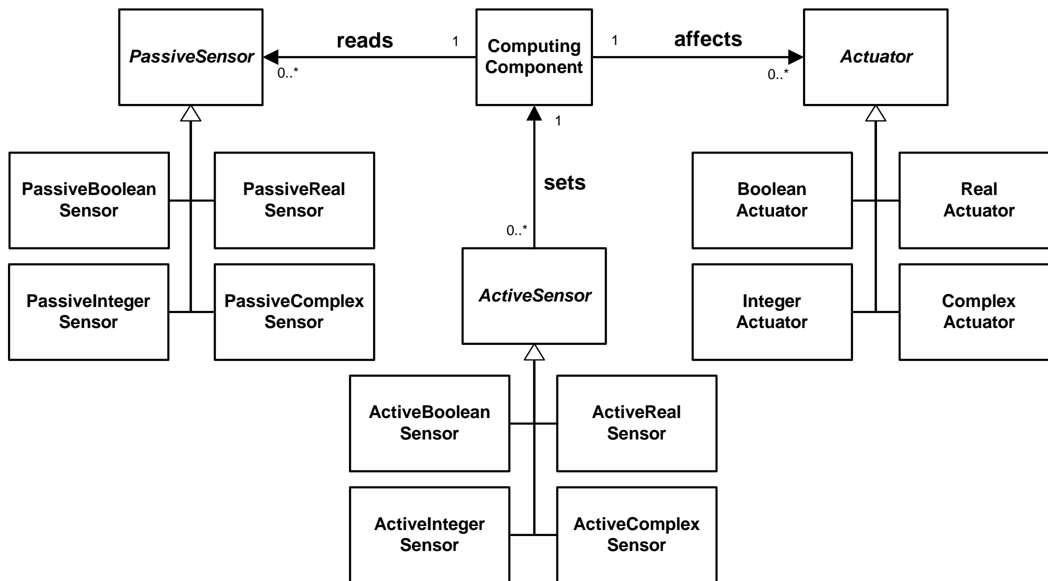

Figure 2. UML class diagram of the *Actuator-Sensor* Pattern

still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

Behavior:

Figure 3 shows a UML sequence diagram for an example of the *Actuator-Sensor* Pattern in a climate control system. Here the ComputingComponent queries a sensor (a passive temperature sensor) and an actuator (one radiator valve) to check the operation state for diagnostic purposes before reading or setting a value. The messages "Set Physical Value" and "Get Physical Value" are not messages between objects, instead, they describe the interaction between the physical devices of the system and their software counterparts. In the lower part of the diagram, below the horizontal line, the TemperatureSensor reports that the operation state is zero. The ComputingComponent then sends the error code for a temperature sensor failure to the FaultHandler that will decide how this error affects the system and what actions are required.

Participants:

- **PassiveSensor {abstract}**: Defines an interface for passive sensors.
- **PassiveBooleanSensor:** Defines passive boolean sensors.
- **PassiveIntegerSensor:** Defines passive integer sensors.
- **PassiveRealSensor:** Defines passive real sensors.
- **ActiveSensor {abstract}**: Defines an interface for active sensors.
- **ActiveBooleanSensor:** Defines active boolean sensors.
- **ActiveIntegerSensor:** Defines active integer sensors.
- **ActiveRealSensor:** Defines active real sensors.
- **Actuator {abstract}:** Defines an interface for actuators.
- **BooleanActuator:** Defines boolean actuators.
- **IntegerActuator:** Defines integer actuators.
- **RealActuator:** Defines real actuators.
- **ComputingComponent:** The central part of the controller; it gets the data from the sensors and computes the required response for the actuators.
- **ActiveComplexSensor:** Complex active sensors have the basic functionality of the abstract ActiveSensor class, but additional, more elaborate, methods and attributes, need to be specified.
- **PassiveComplexSensor:** Complex passive sensors have the basic functionality of the abstract PassiveSensor class, but additional, more elaborate, methods and attributes need to be specified.
- **ComplexActuator:** Complex actuators also have the base functionality of the abstract Actuator class, but additional, more elaborate methods and attributes need to be specified.

Collaborations:

- When the ComputingComponent needs to update the value of a PassiveSensor, it queries the sensors, requesting the value by sending the appropriate message.
- ActiveSensors are not queried. They initiate the transmission of sensor values to the computing unit, using the appropriate method to set the value in the ComputingComponent. They send a life tick at least once during a specified time frame in order to update their timestamps with the system clock's time.
- When the ComputingComponent needs to set the value of an actuator, it sends the value to the actuator.
- The ComputingComponent can query and set the operation state of the sensors and actuators using the appropriate methods. If an operation state is found to be zero then the error is sent to the FaultHandler who is responsible for handling

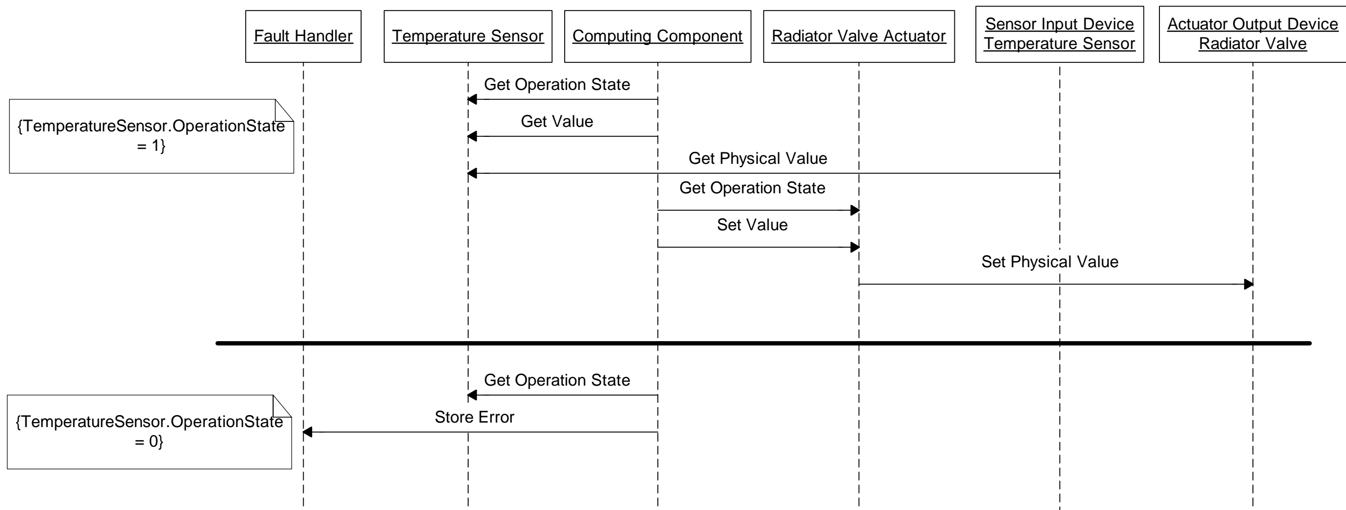

Figure 3. UML sequence diagram example of the *Actuator-Sensor* Pattern

error messages, such as starting a more elaborate recovery mechanism or a backup device. If no recovery is possible, then the system can only use the last known value for the sensor or the default value.

- The ActiveSensors offer methods to add or remove the addresses or address ranges of the components that want to receive the messages in case of a value change.

Consequences:

1. Sensor and actuator classes have a common interface.
2. Class attributes can only be accessed through messages and the class decides whether or not to accept the message. For example, if a value of an actuator is set above a maximum value, then the actuator class may not accept the message, or it might use a default maximum value.
3. The complexity of the system is potentially reduced because of the uniformity of interfaces for actuators and sensors.

Design Patterns:

- ***Factory Method* Design Pattern [6]:** This pattern and related ones can be used to handle the object creation of the actuators and sensors.
- ***Observer* Design Pattern [6]:** Use this pattern for active sensors to notify dependents if the sensor values change.
- ***Feature Coordination* Design Patterns [4]:** These patterns describe different strategies to handle message sequences.

Also Known As:

To be determined.

Related Requirements Patterns:

- ***Controller Decompose* Pattern:** The Actuator, Sensor and ComputingComponent relation can also be found in this pattern and it refers to the *Actuator-Sensor* Pattern in its specification.
- ***Channel* Pattern:** This pattern shows several ways to increase safety by using different strategies to add sensors and actuators to a system.
- ***Monitor-Actuator* Pattern:** The *Monitor-Actuator* Pattern shows how to use redundant sensors to add a monitor channel.

Fault Handler: Behavioral Pattern

Intent:

How to integrate a fault handler into an embedded system.

Motivation:

Fault handling is essential for embedded systems. Embedded systems frequently need to determine what responses are necessary to recover from errors. Consider a flight control system in an airplane, where the system should never shut down completely in response to an error. The system has to decide if the error is sufficiently severe to perform a partial shutdown and offer basic functionality, or if the error is not a threat and logging the error is sufficient. This fault handler must offer the possibility for other devices to read the error log. But it should also have access to the user interface to signal to the user that errors have occurred. An important function of the fault handler is to send the system into different states depending on the severity of the error. These mechanisms have to be implemented in the computing unit, for example, the operation for performing an emergency stop. If an error is reported to the fault handler justifying this action, then it will send this message to the ComputingComponent.

Therefore this fault handler acts as a centralized coordinator for safety monitoring and hence control of system recovery. The following inputs are usually captured [3]:

- Timeout message by the *Watchdog* or *Examiner Pattern*.
- Assertions of software errors.
- Built-in-tests (BITs) that run on a periodic or continuous basis.

- Faults identified by monitors, such as that described for the *Monitor-Actuator* Pattern.

The centralized safety control facilitates the verification and validation of the safety measures and eases the reuse of the fault handler in different systems.

Due to space constraints the problem frame diagram is not included [13].

Constraints:

- The fault handler should be hardware implemented if there are performance constraints and it is not likely to change.
- The fault handler should be protected against against corruption in environments where intensive interference occurs, for example radiation.
- Every possible error message should be classified and, depending on the error severity, safety actions should be defined for each error.
- A user interface should be present to signal errors.
- Hardware and software used in this pattern should have proven its reliability in the past; new technology has to be applied very carefully.

Applicability:
Fault Handler Pattern is applicable

- in any embedded system that does not have the need for distributed fault handling.

Structure:
The UML class diagram of the *Fault Handler* Pattern can be seen in Figure 4. The FaultHandler sends messages to the UserInterface to activate warning levels and sends the ComputingComponent into different safety states. For every safety state defined in the requirements, an operation in the ComputingComponent is needed. The safety states are listed in the Behavior Section.

The FaultHandler also receives error messages from watchdogs and monitors, and decides if recovery devices should be activated or what other actions are required. The Device is representative for all possible devices in the system. These devices send error messages to the FaultHandler. These devices can be monitored by watchdogs and/or monitors. They also report error messages to the fault handler. Depending on the safety measures and policies, the fault handler decides what to do, for example, activating the FailSafeDevice.

Behavior:
Figure 5 shows the state diagram of the ComputingComponent of the *Fault Handler* Pattern. The state diagram shows which states are possible and what messages activate them. Not all of the states are needed in every system, for example ABS systems generally do not have a partial shutdown state because the system constraints usually require that an inactive system should not affect the functioning of the brakes, although there is no longer skid prevention. Therefore, in this case, an emergency stop state where the ABS system shuts off power immediately is sufficient. These states have to be defined in the ComputingComponent. The FaultHandler decides, when an error occurs, which state is

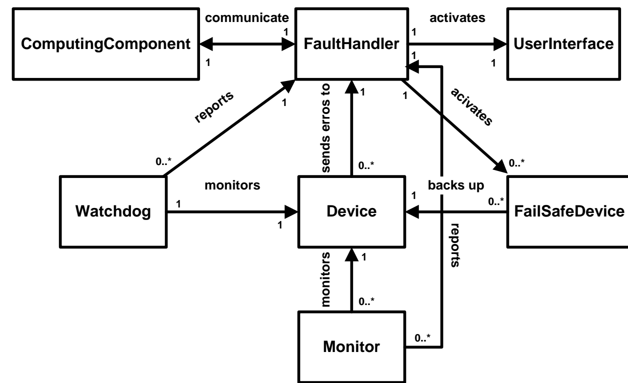

Figure 4. *Fault Handler* **Pattern**

appropriate and sends to the ComputingComponent the appropriate message to activate the corresponding state. It also activates the user interface to notify the user of the system if needed. The definitions for the states are as follows [3]:

- **Normal Behavior:** This state captures the system when no errors have occurred and it is working normally.
- **Manual/External:** In this state, the system is not controlled by a central component but from the outside, either from the user or another system, such as a diagnostic device.
- **Emergency Stop:** Here the system turns off power immediately, regardless of the current condition. This state is simple, but it is shown in the state diagram to illustrate the differences between the three types of stop states.
- **Production Stop:** This state is useful, for example, when a human enters a hazardous area. The system 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 useful, for example, when a machine needs to be stopped, but a cooling aggregate should continue to operate to avoid overheating.
- **Partial Shutdown:** The system only offers basic functionality; for example, 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.
- **Reset:** In this state the system initializes itself.
- **Power Off:** In this state, the system might be connected to a power supply, but is not yet activated. For example, a television set can operate in standby mode.

Participants:

- **FaultHandler:** Stores errors and decides which recovery actions are necessary. Contains safety measures and policies.
- **ComputingComponent:** Component controlling the system.

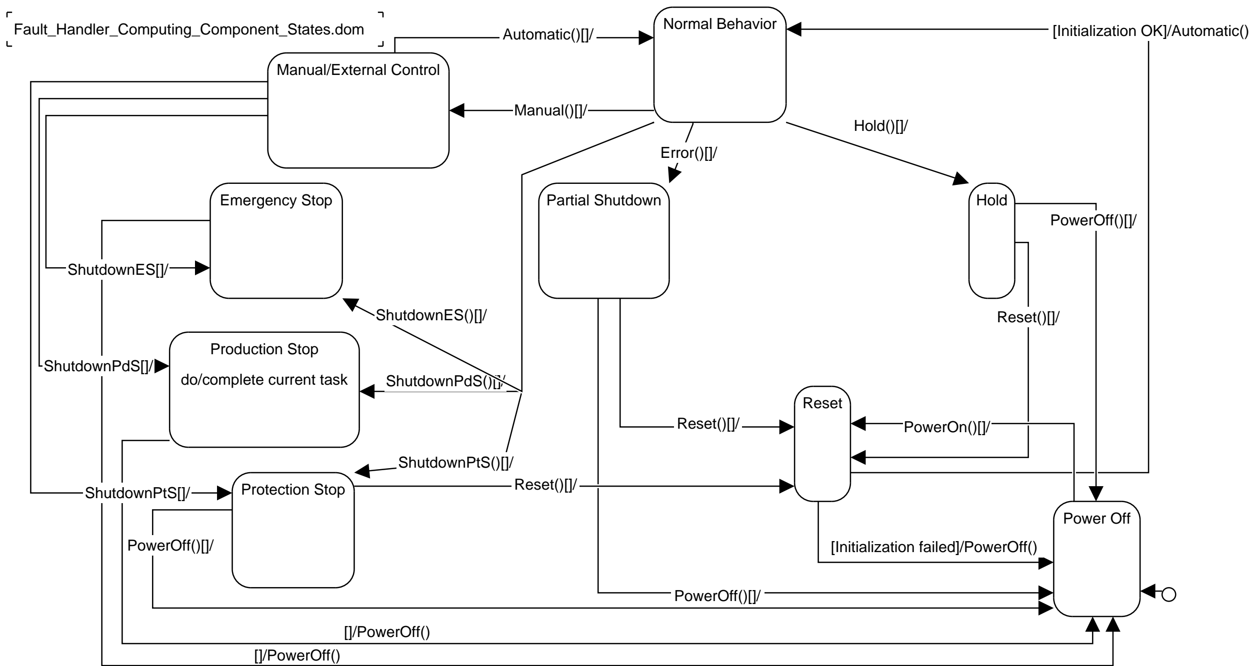

Figure 5. UML State Diagram of the ComputingComponent in the *Fault Handler* Pattern

- **UserInterface:** Class offering functionality to notify user about errors.
- **Device:** Device monitored by watchdog(s) and/or monitor(s).
- **Watchdog:** Watchdog monitoring the device.
- **Monitor:** Monitor monitoring device actuation.
- **FailSafeDevice:** Device activated in case of failure of the main device.

Collaborations:

- The FaultHandler receives all error messages and stores them in an error list.
- It also decides, depending on the safety measures and policies, if a fail-safe state should be entered, or whether the user interface or recovery device should be activated.
- Watchdog and Monitor monitor the device.
- FailSafeChannel is activated to recover from faults.

Consequences:

1. Required safety states should be implemented in the component.
2. Only one fault handler should exist in the system and should handle all error messages to avoid inconsistent handling of faults [3].
3. The fault handler becomes one of the most crucial parts in the system concerning safety issues, so the development and the testing should be thorough.
4. Hardware and software redundancies exist in the system, thus meaning higher system costs.
5. Overall safety of the system is usually signifi cantly improved.

Design Patterns:

- ***Singleton* Design Pattern [6]:** Use this pattern to assure only one fault handler exists in the system.

Also Known As:
To be determined.

Related Requirements Patterns:

- ***Controller Decompose* Pattern:** This requirements pattern defi nes the need for a fault handler in an embedded system.
- ***User Interface* Pattern:** This pattern can be used for the user interface described in the *Fault Handler* pattern.

4. Validation

This section briefly overviews how the patterns have been applied to two systems from the automotive domain, and it describes how these patterns affected the structure and the behavior of the systems. The first system is an *Adaptive Cruise Control* System [10]. In this system, radar was added to a standard cruise control system to provide automated support for collision avoidance when encountering slower targets. The class diagram of the system is shown in Figure

6. The second system is an *Electronically Controlled Steering* System that uses the current car speed, in an embedded system to continuously compute the amount of torque assistance needed from an electrical motor as the driver turns the steering wheel. The class diagram of the system can be found in Figure 7. The problem description, constraints, documents, and prototypes for both systems may be found elsewhere [21].

The two class diagrams look uniform in their structure, although the systems have completely different functionalities. The bold ComputingComponent in the diagrams is an essential part in both systems shown. The abstract classes PassiveSensor and Actuator, denoted by a dashed outline, can also be found in both systems. These classes come from the *Actuator-Sensor* Pattern that was applied to both systems. All sensors and actuators of the system inherit their interfaces from the classes in this pattern, which is illustrated by the dashed inheritance relationships.

The next pattern that can be found in both systems is the *Fault Handler* Pattern. The FaultHandler class, which is denoted by a dashed-dotted outline, is the central class responsible for fault handling in both systems. It receives error messages from the whole system and sends the ComputingComponent into the fail-safe states and activates the UserInterface if needed, which can be seen by the dash-dotted associations to both classes.

For the *Fault Handler* Pattern, both systems are analyzed for which fail-safe states are needed. For the Adaptive Cruise Control System, the Emergency Stop state is appropriate because the system shutdown does not affect the ability to operate the car safely. In the Electronically Controlled Steering System, a Production Stop state is used instead of an Emergency Stop state because the system should ramp down the steering assistance for two seconds and then perform a shutdown, otherwise the driver of the car could be so surprised by the sudden lack of assistance at the steering wheel that an accident could result. The resulting ComputingComponent state diagram for the Adaptive Cruise Control System is given in Figure 8, where the state diagram for the Electronically Controlled Steering System is exactly the same, except that instead of an Emergency Stop state, a Production Stop activity state is used with a rampdown activity.

Several other patterns have been applied to the systems, for example the *User Interface* Pattern. And the EngineControlModule in the Adaptive Cruise Control System is implemented using the *Moderator* Pattern, but due to space constraints, we are not able to include the descriptions here [12]. Using these patterns greatly facilitated the modeling of these systems, especially because many parts of the first system could be reused in the second one. Also, the patterns enable the developer to consider elements of the system early in the modeling process that otherwise would

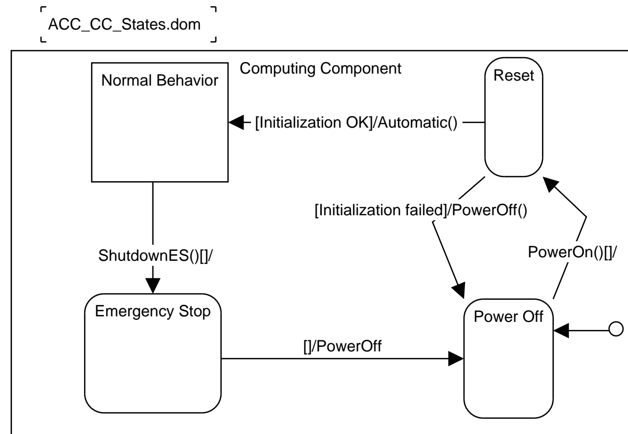

Figure 8. ComputingComponent **for Adaptive Cruise Control**

have been overlooked; for example, should a particular sensor be active or passive or how should the user interface be constructed. Upon visual inspection, it is clear that the class and state diagrams of the systems have a more uniform structure; and once a system is understood it is easier to develop future systems.

5. Conclusions

While this paper describes the early stages of our work into identifying requirements patterns for embedded systems, we have found these patterns to be useful in facilitating the requirements engineering process. We have made three general observations. First, due to the limited number of classes typically used in embedded systems, even having a small number of frequently used requirements patterns can greatly assist novices in the specification of fairly complete embedded systems. For example, most, if not all, embedded systems will use the *Actuator-Sensor* pattern [2]. There are other patterns to be specified that will likely be used by most embedded systems, such as those relating to timing. Second, the requirements patterns facilitate reuse at the design and code levels. For example, if a developer must have fault handling capabilities within a system, the *Fault Handler* pattern is useful for indicating the basic components of a fault handler. Each specific fault handler implementation will have different constraints on the actual implementation that will provide variations in functionality and costs. All of these implementations indexed by their functionality and costs can be associated with these requirements pattern for future use. Third, using requirements patterns enable a more uniform construction of system specifications, thus facilitating their understanding and maintenance.

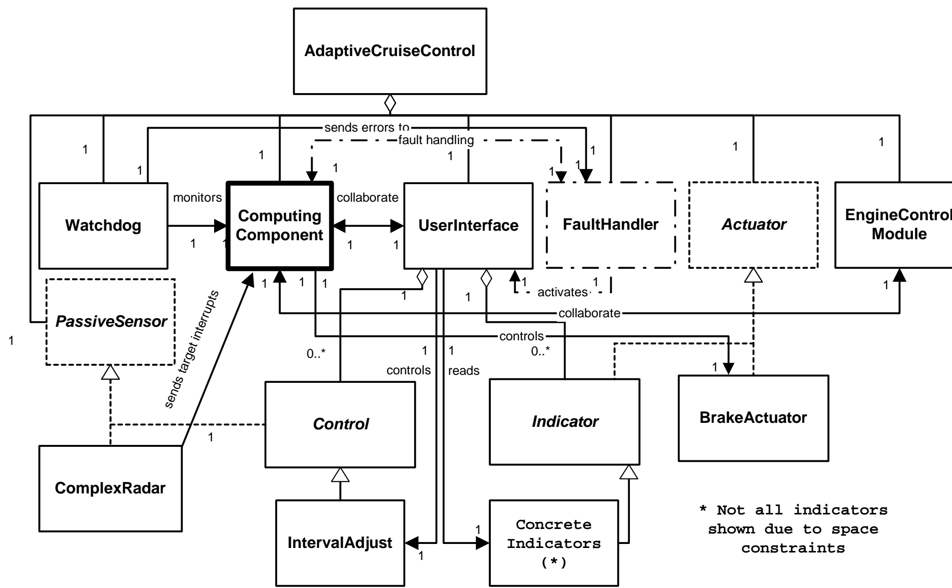

Figure 6. UML Class Diagram of the Adaptive Cruise Control System

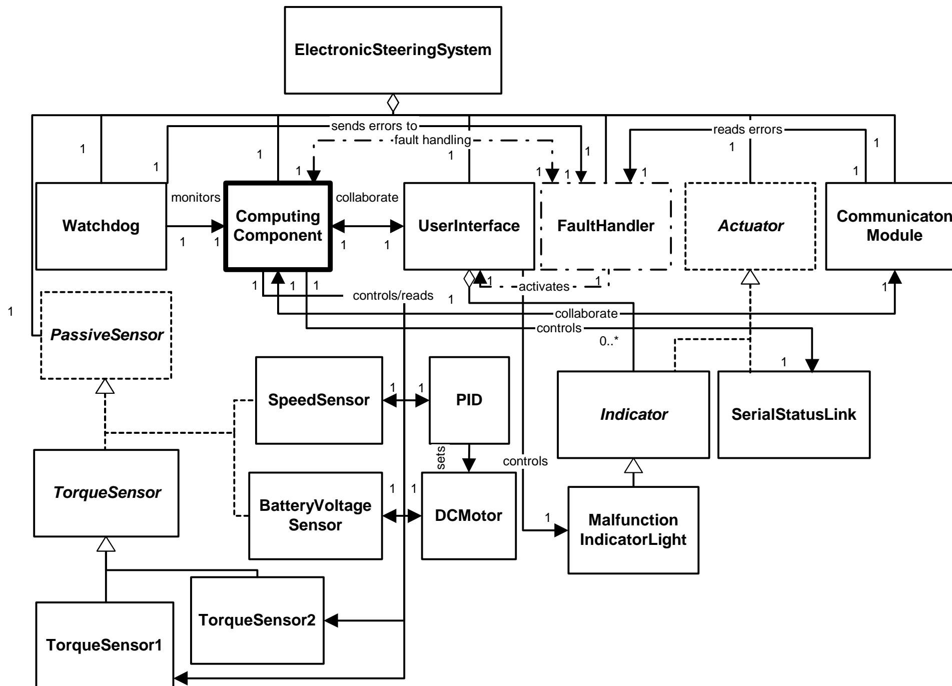

Figure 7. UML Class Diagram of the Electronically Controlled Steering System

Future work includes applying these patterns to more applications as well as expanding the repository of requirements patterns. Also, we plan to integrate this work with our research in formalizing UML diagrams [15] that will enable us to analyze the requirements patterns for various properties such as consistency within and between diagrams, introduce specification patterns into the template that will use model checking to establish concurrent processing properties, and use simulation to validate the behavior component of the requirements pattern specifications.

6 Acknowledgements

The authors are grateful to Erik Kamsties (University of Essen) and Laura Campbell and other members of the Software Engineering and Network Systems Laboratory at Michigan State University for helpful comments on this work. We also appreciate Tony Torre's (formerly at Siemens Automotive and now at Detroit Diesel Corp.) contributions of the automotive applications and answering our numerous questions.

This work is supported in part by NSF grants EIA-0000433, CDA-9700732, CDA-9617310, CCR-9633391, CCR-9901017, Department of the Navy, and Office of Naval Research under Grant No. N00014-01-1-0744. The first author completed a portion of this work while studying under an International Exchange Program through the University of Kaiserslautern.

References

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-tolerant telecommunication system patterns, 1995. http://www.bell-labs.com/user/cope/Patterns/PLoP95_telecom.html.

[2] B. H. Cheng and W. McUmber. CSE 470: Software Engineering, Fall 2001. http://www.cse.msu.edu/~cse470/F01/.

[3] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.

[4] EventHelix.com. Realtime mantra, 2001. http://www.eventhelix.com/RealtimeMantra.

[5] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

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

[7] A. Geyer-Schulz and M. Hahsler. Software engineering with analysis patterns, 2001. http://wwwai.wu-wien.ac.at/~hahsler/research/virlib_working2001/virlib/.

[8] D. Gross and E. S. K. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[9] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[10] W. D. Jones. Keeping cars from crashing. *IEEE Spectrum*, September 2001.

[11] W. Keller. Object/relational access layers - a roadmap, missing links and more patterns.

[12] S. Konrad. Identification, classification, and application of requirements patterns. Technical Report MSU-CSE-02-6, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2002.

[13] S. Konrad and B. H. Cheng. Requirements patterns for embedded systems. Technical Report MSU-CSE-02-4, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2002.

[14] D. Lea. Design patterns for avionics control systems. Technical Report ADAGE-OSW-94-01, 1994.

[15] W. E. McUmber and B. 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] S. Robertson. Requirements patterns via events/use cases, 1996.

[17] K. Ryan. The role of natural language in requirements engineering. *Proceedings of the IEEE Int. Symposium on RE, San Diego California*, pages 80–82, 1993.

[18] M. Shaw. Some patterns for software architectures. *Pattern Languages of Program Design 2 (J.Vlissides, J. Coplien, and N. Kerth eds.)*, pages 255–269, 1996.

[19] A. R. Silva. Dasco project - development of distributed applications with separation of concerns, 2000. http://www.esw.inesc.pt/~ars/dasco/.

[20] A. G. Sutcliffe, N. A. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *Software Engineering*, 24(12):1072–1088, 1998.

[21] A. Torre. Project specifications for adaptive cruise control system and electronically controlled steering system, 2000. http://www.cse.msu.edu/~cse470/F01/Projects/.