Abstract

Some of the most challenging tasks in building a software system are capturing, refining, and analyzing requirements. How well these tasks are performed significantly impacts the quality of the developed software system. The difficulty of these tasks is greatly exacerbated for the software of embedded systems, as these systems are commonly used for critical applications, have to operate reliably for long periods of time, and usually have a high degree of complexity. Current embedded systems software development practice, however, often deals with the (requirements) analysis phase in a superficial manner, instead emphasizing design and implementation. This research investigates how an approach similar to the well-known design patterns, termed object analysis patterns, can be applied in the analysis phase of embedded systems development, prior to design and coding. Specifically, our research explores how object-oriented modeling notations, such as the Unified Modeling Language (UML), can be used to represent structural and behavioral information as part of commonly occurring object analysis patterns. This work also investigates how UML-based conceptual models of embedded systems, based on the diagram templates in the object analysis patterns, can be automatically analyzed using the Spin model checker for adherence to properties specified in linear-time temporal logic (LTL) using a previously developed UML formalization framework. We have applied these patterns to several embedded systems applications obtained from the automotive industry. This paper describes one of our case studies and illustrates how our approach facilitates the construction of UML-based conceptual models of embedded systems and the analysis of these models for adherence to functional requirements.

Keywords:
Object-oriented modeling, embedded systems, requirements, patterns, conceptual modeling, object analysis, formal specification, model checking.
1. Introduction

Some of the most challenging tasks in building a software system are capturing, refining, and analyzing requirements [5, 47]. How well these tasks are performed significantly impacts the quality of the developed software system. Although the requirements of embedded systems are often fewer and more clearly defined than in other domains, real-time and embedded systems must function under strict operating constraints and must perform reliably for long periods of time [15], thus potentially increasing the complexity of developing such systems. To address this problem, development methodologies for embedded systems, such as ROPES [14] and COMET [24], commonly emphasize rapid turnaround, early proofs of correctness, and low risk. These development processes also advocate an analysis phase, where developers ultimately construct an object-based conceptual model of the system (i.e., an abstract, implementation-independent, object-based model that captures functional requirements) before committing to design and coding. For example, the conceptual model created in Douglass’s development process ROPES [14] contains both static and dynamic models that specify the correctness criteria for any solution created in the subsequent design phase of the development process. Within the context of ROPES’ analysis phase, this paper focuses on the description of patterns for use during the object analysis step to guide developers in creating object-based conceptual models of embedded systems, e.g., by providing structural and behavioral templates. Following the convention of being named after the phase of development in which they are applicable (e.g., analysis patterns [21], architectural patterns [46], design patterns [22]), we term these patterns object analysis patterns. We constructed an object analysis pattern template, much in the spirit of the template used by Gamma et al. [22] for design patterns. The object analysis pattern template includes fields that describe 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 [31].

Given the potentially critical nature of embedded systems (e.g., X-by-wire, medical devices, etc.) in which faulty behavior of a system could lead to significant loss, methods for modeling and developing embedded systems and rigorously analyzing behavior before starting the design phase and committing to code are increasingly important. Additionally, errors introduced during
the analysis process are commonly the most expensive to fix, as they usually require subsequent changes to design and code [20]. However, currently much of the embedded systems industry uses ad hoc development approaches [14] that emphasize design and coding over analysis [19]. The large number of design patterns [22], especially those tailored to real-time systems [15, 18] and distributed real-time embedded systems (DREs) [41, 42, 43], is further evidence of this focus. Despite its importance, the analysis phase is often neglected in current development practice, often causing conceptual errors to be propagated to design and coding [20].

To address this problem, we propose object analysis patterns for use in the analysis phase of embedded systems development to guide the construction of a conceptual model of a system. Analysis patterns are not new per se (e.g., [10, 20, 21, 23]); for example, Fowler [21] identified several patterns that might be used during the analysis phase to represent conceptual models of business processes, such as abstractions from accounting, trading, and organizational relationships. While Fowler’s analysis patterns may be relevant to only one application domain, or may span several domains, our object analysis patterns focus explicitly on software development for the embedded systems domain. To distinguish the contents, format, and objectives of our work from Fowler’s, and to leverage Douglass’s ROPES [14] development process, we use the term object analysis patterns instead of analysis patterns. Our patterns not only guide developers in constructing UML-based conceptual models, but also provide property templates so that developers are able to validate their conceptual models, prior to design, by leveraging our previously developed UML formalization framework [40]. This conceptual model does not determine if the functionality captured by an object will be provided via hardware, software, or a combination of both, as this assessment is part of subsequent development stages.

The object analysis patterns are based in part on requirements patterns presented in [30, 31, 33, 34], but modifications have been made to reflect (1) additional feedback from industrial partners and reviewers of previous drafts of this paper, and (2) changes to ensure the patterns are applicable for object analysis. Organizations such as the Society of Automotive Engineers (SAE) are promoting the development of generic requirements for embedded systems in an attempt to reuse and leverage best practices, including investigating the use of UML [17]. In addition, the embedded systems community has expressed interest in how object-oriented modeling, specifically the UML, can be used for embedded systems development [13, 14]. Therefore, our object analysis 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-based conceptual models of embedded systems; these models are also amenable to automated analysis. The constraints are described in prose, specified in linear-time temporal logic (LTL) [38], and generally based on specification patterns (temporal logic templates for commonly occurring property specifications) developed by Dwyer et al. [16]. Previously, we developed UML modeling/analysis and formalization frameworks and tools that facilitate the rigorous development of embedded systems [8, 40]. Specifically, we have tools to support the graphical construction of conceptual models (Minerva [8]), the translation of these models into formal specifications (Hydra [9, 40]) that can then be analyzed using the appropriate tools, such as the Spin simulator and model checker [27], and the visualization of errors captured in terms of the original graphical models (using Minerva), thereby greatly accelerating the development and refinement process. Developers can thus create UML-based conceptual models of their systems and check these models for adherence to critical properties prior to the design phase.

To create the object analysis patterns, we analyzed the development artifacts of several embedded systems, largely obtained from the automotive industry. Subsequently, we applied the object analysis patterns to several embedded systems (e.g., [50, 52]) obtained from our industrial partners to determine the patterns’ utility. Thus far, feedback from industrial collaborators indicates that the object analysis patterns enhance communication between developers with different levels of experience and backgrounds. Additionally, the object analysis patterns enable the developers, guided by the structure and behavior UML diagrams in the templates, to quickly construct conceptual models of their systems. Nevertheless, the patterns will continue to evolve as we receive feedback from academia and industry about possible improvements. Object analysis patterns not only provide guidance to new developers of embedded systems for determining the key elements of many embedded systems, but they also provide examples of how to model these elements with a commonly accepted diagramming notation, UML. With our previously developed UML formalization capability, we are able to validate (using simulation) the behavior of the conceptual models as captured by the state diagrams [8] within the structural context imposed by the class diagrams. Furthermore, constraints from the object analysis patterns can guide developers in the construction of formal properties to check against their UML models. The result is that developers can accelerate
the initial development of conceptual models through the use of object analysis patterns, and then
using our formalization work and tools, they have a means to rigorously check that requirements
are captured appropriately in the models using simulation and model checking techniques.

The remainder of this paper is organized as follows. **Section 2** gives background information,
including an overview of UML, Broy et al.’s [6] logical architecture for embedded systems, and
Douglass’s ROPES [14] development process. It also briefly overviews our pattern-driven approach
in order to provide context for how to apply object analysis patterns within a development process.
**Section 3** overviews the object analysis patterns defined thus far, and presents our template for
describing them. **Section 4** describes the application of the patterns to an example from the
automotive industry, a self-cleaning diesel filter system [50]. **Section 5** overviews related work.
Finally, conclusions and future work are discussed in **Section 6**.

2. Background

Section 2.1 briefly overviews the Unified Modeling Language (UML) [4] and how we use it in
our approach. Additionally, Section 2.2 provides background information on an architectural view
and a development methodology commonly used for embedded systems, both of which we leverage
in our approach. Section 2.3 briefly overviews our pattern-driven approach to provide context for
how our object-analysis patterns can be applied within a development process.

2.1. UML

The Unified Modeling Language (UML) [4] has become the *de facto* standard for general-purpose
visual system modeling; that is, UML can be used to specify and document the artifacts of a system
throughout the development process. The UML comprises a collection of diagrams that depict the
structural and behavioral information of a system. There are several distinct types of diagrams;
diagram types to capture structural information include class, object, and package diagrams, and
diagram types to capture behavioral information include state machine, use case, and interaction
diagrams. Approaches exist that extend the UML to capture additional information. For example,
one of the extensions in the proposed System Modeling Language (SysML) [1] is a requirements
diagram that can be used to capture and structure system requirements.
The input to our UML formalization framework [40] is a UML profile; that is, the framework accepts a subset of UML tailored to a specific application domain (i.e., embedded systems). The UML subset that is used as input to our formalization framework consists of class and state diagrams.¹ Metamodels describing the syntax of UML diagrams used as input to our formalization framework can be found in [40]. For each UML element in the profile, we give a rule for defining the semantics in terms of the target language.

2.2. Development Methodologies for Embedded Systems

Broy et al.’s logical architecture. We base our overall view of embedded systems on Broy et al.’s logical (reference) architecture for embedded systems [6] (as shown in Figure 1). An embedded system is decomposed into five main components: a Controlling Device, controlled Physical Devices (e.g., Actuators and Sensors), a User Interface, the Environment, and the User as a special part of the Environment; each component is essentially a black box, and arrows represent abstract communication links (specific types of communication are introduced during design). Generally, the User interacts with the system via the User Interface and its controls and indicators (e.g., a homeowner setting the thermostat on a central air conditioning system). Nevertheless, the User is also part of the Environment and might also interact with the system in terms of Sensors and Actuators (e.g., a driver of a car stepping on the brake of an anti-lock brake system). The User Interface is connected to the Controlling Device, which, in turn, interacts with the Environment via the Physical Devices, namely Actuators and Sensors. While conceptual models (i.e., high-level structural and behavioral models) created according to this architectural view are abstract and implementation-independent, they are still amenable to automated consistency checking, simulation, and validation of critical properties [6], as we demonstrate in Section 4.

Douglass’s ROPES development process. Development processes for embedded systems, such as ROPES [14] and COMET [24], commonly emphasize rapid turnaround, early proofs of correctness, and low risk. These development processes advocate an analysis phase with an object-based conceptual model as the main artifact. Figure 2 presents an abstracted diagram of a portion

¹Our formalization framework was originally developed based on UML version 1.4 notation. Changes between UML versions 1.4 and 2.0 to the UML subset that we formalized have been incorporated into the framework.
of the ROPES development process, focusing on the analysis and design phases. The ROPES analysis phase serves two main purposes: to capture and refine use cases, and to perform an initial allocation of a set of objects that provide the behavior of the system [28]. Analysis patterns (e.g., [10, 20, 21, 23]), depicted as a data store connected to the “Analysis” process oval in the data flow diagram in Figure 2, are commonly applicable in this three-step analysis phase [14]:

(1) **Requirements Analysis**: During requirements analysis, extract requirements and convert them into a form understandable by the developer as well as the customer. Capture and structure use cases, and associate them with actors. Additionally, use scenarios to capture
behavioral information (using sequence charts) about the interaction between actors and system.

(2) Systems Analysis: During systems analysis, build more rigorously defined models and partition the system into mechanical, electronic, chemical, and software components. It is common practice for systems to be tested at this step using executable models, such as activity diagrams and mathematical models; however, the testing takes place on a functional level rather than on an object level.

(3) Object Analysis: After the previous steps have identified a black-box view of the system capturing the requirements, object analysis creates a conceptual model of the internals of a system in terms of objects, classes, and their relationships. Object analysis comprises two subphases, typically performed concurrently:

- **Structural Object Analysis:** During this phase, capture structural properties of the conceptual model, such as key objects, classes, and relationships in the system.

- **Behavioral Object Analysis:** During this phase, capture high-level behavioral information about the system by refining the behavior of objects/classes defined during structural object analysis.

The resulting object-based conceptual model produced by the analysis phase can be seen as a high-level structural and behavioral model rather than a high-level representation of the implementation of the final system [28]. Specifically, the conceptual model captures characteristics of all possible correct solutions. Thus, it defines the acceptance criteria for any specific solution [14].

Our object analysis patterns for embedded systems complement the structural and behavioral object analysis subphases by providing UML-based structural and behavioral information that can be used to guide developers in constructing an object-based conceptual model amenable to automated analysis. Consequently, a data store for object analysis patterns would be connected to the “Object Analysis” oval in Figure 2. Design, in contrast to analysis, refines or adds elements to the conceptual model produced during analysis in order to define a specific solution [14]. Thus, design refines the conceptual model with respect to the optimization of certain criteria, such as

---

2Douglass terms this model the *Analysis Object Model*, as shown in Figure 2, while we use the more general term *conceptual model* in this paper.
response time. Design patterns (such as those developed for DREs [41, 42, 43] or the patterns by Gamma et al. [22]), depicted as a data store connected to the “Design” process oval in Figure 2, are closer to implementation and are generally more concerned with tasks, active objects, scheduling, packaging generated object artifacts, the distribution of run-time components, addressing non-functional properties, and detailing the definition of interfaces [15].

2.3. Pattern-Driven Modeling and Analysis Approach

To add further benefit to our object analysis patterns, this section describes a pattern-driven approach to modeling and analyzing requirements for embedded systems. Previously, we developed a general framework [40] for formalizing a subset of UML diagrams in terms of different target languages [39, 40]. The mapping process from UML to a target language has been automated in a tool called Hydra [8, 40]. To complement our formalization framework and Hydra’s automatic generation of formal models, we developed a modeling and visualization framework [8] to support a number of tasks necessary to model and analyze UML diagrams. These tasks have been automated in a tool called MINERVA [8] and include the following capabilities: graphical construction of syntactically correct diagrams, and visualization within UML diagrams of consistency checking results, simulation traces, and paths of execution that lead to errors. Object analysis patterns are the focus of this paper; therefore, we only briefly describe the modeling and analysis process to provide context for how the patterns can be applied. Figure 3 overviews our approach, illustrating how object analysis patterns can be combined with the iterative modeling and analysis process supported by MINERVA [8] and Hydra [8, 40] (here instantiated with Promela formalization rules and the model checker Spin [27]). The general UML-to-Promela formalization approach is to map objects to processes in Spin (proctypes) that exchange messages via channels. Nested and concurrent states are also formalized as processes. More details on the modeling and analysis process, the underlying formalization framework, and MINERVA’s visualization capabilities can be found in [8, 33, 40].

As shown in the activity diagram in Figure 3, the user begins by selecting appropriate object analysis patterns based on the results of the requirements analysis of the system. Then, guided by the structural and behavioral diagram templates in the selected object analysis patterns, the user constructs a conceptual model of the system comprised of UML class and state diagrams using
Minerva’s graphical editors. When the user has finished constructing the diagrams representing the conceptual model of the system, Minerva exports the diagrams in a text-based representation for Hydra, which then performs consistency checks. If any errors are detected, Minerva visualizes structural consistency-checking results. (To keep the focus on the object analysis patterns, we omit discussion of these capabilities in this paper; see [8] for details.) Once consistency checking is successful, Hydra generates the formal specification corresponding to the conceptual model that can be used to validate the UML diagrams through simulation using Spin. Minerva visualizes simulation traces within the UML diagrams, enabling the user to address unexpected behavior. In addition, the user may instantiate (as LTL properties) temporal logic templates for system properties from the “Constraints” field of those object analysis patterns used to guide the modeling of the system. These LTL properties, defined in terms of attributes, states, and signals of the UML model, can then be checked against the UML diagrams through model checking using Spin. Minerva visualizes counterexample traces for properties that do not hold 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.

3. Object Analysis Patterns

This section briefly overviews our current object analysis pattern repository, introduces classification criteria for object analysis patterns, and describes the object analysis pattern template in detail. For illustrative purposes we include fields for a specific pattern, the User Interface Object Analysis Pattern, when presenting the template. The complete object analysis patterns repository can be found in [35].

3.1. Pattern Catalogue Overview

The physical portion of embedded systems enables us to recognize commonly-occurring objects with well-defined behavior (e.g., actuators and sensors). Object analysis patterns describe well-defined relationships between such objects (e.g., the relationships between a user interface and the controls and indicators of an embedded system). We leverage the community’s interest in UML
Figure 3. Overview of our pattern-driven approach to modeling and analysis
when representing these patterns. Further, we have identified commonly occurring constraints within the context of the patterns and we leverage specification patterns from Dwyer et al. [16] to express these constraints. We use patterns as a bridge to formal specification and property analysis using our previously developed UML formalization framework. Table 1 lists and gives a brief description of the current object analysis patterns in the repository that we have identified by analyzing several embedded systems (e.g., a diesel filter system [50], an electronically controlled steering system [50], an anti-lock brake system [52], and an adaptive cruise control system [52]).

For ease of use, it is important to have a means to classify patterns. As described in Section 2, object analysis can be divided into two subphases (as shown in Figure 2), structural object analysis and behavioral object analysis [14]. The key tasks in each of these subphases are:

- **Structural Object Analysis:**
  Capture structural properties of the model by
  1. identifying objects,
  2. abstracting identified objects into classes, and
  3. capturing relationships (such as associations, composition, and inheritance) between classes;

- **Behavioral Object Analysis:**
  Specify the behavior of previously defined objects/classes by
  1. defining the essential behavior of reactive objects, and
  2. constructing mechanisms of object correlation.

Structural and behavioral object analyses are commonly performed concurrently. Several iterations may be necessary before the final conceptual model is obtained. Given that structural and behavioral object analyses are mutually dependent, our patterns are neither purely structural or behavioral in nature. Nevertheless, the patterns tend to have an inclination to focus primarily on either the structural or behavioral phase of object analysis and can therefore be classified accordingly as *structural object analysis patterns* or *behavioral object analysis patterns*. The result of the classification according to structural and behavioral object analyses and their respective subphases can be found in Table 2.

For example, the *User Interface* Object Analysis Pattern (see Section 3.3) does not aid in identifying the actual controls and indicators required for a specific system, but does describe how to
<table>
<thead>
<tr>
<th>Name of Pattern</th>
<th>Description of Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuator-Sensor:</td>
<td>The Actuator-Sensor pattern specifies basic types of sensors and actuators in an embedded system. In addition, interaction with the environment via sensors and actuators is one of the main responsibilities of an embedded system. Therefore, the pattern also describes how relationships between actuators and sensors and other components in the system can be captured.</td>
</tr>
<tr>
<td>Communication Link:</td>
<td>Due to the growth in demand for distributed real-time embedded systems, communication capabilities are becoming more important. The Communication Link pattern describes how to capture high-level information about communication capabilities offered by an embedded system, such as sending periodic heart beat messages to other systems.</td>
</tr>
<tr>
<td>Computing Component:</td>
<td>Embedded systems frequently must offer various operational modes as they often function in an environment where a shutdown of the system would lead to a significant loss. In the Computing Component Pattern, various operational modes of an embedded system are specified, such as fail-safe modes that a system enters in response to occurring faults.</td>
</tr>
<tr>
<td>Controller Decompose:</td>
<td>The Controller Decompose pattern describes how to decompose an embedded system into different components according to their responsibilities. This pattern is the foundation upon which all other patterns are based. It introduces a high-level view of an embedded system and refers to other object analysis patterns for refinement.</td>
</tr>
<tr>
<td>Detector-Corrector:</td>
<td>Embedded systems typically have tight timing and operational constraints. Providing a mechanism to assure that a component is operational or that specific constraints on the system are not violated is the objective of the Detector-Corrector Pattern. In this pattern, detectors offer fault detection capabilities, correctors offer fault correction capabilities, and the interaction between both types of components is controlled by a local fault handler.</td>
</tr>
<tr>
<td>Fault Handler:</td>
<td>Fault handing is crucial for embedded systems. In this pattern, a global fault handler and several local fault handlers for an embedded system are specified. The global fault handler introduced in the Fault Handler Pattern collects fault messages from the local fault handlers and is a central coordinator for system recovery and safety.</td>
</tr>
<tr>
<td>User Interface:</td>
<td>User interaction is an important aspect of an embedded system. The system interacts with the user via so-called controls and indicators. Differing from sensors and actuators, interaction via controls and indicators is usually not as tightly constrained, e.g., timing constraints are less strict. The User Interface pattern describes how to specify an object model for a user interface that is extensible and reusable.</td>
</tr>
</tbody>
</table>

Table 1. Current list of object analysis patterns for embedded systems
abstract such objects into classes by denoting super classes that can be used to specify controls and indicators, and it also captures the relationship to the ComputingComponent and the GlobalFault-Handler in the system. Additionally, the User Interface Object Analysis Pattern briefly describes how other components interact with the user interface to carry out their responsibilities. As this pattern clearly focuses on structure, it is classified as a structural object analysis pattern.

On the other hand, the Detector-Corrector Object Analysis Pattern focuses on behavior, although it also denotes basic structural information. It describes the essential behavior of detectors, correctors, and local fault handlers in terms of UML state diagrams. It also captures how the local fault handler coordinates the interaction between detectors and correctors in the fields “Participants” and “Responsibilities” (not shown in this paper; see [35] for details) as well as in terms of a UML sequence diagram in the “Behavior” field. Accordingly, the Detector-Corrector Object Analysis Pattern is classified as a behavioral object analysis pattern.

<table>
<thead>
<tr>
<th>Object Analysis Pattern</th>
<th>Classification</th>
<th>Structural Object Analysis</th>
<th>Behavioral Object Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuator-Sensor Pattern</td>
<td>Structural</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Communication Link Pattern</td>
<td>Behavioral</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Computing Component Pattern</td>
<td>Behavioral</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Controller Decompose Pattern</td>
<td>Structural</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Detector-Corrector Pattern</td>
<td>Behavioral</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Fault Handler Pattern</td>
<td>Behavioral</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>User Interface Pattern</td>
<td>Structural</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2. Pattern classification according to subphases of structural and behavioral object analysis

3.2. Pattern Template

In contrast to other informal presentation styles for patterns (e.g., [21]), we use a template similar in style to that used by Gamma et al. [22] in order to facilitate each object analysis pattern’s
understanding and application. UML diagrams are used to give structural and behavioral information. As suggested in [45], we use natural language to supplement both the diagrams and formal specifications in order to describe important aspects of the patterns as a means for facilitating the understanding of the requirements from different viewpoints.

We modified the original design pattern template in several aspects to address the needs of creating and analyzing conceptual models of embedded systems. Specifically, the design pattern template has been extended with “Constraints”, “Behavior”, and “Applicable Design Patterns” fields. The fields “Implementation” and “Sample Code” have been removed because they were too specific to software design and implementation, and the field “Related Design Patterns” has been renamed to “Related Object Analysis Patterns”.

The “Constraints” field contains properties that are of one of three possible types: invariant properties that have to hold in every state, cause-effect properties that describe the relation between an event and an expected response to this event, and periodic properties that specify recurring events. Invariant and cause-effect properties are generally based on specification patterns by Dwyer et al. [16]. Thus far, our constraints have included, but are not limited to, LTL representations of two of the most commonly used general specification pattern categories, universality/absence (to capture invariant properties) and response (to capture cause/effect relationships in system behavior). The symbols “□” and “♦” denote “it is always the case” and “eventually”, respectively. Additionally, “∧” denotes the boolean AND operator, “∨” the boolean OR operator, “¬” a boolean negation, and “→” a boolean implication. Finally, “∪” denotes the Strong-Until operator and “W” the Weak-Until operator that is related to the ∪ operator in the following way: 
\[ p \ W q = (□ p) \lor (p \cup q) \] [3]. (Meaning that \( p \) is either globally true or \( p \) is true until \( q \) becomes true.) Additionally, the keyword “in” denotes that an object is in a specified state, while the keyword “send” specifies that a message is sent to an object. Constraints provide temporal logic templates for instantiating properties specific to a modeled system in terms of the UML diagrams describing the system. An example property is the so-called response property pattern [16]. Specified in LTL and with a global scope, the response property is denoted by □ (p → ♦ q). (It is always the case that an occurrence of \( p \) is eventually followed by an occurrence of \( q \).) For readability and understandability purposes, the temporal logic templates contain prose representations of expressions that can be refined to be more precise for analysis purposes. For example, a general
property in the “Constraints” field of a object analysis pattern could be of the form

\[ \square\Leftarrow\text{"Fault reported to global fault handler"} \rightarrow \Diamond\text{"Defined recovery action is started"} \]

to indicate that it is always the case that when a fault is reported to the global fault handler, eventually the global fault handler starts the recovery action appropriate for the reported fault. When this property is instantiated for a modeled system, the prose expressions (for example, “Fault reported to global fault handler’) would be replaced by the model’s representation of the expression (for example, \( \text{send(GlobalFaultHandler.ReportGlobalFault(FaultID))} \)).

Finally, the “Behavior” field contains sequence and state diagrams that illustrate sample behavior, and the “Applicable Design Patterns” field contains the names of related design patterns that can be used to further refine the object analysis pattern in subsequent development phases.

### 3.3. Pattern Example

This section describes the contents of each field of our object analysis pattern template (encapsulated in bold boxes) and then provides the contents of the corresponding field of the User Interface Pattern, prefixed by “Example:”, for illustration purposes.

**Pattern Name and Classification:**

The pattern name consists of a description of the pattern; the classification denotes if the pattern is of structural or behavioral nature.

**Example:**

*User Interface:* Structural Pattern

**Intent:**

A brief description of the problem(s) that the pattern addresses.

**Example:**

Specify a user interface that is responsible for interaction with the user of the system via controls and indicators.

**Motivation:**

A description of sample goals and objectives of a system that motivate the use of the pattern. Use cases and use case diagrams describe goals of the pattern’s application.

**Example:**

A user interface is an important part of an embedded system because it is responsible for most interactions with the user of the system. The user interface receives inputs and displays information
about the status of a system. Typically, user input to an embedded system does not have tight timing constraints, and the sensors capturing this input are termed *controls*, such as a homeowner setting the thermostat for a central air conditioning system. Similarly, actuators of the user interface that are used to convey the current state of the system are named *indicators*.

One important task of the user interface is fault signaling. Various indicators can be used to show the user the current state of the system. The faults should be classified in the global fault handler so that minor and severe faults can be distinguished [53]. The reactions defined in the user interface should be classified into various warning levels according to the severity of a fault. The global fault handler activates, depending on the fault class, the appropriate warning level. The indicators and controls inherit their interface from the *Actuator-Sensor* Pattern. The global fault handler is not necessary for the functioning of the user interface, but most embedded systems have some type of fault handler. Therefore, the interaction between a global fault handler and the user interface is described in detail in this pattern.

The second important task is receiving user inputs. Usually user inputs through the user interface are rare or not possible. For example, a driver cannot set the operational status of an anti-lock brake system. Therefore, having the computing component continuously query the controls would be inefficient. In order to reduce the burden on the computing component, a mechanism similar to the *Observer Design Pattern* [22] can be used. The user interface is responsible for querying the controls; only if a change is detected, does the user interface send an update message to the computing component, thus prompting the computing component to update its values for the controls.

Finally, the user interface is also responsible for displaying the current operational mode of the system to the user. Therefore, the user knows whether the system is initializing or in a fail-safe state due to faults that have occurred.

Using this pattern for the user interface has the advantage that a user interface of an embedded system can be easily reused and extended. The same user interface can be used for another system as long as faults and operational modes of a system may be classified in a similar manner.

Figure 4 shows the use case diagram for the *User Interface* Pattern. The diagram describes the user interactions with a user interface of a system.

![Figure 4. UML use case diagram of the User Interface Pattern](image-url)
Use Case: Interact with user
Actors: User
Description: The system receives control values and indicates current system conditions to the user.
Includes: Read input from controls, Activate indicators
Extends: -

Use Case: Read input from controls
Actors: None
Description: The system reads input from the controls.
Includes: -
Extends: -

Use Case: Activate indicators
Actors: None
Description: The system activates indicators.
Includes: -
Extends: -

Applicability:
Describes the conditions in which the pattern may be applied.

Example:
The User Interface Pattern is applicable

- in any embedded system that needs to interact with a user.

Do not use the User Interface Pattern when

- controls and indicators have tight timing constraints; in this case, use the Actuator-Sensor Pattern to connect controls and indicators, as actuators and sensors respectively, directly to the computing component.

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

Example:
The class diagram of the User Interface Pattern can be seen in Figure 5. The controls and indicators inherit from the AbstractActuator and AbstractPassiveSensor classes from the Actuator-Sensor Pattern. Therefore, they have the same interface as a sensor or an actuator, respectively.

Behavior:
Provides an illustrative representation of scenarios for class and object interaction. Also gives a description of the behavior of the pattern by using sample or high-level, abstract UML state and sequence diagrams.

Example:
Figure 6 shows an example sequence diagram for the behavior of the UserInterface in case of a minor and a severe fault. Above the bold horizontal line, the GlobalFaultHandler activates the warning level for a minor fault. The UserInterface activates Indicator1 for a short time. Below the bold horizontal line, the GlobalFaultHandler activates the warning level for a severe fault. In this case, the UserInterface activates both indicators continuously.
Participants:
Itemizes the classes/objects that are included in the object analysis pattern and their responsibilities.

Example:

- **ComputingComponent**: Represents the central computing component of the system.
- **UserInterface**: Class representing the user interface of a system.
- **GlobalFaultHandler**: Responsible for centralized fault handling in the system.
- **AbstractIndicator** {abstract}: Inherits from abstract class AbstractActuator and provides an interface for all indicators.
- **AbstractControl** {abstract}: Inherits from abstract class AbstractPassiveSensor and provides an interface for all controls.
- **AbstractActuator** {abstract}: Defines an interface for the AbstractIndicator class, taken from the Actuator-Sensor Pattern.
- **AbstractPassiveSensor** {abstract}: Defines an interface for the AbstractControl class, taken from the Actuator-Sensor Pattern.
Figure 6. UML sequence diagram example of the User Interface Pattern

- AbstractBooleanControl {abstract}: Defines boolean controls.
- AbstractIntegerControl {abstract}: Defines integer controls.
- AbstractRealControl {abstract}: Defines real controls.
- AbstractComplexControl {abstract}: Complex controls have the same basic functionality as the Control class, but additional, more elaborate, methods and attributes need to be specified.
- AbstractBooleanIndicator {abstract}: Defines boolean indicators.
- AbstractIntegerIndicator {abstract}: Defines integer indicators.
- AbstractRealIndicator {abstract}: Defines real indicators.
- AbstractComplexIndicator {abstract}: Complex indicators have the same basic functionality as the Indicator class, but additional, more elaborate, methods and attributes need to be specified.
- ConcreteBooleanControl1, ConcreteIntegerControl1, ConcreteRealControl1, ConcreteComplexControl1, ConcreteBooleanIndicator1, ConcreteIntegerIndicator1, ConcreteRealIndicator1, ConcreteComplexIndicator1: Examples of concrete controls and indicators that can be instantiated.

Collaborations:
Describes how objects and classes interact to carry out the responsibilities given in the “Participants” field.

Example:
• The GlobalFaultHandler sends a message to the UserInterface to activate a specific warning level. The UserInterface then activates all indicators appropriate for the desired warning level.

• The UserInterface queries the controls. If a change is detected, then the ComputingComponent is notified to update its values corresponding to the UserInterface. Additionally, the ComputingComponent notifies the UserInterface about its current operational modes. The UserInterface then displays this information to the user.

• The ComputingComponent reports changes in its operational status to the UserInterface. Therefore, the user of the system is informed about the current operational status of the system.

• Indicators can either be activated by setting specific warning levels or by sending an activation message to the UserInterface for a specific indicator.

Consequences:
Describes how objectives are supported by a given pattern and gives the trade-offs and outcomes of the pattern’s application.

Example:

1. A global fault handler should be present to drive the user interface.

2. A method to notify the user interface of value changes must be present in the computing component.

3. All possible faults should be classified according to different levels of severity, where these levels may or may not be directly mirrored in the user interface.

4. The computing component should report the current operational mode to the user interface. The user interface then displays this information to the user.

Constraints:
Constraints describe properties that the model must satisfy after the pattern is applied. The “Constraints” field contains the property type of a template (followed in parentheses by the name of the corresponding specification pattern [16], if one exists), a prose description of the property, and the LTL representation of the template. By default, the scope of the LTL templates is global, but scoping templates based on specification patterns is straightforward as scoped versions of specification patterns are provided in [3].

Example:
The following constraints must be satisfied after the User Interface pattern has been applied:

• Cause-Effect Property (Response Specification Pattern):
  
  It is always the case that when a message is sent to the user interface to activate a specific warning level, then eventually all the indicators corresponding to this level will be activated.

  \(\Box\(''\text{Warning level sent to user interface}'' \rightarrow \Diamond\(''\text{Appropriate indicators for warning level activated}''\))\)
• **Cause-Effect Property (Response Specification Pattern):**
  It is always the case that when a control value is changed, the variable that the computing component monitors to detect changes will eventually be updated.

\[
\square (\text{Control value change} \rightarrow \\
\diamond (\text{ComputingComponent.‘Control value change notification’}))
\]

• **Cause-Effect Property (Response Specification Pattern):**
  It is always the case that when the computing component needs to update the control values, it will eventually do so or power off.

\[
\square (\text{ComputingComponent.‘Control value change notification’} \rightarrow \\
\diamond (\text{ComputingComponent.‘Control values updated’} \lor \text{‘System power off’}))
\]

• **Cause-Effect Property:**
  It is always the case that if the computing component is notified of a change in the user interface control values, then this notification is only reset when an update through the computing component takes place, or the system stops before the value can be updated, such as a system shutdown. In other words, the computing component does not miss an update because the value was reset before it could perform the update.

\[
\square (\text{ComputingComponent.‘Control value change notification’} \rightarrow \\
(\text{ComputingComponent.‘Control value change notification’} \lor \\
\text{‘Update through ComputingComponent’}))
\]

**Applicable Design Patterns:**
Applicable design patterns that can be used to refine the object analysis pattern in subsequent development stages.

**Example:**

- **Observer Design Pattern [22]:**
  This pattern describes a technique where observers can register with a subject to be notified about changes in the state of the subject. This mechanism can be used to update the computing component when the state of the user interface changes.

- **Singleton Design Pattern [22]:**
  This design pattern can be used to assure that only one instance of the user interface exists in the system.

**Also Known As:**
Lists alternative names for the object analysis pattern.

**Example:**
To be determined.
Known Uses:
Examples of the pattern found in real systems.

Example:
To be determined.

Related Object Analysis Patterns:
Lists related object analysis patterns and advantages/shortcomings that would result from pattern combination.

Example:
- **Computing Component Object Analysis Pattern:**
  The computing component uses the user interface to signal operational modes to the user.

- **Fault Handler Object Analysis Pattern:**
  The global fault handler uses the user interface to signal the occurrence of faults to the user.

- **Actuator-Sensor Object Analysis Pattern:**
  Indicators and controls in this pattern inherit the interface from the actuators and sensors of the Actuator-Sensor Pattern.

4. Construction and Model Checking of Object Analysis Pattern-Based Models

This section illustrates how we applied our object analysis patterns to the modeling and analysis of an automotive application obtained from one of our industrial partners, Detroit Diesel. Specifically, we illustrate how several object analysis patterns interplay to guide the creation of a UML-based conceptual model of a diesel filter system (DFS), an embedded system controlling a self-cleaning particulate filter that reduces the amount of pollutants emitted from the exhaust of diesel trucks [50]. We also illustrate how constraints from the object analysis patterns guide the instantiation of temporal-logic-based formal properties to check against the model, and how Minerva [8] and Hydra [8, 40] enable simulation and model checking with Spin [27] of the user-constructed UML models.

4.1. Diesel Filter System 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. Figure 7
shows the physical architecture of an exhaust filter system. 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 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 current pressure in the filter canister is read from a pressure sensor that is contained within the filter canister. The grid heating sequence will not begin if too few engine revolutions have occurred since the last time the cleaning cycle was completed, or if the current engine revolutions per minute (RPMs) are too low. The RPM information is acquired from the engine control module.

![Figure 7. Physical structure of a diesel filter system](image)

### 4.2. Object Analysis Patterns for the diesel filter system

We present six object analysis patterns that we identified to be appropriate for this system based on the DFS requirements [50]: the Actuator-Sensor, Computing Component, Controller

---

3For brevity, only one filter tube is shown in Figure 7.
Decompose, Fault Handler, User Interface, and Detector-Corrector patterns. (The full description of the User Interface Pattern can be found in Section 3.3, while the complete object analysis patterns repository can be found in [35].) Figure 8 illustrates how the UML class diagram template and other information in the “Structure” field of each of these patterns can be used to guide the creation of a preliminary UML class diagram for the DFS. We briefly overview the patterns and how each is depicted in Figure 8 in the following paragraphs.

![UML class diagram of the diesel filter system](image)

**Figure 8. Object-analysis-pattern-guided UML class diagram of the diesel filter system**

**Structural patterns.**

We applied the following structural object analysis patterns to the DFS:
Controller Decompose Pattern: The Controller Decompose Pattern, our most general structural object analysis pattern, leverages Broy’s reference architecture [6] and should be applied first. We applied this pattern to the DFS in order to decompose the system into various components according to their responsibilities. In the DFS, sensors and actuators are controlled by a central ComputingComponent, while a user interface is responsible for controlling an indicator that indicates system status to the user. The DFS does not possess a communication link. Components captured by the Controller Decompose Pattern are refined through the application of additional object analysis patterns.

Actuator-Sensor Pattern: To further refine the sensors and actuators of the DFS as captured by the Controller Decompose Pattern, we applied the Actuator-Sensor Pattern. This pattern, denoted by dashed boxes and lines in Figure 8, shows how abstract sensor and actuator classes are used to give a common interface to the concrete sensors (e.g., CurrentMirrors that sense the amount of current flowing through their respective HeaterRegulators, a PressureSensor that monitors the system pressure) and actuators (e.g., a DriverDisplay that indicates system status to the user, HeaterRegulators that burn off particulates) in the DFS.

User Interface Pattern: We applied the User Interface Pattern to further refine the user interface of the DFS as captured by the Controller Decompose Pattern. It is represented by the shaded boxes and lines in Figure 8. The DFS UserInterface controls only one indicator, the DriverDisplay, which represents a simple warning device, such as an indicator light.

Behavioral patterns.
We applied the following behavioral object analysis patterns to the DFS. Behavioral object analysis pattern templates contain UML state diagram templates as well as UML class diagram templates; the class diagram templates guide creation and refinement of the DFS class diagram as illustrated in Figure 8, while the state diagram templates guide creation of state diagrams for components of the DFS as described in Section 4.4.

Computing Component Pattern: The system requirements indicate that the DFS requires several operational modes. Therefore, we applied the Computing Component Pattern to the ComputingComponent, which is shown in bold in Figure 8. The identified operational modes according to the pattern are: PowerOff, Normal Behavior, and Initialize, which appear in the state diagram in Figure 11.

Fault Handler Pattern: To add fault handling capabilities to the DFS, we applied the Fault Handler Pattern. The GlobalFaultHandler and the LocalPressureFaultHandler are illustrated with dash-dotted boxes and lines in Figure 8. The GlobalFaultHandler controls the ComputingComponent to initiate safety actions when faults occur. It also controls the UserInterface, warning the user that faults have occurred.

Detector-Corrector Pattern: The DFS must monitor the pressure in the system and initiate corrective actions if needed. Therefore, we applied the Detector-Corrector Pattern to the system. The PressureDetector, denoted by a striped box and long-short-short dashed lines in Figure 8, monitors the PressureSensor. If it detects a violation of the maximum pressure value,
then it notifies the LocalPressureFaultHandler of the fault as described in the state diagram in Figure 12. The system does not contain any correctors, as the response to excessive pressure in the filter tube will be to turn off the DFS (this safety action will be performed by the GlobalFaultHandler), and a mechanical backup valve will let the exhaust gas bypass the filter.

4.3. Abstraction and Equivalence Classes

*Abstraction* can significantly simplify the modeling and analysis of a system. First, we create specialized models for different aspects of the embedded system. In this study, we are interested in specifying and analyzing the DFS cleaning cycle. Thus, we model only those portions of the system that are relevant to our focused 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 analysis model.

Second, we construct *equivalence classes* for the possible values of system conditions. These equivalence classes are determined according to their impact on the behavior of the system. Generally, the operational status of a component is represented as *non-working* (false) or *working* (true), as shown in Expression (1) in Figure 9. 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 document, as shown in Expressions (2), (3), and (4), respectively, in Figure 9 (∞ represents the target language-dependent upper bound). We also introduce physical abstraction values for modeling purposes (Figure 9, 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). Appropriate values must be determined by a domain expert. The equivalence classes are used to model check the system under different scenarios. A scenario is collection of possible initial system conditions randomly determined from the equivalence classes. The equivalence classes in which these system values are found in denote the expected behavior of the system. For example, in a scenario in which the system pressure is above 10,000 Pa the DFS is expected to perform a system shutdown, while a system in a scenario with a pressure value of 10,000 Pa or below is expected to initiate a cleaning cycle (assuming all other system values in the scenario are in equivalence classes that permit the initiation of a cleaning cycle).
Thus far, abstractions are introduced by the developer, but we are investigating several approaches to automated abstraction, such as integrating the variable restriction and variable abstraction approaches presented in [26].

(\text{Component})\text{OperationState} = \begin{cases} 0 & \text{non-working} \\ 1 & \text{working} \end{cases} \quad (1)

Each component in the system can report its operational status as working or non-working. We are particularly interested in the status of the sensor \text{PressureSensor} and the actuators \text{HeaterRegulator1} and \text{HeaterRegulator2}.

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

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

\text{TotalRPMValue} = \begin{cases} [0; 10,000] \\ (10,000; \infty) \end{cases} \quad (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.

\text{CurrentRPMValue} = \begin{cases} [0; 700] \\ (700; \infty) \end{cases} \quad (4)

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

\text{PressureSensorCleanupValue} = \begin{cases} -250 \\ 300 \\ 3,000 \end{cases} \quad (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.

\text{HeaterCurrentConversionRatio} = \begin{cases} 2 \\ 3 \\ 4 \end{cases} \quad (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.

\textbf{Figure 9. Equivalence classes for system conditions}
4.4. UML-based Conceptual Modeling of the diesel filter system

Based on the UML class diagram in Figure 8 (obtained by applying the UML class diagram templates from the object analysis patterns relevant to the DFS as described in Section 4.2), state diagram templates from behavioral object analysis patterns, and the abstractions described in the previous section, we created UML object and state diagrams to obtain the conceptual model of the DFS. Figure 10 overviews the UML object diagram for the DFS (attributes and methods have been elided for brevity; components attributed to the different patterns retain their shading/line characteristics from Figure 8). The ComputingComponent, the core of the system, reads values from the sensors PressureSensor, CurrentMirror1, 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 GlobalFaultHandler processes error messages received and takes appropriate actions (defined in the GlobalFaultHandler state diagram). The PressureDetector monitors the PressureSensor, notifying the LocalPressureFaultHandler if the pressure exceeds 10,000 Pa. The LocalPressureFaultHandler notifies the GlobalFaultHandler of occurring faults, which, in turn, shuts down the ComputingComponent and activates the UserInterface to signal to the user that faults have occurred and that the system has shut down. The UserInterface controls the DriverDisplay, which represents a simple warning light.

Additionally, in order to enable the analysis of a model, our approach introduces two special classes. First, an instance of an Environment class specifies values for attributes chosen according to the equivalence classes for system and environment conditions as depicted in Figure 9. Second, an instance of a _SYSTEMCLASS_ class instantiates components of the system and non-deterministically selects values for system and environment conditions as defined in the instance of the Environment class. Furthermore, attributes declared in the _SYSTEMCLASS_ class are global variables and can be used by objects to exchange information using shared memory.

In our approach, each component has its own state diagram; to demonstrate how the UML state diagram templates contained in the “Behavior” field of the behavioral object analysis patterns
can guide the construction of concrete state diagrams for a specific system, we show in this paper
the (elided) state diagram of the ComputingComponent, the central component of the DFS, in
Figure 11, and the state diagram of the PressureDetector for the DFS in Figure 12. The structure
of both state diagrams follows that of the state diagram templates given in the “Behavior” field of
the Computing Component Pattern and the Detector-Corrector Pattern, respectively. Specifically,
the state diagram of the ComputingComponent has the state PowerOff and the composite states
Initialize and NormalBehavior (elided in Figure 11). The state diagram of the class PressureDetector
(Figure 12) contains all the states of the diagram template denoted in the Detector-Corrector
Pattern, while the transition labels and “do” actions of the states have been instantiated with
respect to the DFS to detect a pressure violation (with the exception of the DetStart() and DetStop()
signals).
Figure 11. UML state diagram of the ComputingComponent (elided)

Figure 12. UML state diagram of the PressureDetector
As illustrated in Figure 11, the DFS performs three main steps. First, upon system activation, the DFS enters a composite state Initialize. If the initialization is performed successfully, then the system enters an idle phase. The three states GetPressure1, GetPressure2, and Idle represent the idle phase 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. 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 state, waiting for either the pressure to exceed 8,000 Pa again or a system shutdown message to arrive.

4.5. Analysis Using Constraints from Object Analysis Patterns

After we used Minerva to construct UML diagrams of the system, we used Hydra to generate an executable specification of the system in terms of Promela. Briefly, objects are captured as proctypes that communicate via channels using queueing semantics [9, 40]. In this section, we examine one requirement for the DFS, shown in a text box. We give the prose requirement, the relevant object analysis pattern(s) in bold italics, the relevant constraint(s) from the “Constraints” field of each object analysis pattern, the instantiated constraints checked against the generated Promela specification, the analysis results (including visualizations), and the corrective actions taken.

Requirement to be analyzed.

If the pressure in the canister exceeds 10,000 Pa, then a warning light will be turned on.

As shown in Figure 12, excessive pressure is detected by the PressureDetector and reported to the LocalPressureFaultHandler. We use the numeric fault identifier code “200” in the model to represent
this fault. As our formalization framework currently does not contain a notion of time (we are investigating the extension of our formalization framework to include timing information [32]), the PressureSensor notifies the PressureDetector in case of a pressure change instead of the PressureDetector periodically querying the pressure sensor after a time out period. The LocalPressureFaultHandler forwards the fault to the GlobalFaultHandler, which, upon receiving the fault message, interacts with the UserInterface. Finally, upon notification from the GlobalFaultHandler, the UserInterface takes appropriate action, in this case turning on a warning light. Five constraints from the “Constraints” field of the Detector-Corrector, Fault Handler, and User Interface object analysis patterns combine to specify this requirement. The five constraints are described below:

1. **Detector-Corrector Pattern Constraint:**

   If there is a violation, then the corresponding detector should eventually detect the violation.

   \[ \square(\text{"Physical violation"} \rightarrow \Diamond(\text{"Detector detects violation"})) \]  

   **Instantiated Constraint:** In the DFS conceptual model, pressure exceeding 10,000 Pa is a violation of a specified safety constraint, and therefore the pressure value sensed by the PressureSensor is monitored by the PressureDetector. In the conceptual model of the DFS, the current pressure value sensed by the PressureSensor is represented by the attribute PressureValue. Possible values of the PressureDetector attribute Violation are zero (no pressure violation has been detected) and one (a pressure violation has been detected).

   \[ \square((\text{PressureSensor.PressureValue}>10000) \rightarrow \Diamond(\text{PressureDetector.Violation==1})) \]

   **Analysis Results:** Spin verified Constraint 8 after 372,246 transitions, storing 212,582 states and consuming 27.737 Megabytes of memory.

2. **Detector-Corrector Pattern Constraint:**

   When the detector detects a violation, the corresponding local fault handler has to be notified.

   \[ \square(\text{"Detector violation"} \rightarrow \Diamond(\text{"Report fault to local fault handler"})) \]

   **Instantiated Constraint:** As described above, a detected pressure violation is denoted by the PressureDetector attribute Violation being true. As a result of the violation, the PressureDetector sends a fault identifier code “200” to the LocalPressureFaultHandler (which
is denoted by calling the method \textit{ReportLocalFault} of the \texttt{LocalPressureFaultHandler} with parameter value 200), as shown in Figure 12, state \textit{Violation}.

\begin{equation}
\Box((\text{PressureDetector.Violation}=1) \rightarrow \Diamond(\text{send}(\text{LocalPressureFaultHandler.} \text{ReportLocalFault}(200))))
\end{equation}

\textbf{Analysis Results:} Spin verified Constraint 10 after 373,769 transitions, storing 212,068 states and consuming 27.737 Megabytes of memory.

3. \textit{Detector-Corrector} Pattern Constraint:

When a fault is reported to the local fault handler, the global fault handler is eventually notified.

\begin{equation}
\Box(\text{"Local fault handler is notified of fault"} \rightarrow \text{\Diamond(\text{"Global fault handler is notified of fault"})})
\end{equation}

\textbf{Instantiated Constraint:} In the DFS conceptual model, if the fault identifier code “200” was reported to the \texttt{LocalPressureFaultHandler} (indicating a pressure violation), then the same fault identifier code has to be reported eventually to the \texttt{GlobalFaultHandler} (which is denoted by calling the method \textit{ReportGlobalFault} of the \texttt{GlobalFaultHandler} with parameter value 200).

\begin{equation}
\Box((\text{send}(\text{LocalPressureFaultHandler.} \text{ReportLocalFault}(200))) \rightarrow \text{\Diamond(\text{send}(\text{GlobalFaultHandler.} \text{ReportGlobalFault}(200)))))
\end{equation}

\textbf{Analysis Results:} Spin verified Constraint 12 after 369,108 transitions, storing 212,041 states and consuming 27.300 Megabytes of memory.

4. \textit{Fault Handler} Pattern Constraint:

If a fault message is sent to the global fault handler, then it should activate the appropriate user interface warning level if required.

\begin{equation}
\Box(\text{"Fault reported to global fault handler"} \rightarrow \text{\Diamond(\text{"Activate appropriate user interface warning level"})})
\end{equation}

\textbf{Instantiated Constraint:} As described above, a fault detected by the \texttt{PressureDetector} is modeled in our system with the fault identifier code “200”. This fault requires the activation of the appropriate user interface warning level. In the DFS conceptual model, the possible
values of the UserInterface attribute WarningLevel are zero (no warning) and one (warning). A warning is initiated by calling ActivateWarningLevel of the UserInterface with parameter value 1.

\[ \square((\text{send(GlobalFaultHandler.ReportGlobalFault(200))) \rightarrow (14)} \]
\[ \diamond((\text{send(UserInterface.ActivateWarningLevel(1)))) \]

**Analysis Results:** Spin verified Constraint 14 after 479,541 transitions, storing 233,502 states and consuming 28.324 Megabytes of memory.

5. **User Interface Pattern Constraint:**

When a message is sent to the UserInterface to activate a specific warning level, then eventually all the indicators corresponding to this level will be activated.

\[ \square(\text{‘‘Warning level sent to user interface’’} \rightarrow (15)} \]
\[ \diamond(\text{‘‘Appropriate indicators for warning level activated’’}) \]

**Instantiated Constraint:** After the user interface is activated, the corresponding indicator has to be activated also. In the DFS conceptual model, a light in the actuator DriverDisplay is represented by the DriverDisplay attribute DriverDisplayValue. The possible values for this attribute are zero (the light is off) and one (the light is on). Therefore, this last claim checks if, after a user interface activation, the attribute DriverDisplayValue of the DriverDisplay will eventually have the value one.

\[ \square((\text{send(UserInterface.ActivateWarningLevel(1)))) \rightarrow (16)} \]
\[ \diamond(\text{DriverDisplay.DriverDisplayValue==1}) \]

**Analysis Results:** When analyzing Constraint 16, Spin detected a counterexample after analyzing 52,245 states. From the counterexample, Minerva generated a sequence diagram in terms of the UML objects in the original UML model which can be seen in Figure 13 (messages not essential for understanding the counterexample have been elided). In the sequence diagram, the ComputingComponent first checks the operational status of the PressureSensor. The PressureSensor sends a CCOK() message to the ComputingComponent, indicating that the PressureSensor is working. However, the PressureDetector then reports a fault to the LocalPressureFaultHandler because the pressure value reported by the PressureSensor exceeds...
with 11,000 the system maximum of 10,000 Pa. The LocalPressureFaultHandler forwards the fault identifier code “200” to the GlobalFaultHandler, which, in turn, sends a ShutdownEmergency() message to the ComputingComponent and a warning message to the UserInterface. The UserInterface then erroneously sets the DriverDisplayLight to zero (i.e., turns off the light) by sending the message SetDriverDisplayValue(0) to the DriverDisplay, as depicted by the bottom-most directed arrow in Figure 13.

While the sequence diagram depicts the sequence of messages that lead to a violation of the property checked (i.e., that the display light was turned off instead of on as expected), it does not reveal why the error occurred. State diagram animation of the entire system revealed that the problem was an erroneous guard on a transition in the UserInterface state diagram. Figure 14 shows the UserInterface state diagram and the subset of the system animation trace pertaining to the UserInterface. Figure 14(a) shows a snapshot of one animation step of the trace, that corresponds to Step 3 in Figure 14(b). The guard on the bold transition in Figure 14(a), italicized for emphasis, unintentionally creates non-determinism in transitioning from the Check to the Idle state (i.e., both transitions have the same guard condition, so either transition could be taken if the guard condition is true). This non-determinism erroneously allows the warning light to be turned off (depicted by the message SetDriverDisplayValue(0) on the bold transition) when it should instead indicate a warning to the user. Figure 14(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 14(a), distinguishing which one of the transitions was taken.

**Corrective Actions:** We corrected the italicized guard in Figure 14(a) to compare the warning level to zero and regenerated the specification. Spin verified the claim after 538,961 transitions, storing 242,630 states and consuming 26.481 Megabytes of memory.

5 Related Work

Several other approaches exist that make use of patterns in the early stages of a software development process. For example, Gross and Yu [25] have investigated the relationship between
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”

(a) Snapshot of user interface state diagram animation

(b) Subset of system animation trace

Figure 14. Animation snapshot and trace pertaining to UserInterface state diagram
non-functional requirements and design patterns, and Robertson [44] discusses the use of event/use case modeling to identify, define, and access requirements process patterns. Sutcliffe et al. [49] describe how scenarios of use cases can be investigated to identify generic requirements for different application classes.

Independent of application domain, Coad et al. [10] performed initial work on object-oriented analysis patterns. More recently, Fowler [21] identified several analysis patterns that might be used to represent conceptual models of business processes, such as abstractions from accounting, trading, and organizational relationships. Fernandez and Yuan [20] introduced semantic analysis patterns, which include either a few use cases or a small set of requirements and capture dynamic behavior of the proposed solution in terms of state and sequence diagrams. In contrast to Fowler, semantic analysis patterns use a structured template, rather than an informal description style, to capture a pattern. Geyer-Schulz and Hahsler [23] also added more structure to their descriptions of analysis patterns and developed analysis patterns that focus on the domain of cooperative work and collaborative applications.

Finally, van Lamsweerde et al. developed KAOS (Knowledge Acquisition in autOmated Specification) [11], an approach to support a goal-driven requirements engineering approach by offering a metamodel for capturing initial requirements and a strategy for the requirements acquisition process. A rich formal language enables software engineers to capture functional as well as non-functional requirements in terms of both informal and formal specifications. In addition, KAOS offers goal patterns [11] to support the specification of common system behavior in temporal logic, and goal refinement patterns [12] that can be used to aide in the construction of complete and correct goal refinement graphs.

Others have identified software architecture patterns [46], database access patterns [29], fault-tolerant telecommunication system patterns [2], design patterns tailored to distributed real-time embedded systems (DREs) [41, 42, 43, 48], design patterns for avionics control systems [37], real-time design patterns [15, 18], security patterns [36], etc. In comparison to our work, none of the aforementioned approaches offers a combination of patterns to guide construction of conceptual models, templates for the formal specification of critical properties, and a formalization framework tailored to the software analysis process of embedded systems development.
6. Conclusions

This paper described the identification and application of object analysis patterns for embedded systems. In order to facilitate their use, we use a template similar in spirit to that used by Gamma et al. to describe design patterns, with appropriate modifications to reflect an emphasis on analysis-oriented information, rather than design. Using a previously developed UML formalization framework, UML diagrams constructed from the structural and behavioral templates in the object analysis patterns can be translated into Promela, the specification language for the Spin model checker and simulator. The Promela representation provides a means to validate the UML diagrams using Spin's simulation features. In addition, the object analysis patterns contain a “Constraints” field used to give templates for describing properties applicable to a model constructed using the given pattern. These templates, when instantiated, yield properties specified in terms of LTL. The UML diagrams, using the Promela representation, can be formally checked for adherence to these LTL properties using Spin's model checking capabilities. Any discrepancies found can be traced back to the original UML diagrams using a number of visualization techniques available with our formalization framework. Therefore, using a combination of the object analysis patterns with the UML formalization framework, an embedded systems developer has a means to perform roundtrip construction, analysis, and refinement of UML diagrams for embedded systems, while largely keeping the formal aspects of the process transparent to the developer. Feedback from industrial collaborators indicates that the object analysis patterns considerably improve the overall analysis development process of embedded systems.

In reflecting upon this work, several observations can be made. First, due to the limited number of classes typically used in embedded systems, even having a small number of frequently used object analysis patterns can greatly assist embedded system developers in identifying the key objects of an embedded system and capturing their behavior in conceptual models. Subsequently, these models can be rigorously analyzed for adherence to critical properties using our formalization framework. This analysis greatly reduces the likelihood of major conceptual errors being propagated into design or code. Industrial feedback indicates that UML is considered a much easier to use notation than text-based formal specification languages to specify the conceptual models. Second, the information provided in the pattern template enables developers to understand the consequences...
of a pattern application, as well as helps them avoid common errors even if a pattern is not fully applied. Next, the application of patterns among several systems leads to uniformity among the analysis models, thereby greatly enhancing the understandability of the systems and improving communication between the different members of development teams. In addition, code reuse is potentially facilitated not only at the analysis but also at the design and code levels. For example, if a developer must model global fault handling capabilities within a system, the Fault Handler Pattern is useful for creating structural and behavioral analysis models for a global fault handler. Each modeled global fault handler will have different constraints on its actual design and implementation that will vary in functionality and cost. These implementations, indexed by their functionality and costs, can be associated with the Fault Handler Object Analysis Pattern for future use. Finally, the constraints provided by the object analysis patterns used in conjunction with our formalization framework facilitate the formulation of claims to check the system for adherence to specific properties.

Current and future work includes extending the patterns in several directions. First, we are extending the patterns to capture timing information in the conceptual analysis model [32], a crucial aspect in the development of real-time embedded systems. We are also expanding the pattern repository to address other embedded systems domains. Additionally, we are investigating the inclusion of diagnostics patterns [51] and fault-tolerance patterns [7] that can be used to model different types of diagnostic checks and increase the fault-tolerance of a system, respectively, in this work. Finally, we are exploring optimization and abstraction techniques to make the analysis of large-scale models more manageable. Specifically, we are investigating automated abstraction techniques to keep the model checking portion of the analysis tractable.

7. Acknowledgements

The authors are grateful to Dr. Erik Kamsties (University of Essen), Min Deng, Ali Ebnenasir, and other members of the Software Engineering and Network Systems Laboratory at Michigan State University for helpful comments on this work. We also greatly appreciate the feedback from our collaborators at Eaton Corporation, Siemens (several sectors), Detroit Diesel Corporation, and General Dynamics Land Systems. In particular, Anthony Torre has been an invaluable source in providing sample projects and feedback for this work.
The first author completed a portion of this work while studying under an International Exchange Program through the University of Kaiserslautern and Michigan State University.

References


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


