A Framework for Modeling and Analyzing Fault-Tolerance[1]

Ali Ebnenasir Betty H.C. Cheng
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

The development of fault-tolerant systems continues to be difficult due to the evolving and crosscutting nature of fault-tolerance requirements. Fault-tolerance research has largely focused on *how* to design and implement fault-tolerant systems. Regardless of *how* fault-tolerance is provided, however, it is equally important to determine *what* constraints should be met both in the absence and in the presence of faults. In order to address this question, this paper describes a systematic method for modeling and analyzing fault-tolerance concerns in UML at the requirements analysis phase. First, we present an approach for use case analysis of fault-tolerance requirements based on two canonical families of constraints, namely *detection* and *correction* constraints. Second, we present a method for object analysis of fault-tolerance requirements where we refine our use case analysis model using two object analysis patterns, called *detector* and *corrector* patterns. These detector and corrector object analysis patterns are based on the formal definition of detector and corrector components, which have been proven to be necessary and sufficient for the design of fault-tolerance. Finally, we define an Object Computation Model to provide a formal semantics for our fault-tolerance object analysis patterns. The Object Computation Model can be instantiated to different target specification languages using previously developed UML formalization framework, Hydra. As a result, we are able to automatically generate formal specification for the UML diagrams comprising the composition of functional and fault-tolerance requirements, thus enabling the automated analysis of fault-tolerance. We demonstrate our analysis method in the context of an industrial automotive application.

Keywords: Requirements Analysis, Fault-Tolerance, Formal Methods Detectors, Correctors

[1]Email: ebnenasi@cse.msu.edu, chengb@cse.msu.edu
Web: `http://www.cse.msu.edu/~{ebnenasi,chengb}`
Tel: +1-517-355-2387, Fax: +1-517-432-1061
This work was partially sponsored by NSF grants EIA-0000433, CDA-9700732, CCR-9901017, and a grant from Michigan State University.

1 Introduction

The complexity of developing fault-tolerant systems is due to (i) the crosscutting and evolving nature of fault-tolerance requirements, and (ii) the fact that user requirements should be met both in the absence and in the presence of faults. To deal with this complexity, more work is needed to analyze the consistency of functional and fault-tolerance requirements in order to prevent the propagation of logical inconsistencies from the requirements to the design and coding phases. Towards this end, this paper presents a method for implementation-independent modeling and analysis of fault-tolerance requirements.

Numerous approaches exist for designing fault-tolerant systems [1–8], most of which focus on implementation constraints to support fault-tolerance. For example, Randall's seminal work [1] focuses on the use of recovery blocks in the implementation of fault-tolerant software systems. Also, in checkpointing and log-based recovery mechanisms [6], the fault-tolerant system first saves the state of the system computations (i.e., checkpointing), and then recovers to a legitimate state once an error state is detected. Schneider [4] presents a mechanism for the design of replicated servers in a client-server model. In such a replication-based approach, a deterministic replica management protocol coordinates the execution of the replicated server. All of these approaches have a common focus on the design of fault-tolerance against specific types of faults (e.g., fail-stop faults [4, 6]) and for particular architectures (e.g., client-server [4]). While fault-tolerance should eventually be considered at design time, we believe that, regardless of *how* fault-tolerance is implemented, developers should first consider *what* constraints a fault-tolerant system should meet, both in the presence and in the absence of faults. Such an emphasis on the analysis of fault-tolerance concerns prevents the propagation of inconsistencies of functional and fault-tolerance requirements to the design and coding phases. For example, in the development of a distributed database management system that is expected to be fault-tolerant against crash faults, restoring the state of the database after a node crashes may be inconsistent with preserving the integrity of the database during restoration.

We introduce a method for modeling and analyzing faults and fault-tolerance in UML at the requirements analysis level. Our approach separates functional and fault-tolerance concerns. We start with a *valid* model of a system (i.e., *fault-intolerant system model*) that meets its requirements in the absence of faults. Then we extend the fault-intolerant model by modeling faults and fault-tolerance requirements. Our approach comprises three main steps. First, we perform a use case based analysis of faults and error detection and correction. Second, we apply our newly developed fault-tolerance analysis patterns for error detection and correction towards creating UML class and state (respectively, sequence) diagrams for fault-tolerant systems. Finally, we generate formal specifications from the UML diagrams using the formal semantics of the fault-tolerance analysis patterns, and a previously developed semantics for the UML diagrams [9, 10].

Our method comprises the analysis of faults and fault-tolerance requirements at the use case level and at the object analysis level. Specifically, we extend the fault-intolerant use case model by modeling (i) faults as actors that may temporarily change the behavior of the system; (ii) the effect of faults as faulty use cases, and (iii) fault-tolerance requirements as a set of *detection* and *correction* constraints that must be met in the absence and in the presence of faults. We also model use cases that detect (respectively, correct) corresponding detection (respectively, correction) constraints. At the object analysis level, we use the notion of state perturbation to model faults. State perturbation has been used in previous work [11–13] for modeling different types of faults with different behavioral natures. In addition to modeling faults, we extend the notion of detector and corrector components from [14, 15] to develop detector and corrector patterns for modeling fault-tolerance requirements. Arora and Kulkarni [14, 15] have shown that detectors and correctors are necessary and sufficient for specifying fault-tolerance requirements. As such, detector and corrector patterns serve to modularize fault-tolerance requirements. Such modularization helps system modelers in reasoning about fault-tolerance, and in tracing the fault-tolerance concerns at different stages of the software lifecycle. Note that, in this paper, detector and corrector patterns provide a systematic approach for refining fault-tolerance requirements instead of providing patterns for designing a system. To provide a framework for automated analysis, we introduce a formal semantics for the object analysis of UML models. Our formal semantics provides an Object Computation Model (OCM) for object-oriented models in UML that can be translated to different target specification languages. In particular, we generate the specification of the fault-tolerant UML models in Promela [16]. Such formalization enables the automatic analysis of fault-tolerant systems, where we create a formal model of the fault-tolerant system and use model checkers (e.g., SPIN [17]) and synthesis tools (e.g., [18]) for reasoning about fault-tolerance requirements.

We demonstrate our analysis method in the modeling of an *adaptive cruise control* (ACC) system in UML. Specifically, we illustrate how we model faults and how we specify detection and correction constraints to generate a fault-tolerant use case model of the ACC system. Subsequently, we refine the use case model of the ACC system using detector and corrector patterns. Using our approach, developers can model fault types that would be difficult (if not impossible) to model using existing approaches [8, 19, 20]. For example, transient faults may perturb the state of a buffer and cause temporary buffer overflow that is difficult to model using the notion of component failure discussed in [8, 19]. Further, the notion of fault-tolerance in the existing approaches is mostly based on using replication to *mask* faults (i.e., masking fault-tolerance); less emphasis has been put on other forms of fault-tolerance (e.g., failsafe fault-tolerance [21] and self-stabilization [11]). We illustrate how to model such levels of fault-tolerance using our approach. The remainder of this paper is organized as follows. Section 2 presents preliminary concepts, where we introduce the Object Computation Model for UML models. Section 3 illustrates modeling of faults and fault-tolerance at the use case level in UML. Section 4 demonstrates how we model faults in UML object models. Sections 5 and 6 respectively present the detector and corrector patterns that are applied for solving detection and correction constraints at the object analysis level. Section 7 presents a translation of detector and corrector patterns from OCM to the Promela modeling language. Such a formalization enables analysts to use the model checker SPIN [17] for verifying the *interference-freedom* of functional and fault-tolerance concerns; i.e., the detection/correction constraints and functional requirements are met in the presence of each other. Section 8 summarizes the requirements analysis method presented in this paper. Section 9 illustrate the proposed analysis method in the context of a Diesel Filter System. Section 10 discusses and compares related work. Finally, Section 11 gives concluding remarks and discusses future work.

2 Preliminaries

In this section, we present basic concepts of (i) UML models; (ii) our formal representation of object-oriented systems in UML, and (iii) the definition of faults and fault-tolerance.

2.1 UML

The Unified Modeling Language (UML) [22] provides visual artifacts for system specification. Specifically, UML presents a visual language for requirements elicitation, and structural and behavioral system specification. To obtain user requirements, UML has adopted Jacobson's notion of use cases [23]. A use case specifies scenarios of using the subject system which represent *user goals* for the system. At the design level, UML comprises a set of diagrams for representing the structure and the behavior of a system. At the structural design level, modelers use class, object, and package diagrams to represent the structure of a system and its components. At the behavioral design level, UML provides state and sequence diagrams for representing system behaviors.

In order to enable formal analysis (e.g., model checking) of UML models, it is necessary to have formal specification of the UML models. There exist approaches for generating domain-specific formal specification from UML models. For example, McUmber and Cheng [9] present a method for automatic transformation of UML behavioral models to Promela [16] for requirements analysis of embedded systems. We extend McUmber and Cheng's technique to generate formal specification of the UML models of fault-tolerant systems in Promela (see Section 7), which enables us to use the SPIN model checker simulation environment.

We use a subset of UML in modeling and specifying fault-tolerant systems. More specifically, we use UML artifacts for use case analysis of fault-tolerant systems, where we model faults and fault-tolerance requirements at the use case level. As we refine our use case model, we use the UML subset for object modeling and analysis towards creating the object analysis model of fault-tolerant systems.

2.2 Object Computation Model

In this section, we represent the Object Computation Model (OCM) that we have developed (in [24]) to provide a computational semantics for object-oriented models in UML. Each object[2] consists of a set of state variables (with finite domains) and a set of methods (i.e., actions). A state of an object is a valuation to its state variables. (The threads of control of an object can be considered as state variables as well.) The finite state space of an object O, denoted S_O, is the set of all states of that object. An OCM $\mathcal{M}$ is a triple $\langle \mathcal{O}, \mathcal{T}, \mathcal{P} \rangle$, where $\mathcal{O}$ is a set of objects $O_1, \cdots, O_k$ $(1 \leq i \leq k)$, $\mathcal{T}$ is a relation that represents the interconnection of the objects in $\mathcal{O}$, and $\mathcal{P}$ is a set of state predicates. A state predicate, X, is any subset of $S_{\mathcal{M}} = S_{O_1} \times S_{O_2} \times \cdots \times S_{O_k}$, where S_{O_i} is the state space of O_i $(1 \leq i \leq k)$. A global state of $\mathcal{M}$ is a k-tuple in $S_{\mathcal{M}}$. The state space $S_{\mathcal{M}}$ of $\mathcal{M}$ comprises all global states. We define the set of methods of an object O_i by a set of transitions $\delta_{O_i} \subseteq S_{\mathcal{M}} \times S_{\mathcal{M}}$ $(1 \leq i \leq k)$. A computation of an OCM $\mathcal{M}$ is an infinite sequence $\langle s_0, s_1, \cdots \rangle$ of global states in which each transition (s_j, s_{j+1}) $(0 \leq j)$ belongs to some object O_i $(1 \leq i \leq k)$. Each computation is maximal; i.e., if it is finite then the last state represents a valid halting scenario of the system model. If the requirements allow a finite computation $c = \langle s_0, s_1, \cdots, s \rangle$ to be extended to an infinite sequence by including the transition (s, s) in the set of transitions of $\mathcal{M}$, then c is a halting computation. Otherwise, c is a deadlock computation and s is a deadlock state; i.e., a state with no outgoing transitions.

Encapsulation model. The encapsulation of methods and state variables of objects imposes a

[2]An object is a behavioral representation of a class.

set of read/write restrictions on each object. An object is allowed to read/write all its private and public variables. Although an object cannot read/write private variables of other objects, it may be able to read/write the public variables of other objects. The set of variables that are readable for an object determines the locality of that object. A local state predicate represents a set of global states in the locality of an object. A global state predicate consists of a set of global states of an OCM $\mathcal{M}$.

Object association model. In the analysis of fault-tolerant systems, we focus on the essence of inter-object communications rather than the type of inter-object associations (e.g., inheritance, aggregation, etc.). Hence, at the OCM level, we abstract out the semantics of inter-object associations (e.g., inheritance, aggregation, etc.) specified at the UML level. For example, if an object A inherits its behaviors from another object B then at the OCM level, the relations of A with B and other objects are modeled in the context of the variables of B that A is allowed to read/write and vice versa. The fact that object A inherits the behaviors of B has less effect on the results of analyzing fault-tolerance concerns. While there exist formal methods [25] and refinement mapping techniques [26–28] for capturing inheritance in an object-oriented system, we abstract out the type of inter-object associations in OCM.

The complexity of analyzing fault-tolerance requirements could vary from polynomial to undecidable [21, 29, 30], depending on the computational model. We have adopted a shared memory (interleaving) model (i.e., OCM) for the analysis of fault-tolerance requirements to reduce the complexity of automated analysis of fault-tolerance requirements. Sound techniques [26, 28, 31] can be used to refine the fault-tolerant conceptual model of a system from OCM to a message-passing model.

2.3 Properties of Interest

In this section, we formally define functional requirements in OCM. We focus on the set of user requirements that can be classified in terms of two categories of *safety* and *liveness* requirements [32]. Subsequently, we define what it means for an OCM to meet its safety and liveness requirements.

We formally represent the functional requirements as a set $\mathcal{R}$ of sequences $\langle s_0, s_1, \cdots \rangle$ of global states in OCM. In an OCM $\mathcal{M}$, the set of functional requirements $\mathcal{R}$ is an intersection of the safety and liveness requirements. A safety requirement stipulates that nothing bad will ever happen. For example, if the objects in an OCM $\mathcal{M}$ share a critical section then the safety of the model is violated if more than one object enters the critical section simultaneously. We follow Alpern and Schneider [32] where we represent the safety requirements of an OCM $\mathcal{M}$ as a set of *finite* sequence of transitions $\langle s_0, \cdots, s_n \rangle$ that must not be executed by $\mathcal{M}$; i.e., a set of *bad* sequences. Intuitively, a liveness requirement specifies that something good will eventually happen. We represent the liveness requirements as a set of infinite sequences $\langle s_0, s_1, \cdots \rangle$ of states, where each sequence in the set has a suffix that is in the liveness requirements. For example, in reactive systems[3], *leads-to* properties [33] are often used to specify the liveness requirements, where a *leads-to* property stipulates that the system will eventually respond to the stimuli of its environment.

The validation of an OCM $\mathcal{M}$ depends on the validation of the computations of $\mathcal{M}$. A computation c of an OCM $\mathcal{M}$ validates the safety and liveness requirements iff (if and only if) $c \in \mathcal{R}$. The OCM model $\mathcal{M}$ validates the safety and liveness requirements iff all its computations validate the safety

[3]Reactive systems have non-terminating behaviors and always react to the stimuli of their environment (e.g., network protocols, controlling software of embedded systems).

and liveness requirements. A computation c violates safety requirements (i.e., c is a safety-invalid computation) if c includes a sequence of transitions that is ruled out by the safety requirements; i.e., includes a bad sequence. A computation c violates liveness requirements (i.e., c is a liveness-invalid computation) if c includes (i) a deadlock state, or (ii) a non-progress cycle of transitions in which nothing good ever happens. An OCM $\mathcal{M}$ is invalid with respect to its safety requirements iff $\mathcal{M}$ includes a safety-invalid computation. An OCM $\mathcal{M}$ is invalid with respect to its liveness requirements iff $\mathcal{M}$ includes a liveness-invalid computation. An OCM $\mathcal{M}$ is valid iff $\mathcal{M}$ includes no safety-invalid and no liveness-invalid computations. An invariant $I_{\mathcal{M}}$ for an OCM $\mathcal{M}$ is a non-empty global state predicate such that all computations starting in $I_{\mathcal{M}}$ validate the safety and liveness requirements.

2.4 Behavioral Association of UML and OCM

In this section, we define the underlying semantics of UML behavioral models in OCM. Specifically, for an UML model M and its corresponding OCM $\mathcal{M}$, a state in an UML state transition diagram represents a state predicate in $S_{\mathcal{M}}$ (i.e., a *mode* of operation of an object O). Each transition in an UML state diagram represents a set of transitions in $S_{\mathcal{M}} \times S_{\mathcal{M}}$ in the underlying OCM $\mathcal{M}$. A *sequence diagram* in M captures an equivalence class of computations in the corresponding OCM $\mathcal{M}$ that represents a requirements scenario. A deadlocked sequence diagram (i.e., deadlock scenario) represents a set of deadlocked computations in the corresponding OCM $\mathcal{M}$. An invalid scenario represents a set of invalid computations of the corresponding OCM $\mathcal{M}$. Thus, a safety-invalid sequence diagram represents a scenario that violates the safety requirements. Likewise, a liveness-invalid sequence diagram represents a scenario that violates the liveness requirements.

2.5 Faults and Fault-Tolerance

In this section, we illustrate how we formally represent faults in OCM. Also, we define three levels of fault-tolerance requirements based on the extent up to which a system meets its safety and liveness requirements in the presence of faults.

In an OCM $\mathcal{M}$, we represent a fault-type f as a set of transitions in the entire state space of $\mathcal{M}$[4]. In other words, faults f are transitions that perturb the state of a system in an uncontrollable manner; i.e., the system does not have execution control over fault transitions. Such a notion of state perturbation is expressive enough to represent different types of faults (e.g., stuck-at, crash, fail-stop, omission, Byzantine, and design flaws) with different behavioral natures (i.e., transient, intermittent, and permanent) [11–13]. A computation of an OCM $\mathcal{M}$ in the presence of faults f is a maximal sequence $\langle s_0, s_1, \cdots \rangle$ of global states that satisfies the following conditions: (1) every transition (s_j, s_{j+1}), where $0 \leq j$, belongs either to f or to one of the objects in $\mathcal{M}$, and (2) the number of fault occurrences in that computation is finite. The latter requirement is the same as that made in previous work [11, 12, 34, 35] to ensure that eventually recovery can occur. The boundary up to which the state of $\mathcal{M}$ could be perturbed by fault f is called the f-span (i.e., fault-span) of $\mathcal{M}$. For an OCM $\mathcal{M}$, we represent its fault-span by a state predicate that is a superset of its invariant $I_{\mathcal{M}}$. Next, we extend the definition of three levels of fault-tolerance from [13, 30] in OCM. The definition of these three levels of fault-tolerance is based on the extent up to which user

[4]We follow the approach of [13] in modeling faults as a set of transitions that are triggered non-deterministically.

requirements are met in the presence of faults.

Failsafe fault-tolerance. Intuitively, failsafe fault-tolerance requires that nothing bad ever happens even in the presence of faults. An OCM $\mathcal{M}$ is failsafe f-tolerant (i.e., fault-tolerant for fault-type f) from $I_{\mathcal{M}}$ iff starting from $I_{\mathcal{M}}$ (1) in the absence of faults f, $\mathcal{M}$ validates its safety and liveness requirements, and (2) in the presence of faults f, $\mathcal{M}$ guarantees to validate its safety requirements.

Nonmasking fault-tolerance. Nonmasking fault-tolerance requires that recovery to the invariant is provided if faults perturb the state of a system outside its invariant. An OCM $\mathcal{M}$ is nonmasking f-tolerant from $I_{\mathcal{M}}$ iff starting from $I_{\mathcal{M}}$ (1) in the absence of faults f, $\mathcal{M}$ validates its safety and liveness requirements, and (2) in the presence of faults f, every computation of $\mathcal{M}$ that starts in f-span of $\mathcal{M}$ will eventually reach a state in $I_{\mathcal{M}}$.

A special case of nonmasking fault-tolerance where the fault-span is equal to the entire state space is called self-stabilization. A system is self-stabilizing if it eventually recovers to its invariant from every state in its state space [11]. Self-stabilization has important applications in the development of resilient network protocols that survive in the presence of transient faults; i.e., faults that occur spontaneously and then disappear. This paper does not specifically focus on self-stabilization, nonetheless, the techniques used for the analysis of nonmasking fault-tolerance can be applied in the analysis of self-stabilizing systems.

Masking fault-tolerance. Masking fault-tolerance requires that (i) nothing bad will ever happen, and (ii) recovery to the invariant is guaranteed if faults perturb the state of a system outside its invariant. An OCM $\mathcal{M}$ is masking f-tolerant from $I_{\mathcal{M}}$ iff starting from $I_{\mathcal{M}}$ (1) in the absence of faults f, $\mathcal{M}$ validates its safety and liveness requirements; (2) in the presence of faults f, (a) every computation of $\mathcal{M}$ that starts in the f-span of $\mathcal{M}$ will eventually reach a state in $I_{\mathcal{M}}$, and (b) no computation of $\mathcal{M}$ that starts in the f-span of $\mathcal{M}$ violates the safety requirements.

Notation. Whenever the OCM $\mathcal{M}$ is clear from the context, we will omit it; thus, "$\mathcal{M}$ is failsafe (respectively, nonmasking or masking) f-tolerant" abbreviates "$\mathcal{M}$ is failsafe (respectively, nonmasking or masking) f-tolerant from $I_{\mathcal{M}}$".

3 Use Case Analysis of Fault-Tolerant Systems

In this section, we present our method for specifying faults and fault-tolerance requirements at the use case analysis level. First, we illustrate how to model faults, the impact of faults on a system, and fault-tolerance requirements. Then we demonstrate our approach in the context of an adaptive cruise control (ACC) system. We use the ACC system as a running example in the rest of the paper.

In use case analysis of fault-tolerant systems, we separate the use case modeling of functional requirements from the modeling of fault-tolerance requirements. Specifically, we assume that the input to our analysis method is the use case model of the *fault-intolerant* system that has been validated with respect to functional user requirements. In addition to the regular use cases, the fault-intolerant use case model should include (i) a set of misuse cases [36], and (ii) a global invariant constraint (see Figure 1). A misuse case specifies behaviors that must never appear in the set of system functionalities. We use misuse cases to specify the safety requirements of the fault-intolerant system model; i.e., bad things that must never happen. Such a method of modeling safety requirements using misuse cases can also be found in previous work [37] (see use cases with inverted colors in Figure 1). The invariant specifies a global constraint that the system satisfies in the absence of faults; i.e., if the invariant constraint holds then the user requirements will be met. If the invariant does not hold then there will be no guarantees that the system will meet its requirements. Therefore, depending on the problem at hand, the analysts and the system users should collaborate in specifying a global invariant constraint before modeling fault-tolerance requirements. The specification of the invariant constraint simplifies reasoning about the behavior of fault-tolerant systems. Although identifying an invariant constraint is not an easy task, there exist systematic (and automatic) approaches for generating invariant constraints from user requirements [38, 39].

Specifying faults and their impact on the subject system. We model faults as actors and model the impact of faults on the system as faulty use cases. A fault-type (e.g., fail-stop, crash, and Byzantine) is a type of event that temporarily takes the control of the system execution. In other words, a system that is subject to faults does not have control over the occurrence of faults. Thus, each fault-type can be considered as a logical entity whose goal is to perturb the state of the system. Hence, we model each type of faults as an actor whose actions may not be necessarily towards the system goals (see the filled-head actor in Figure 1). Also, we model the effect of faults on the system as faulty use cases (see the shadowed use case in Figure 1). A faulty use case represents a set of behaviors exhibited by the system due to the actions of a fault actor that may or may not result in the violation of (safety and liveness) requirements. A faulty use case may be a misuse case, thus *directly* leading to the violation of the safety requirements. Also, a faulty use case may perturb the system outside its invariant from where the system itself may exhibit a misuse case and *indirectly* violate the safety requirements. Furthermore, when a faulty use case perturbs the state of the system outside the invariant, the system may either (i) deadlock, or (ii) trap in a non-progress cycle.

Specifying fault-tolerance. In order to specify fault-tolerance at the use case level, we identify two canonical families of constraints, namely detection and correction constraints, that must be met both in the presence and in the absence of faults. Such constraints (i) separate fault-tolerance requirements from functional requirements, and (ii) provide traceability from use case level to the object analysis level and further to design and coding.

We use detection constraints to model failsafe fault-tolerance. A failsafe fault-tolerant system must not execute any misuse cases even in the presence of faults. Thus, if there exist any behavioral similarities[5] between the set of misuse cases and a faulty use case UC_f then the analysts should identify the *conditions* under which UC_f executes. Such conditions represent constraints that the system should *detect* (see Figure 1) to prevent the violation of safety requirements. Note that the system cannot prevent the occurrence of faults, however, the system can control the preconditions that lead to the occurrence of faults. The detection constraints model such preconditions.

We use correction constraints to model nonmasking fault-tolerance. If faults perturb the system outside its invariant then there will be no guarantees that the system will meet its requirements. Thus, to provide nonmasking fault-tolerance (i.e., recovery in the presence of faults), we need to identify conditions under which the invariant constraint will eventually hold. We model such conditions as *correction constraints*. The correction constraints specify postconditions that should

[5]Previous work [40–42] provides techniques for behavioral specification of use cases that can be used for checking behavioral similarities.

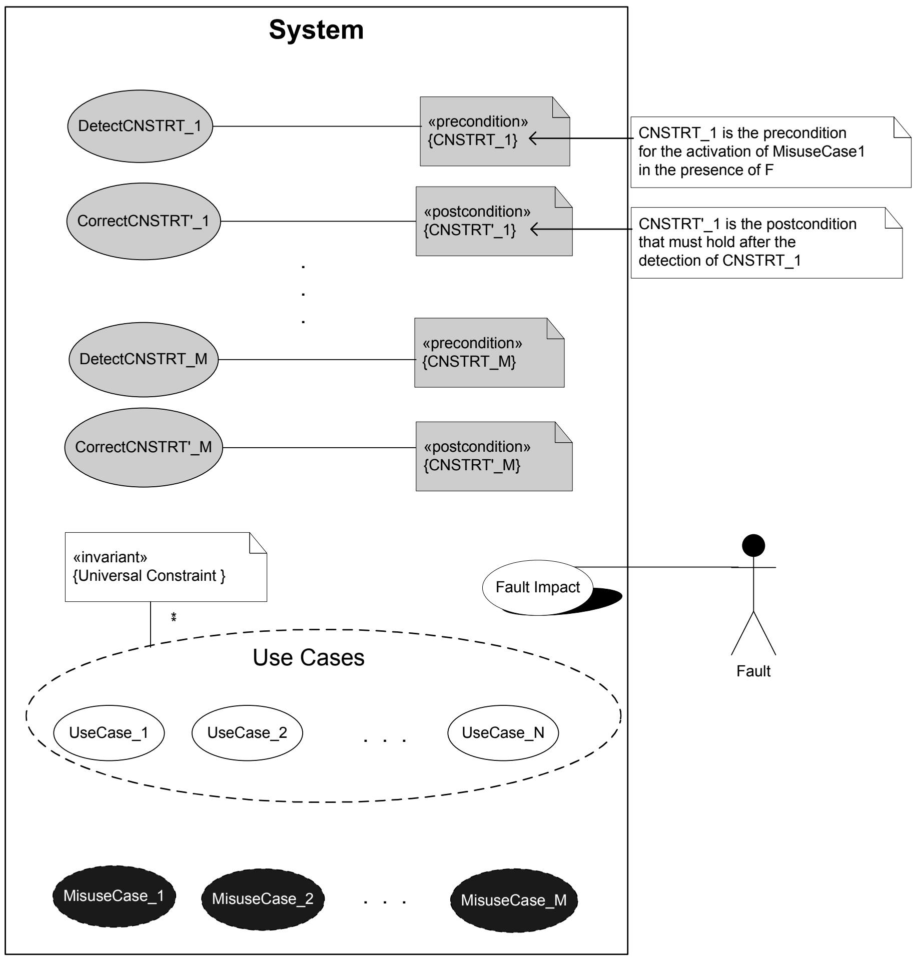

System
DetectCNSTRT_1
«precondition» {CNSTRT_1}
CNSTRT_1 is the precondition for the activation of MisuseCase1 in the presence of F
CorrectCNSTRT'_1
«postcondition» {CNSTRT'_1}
CNSTRT'_1 is the postcondition that must hold after the detection of CNSTRT_1
DetectCNSTRT_M
«precondition» {CNSTRT_M}
CorrectCNSTRT'_M
«postcondition» {CNSTRT'_M}
«invariant» {Universal Constraint }
*
Fault Impact
Fault
Use Cases
UseCase_1
UseCase_2
UseCase_N
MisuseCase_1
MisuseCase_2
MisuseCase_M

Figure 1: Use case diagram.

hold after the occurrence of faults. Note that the only requirement of nonmasking fault-tolerance is recovery, and the safety requirements may be violated during recovery.

We use both detection and correction constraints to model masking fault-tolerance. Since a masking fault-tolerant system should simultaneously guarantee safety and recovery, we use (1) correction constraints to model recovery to the invariant, and (2) detection constraints to model safety in the presence of faults. Therefore, the specification of masking fault-tolerance at the use case level is indeed the conjunction of specifying failsafe and nonmasking fault-tolerance simultaneously.

Comment on the complexity of identifying detection and correction constraints. Detection and correction are problems that should be modeled and specified during the analysis of fault-tolerance requirements. It is the responsibility of all stake holders to specify detection and correction constraints together. We believe that analysts should provide abstract and platform-independent solutions for specifying detection and correction constraints, while users provide domain knowledge to customize the specification of such constraints. As a result, it will be feasible to investigate potential inconsistencies between satisfying detection and correction constraints and satisfying system functionalities. While the identification of detection and correction constraints may prolong the requirements analysis phase, we believe that it is worth to analyze fault-tolerance requirements at the early stages of the software lifecycle instead of paying a higher cost in the design and implementation phases for resolving the potential inconsistencies of functional and fault-tolerance requirements. In Sections 5 and 6, we respectively present detector and corrector analysis patterns that provide a strategy for specifying detection and correction constraints (i.e., fault-tolerance requirements) at the object analysis level. Next, we demonstrate our use case analysis method for an adaptive cruise control system.

3.1 Case Study: Adaptive Cruise Control

In this section, we demonstrate our use case analysis method in the context of an adaptive cruise control (ACC) system (see Figure 2). Specifically, we first introduce a simplified version of the (fault-intolerant) model of the ACC system (from [43]). Then we illustrate how we generate a use case model in the presence of faults, and how we model fault-tolerance using detection and correction constraints.

Adaptive cruise control (ACC) system. The ACC system interacts with three actors, driver, car, and radar (see Figure 3). The driver uses the ACC system either in the standard cruise mode or in the adaptive cruise mode. In the standard cruise mode, the car should have a desired speed determined by the driver (i.e., setpoint). In the adaptive mode, the car adapts its distance with the front vehicle to avoid collision. The radar detects a target vehicle in its Range, and measures (i) the speed of the target vehicle, and (ii) the distance to the target vehicle (see Figure 3). The car is responsible for engine management so that the actual speed of the car becomes equal to the setpoint. Also, the car (i) updates its speed by acceleration or deceleration (see Figure 3), and (ii) sends its current speed to the radar upon request.

The ACC system has different modes of operation (see Figure 2). When the radar detects a target vehicle (i.e., *target* mode), the ACC system will be in either one of the following modes: *closing*, *coasting*, *matching*, *alarm*, or *disengaged* mode. The ACC system enters the closing mode once the radar detects the target vehicle. In the closing mode, the goal is to control the way that the car approaches the target vehicle, and to keep the car in a fixed *trail distance* from the target vehicle with a zero relative speed. The *trail distance* is the distance that target vehicle travels in a fixed amount of time (e.g., 2 seconds). The distance with the target vehicle must not be less than a *safety zone*, which is 90% of the *trail distance* (see Figure 2). In the *coasting* mode, the car should keep its forward movement without throttle. The ACC system calculates a *coasting distance* that is the distance at which the car should start decelerating in order to achieve the trail distance. When the car reaches the trail distance, the relative speed of the car should be zero; i.e., the speed of the car matches the speed of the target vehicle (i.e., *matching* mode). Whenever the driver applies the brake, the ACC system must disengage the *cruise* mode; i.e., *disengaged* mode. In cases where the speed of the car is so fast that a collision is unavoidable then the ACC system enters the *alarm* mode where it raises an alarm for the driver and must disengage the cruise control.

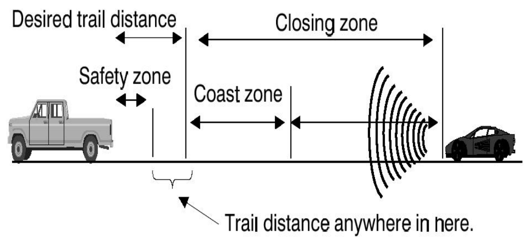


Figure 2: The adaptive cruise control system.

Safety and liveness requirements of the ACC system. We represent the use case model of the fault-intolerant ACC system in Figure 3 along with the misuse cases that specify the safety requirements. The misuse cases comprise the situations where (i) the car enters the safety zone, or (ii) the car accelerates in the coasting mode. In other words, the misuse cases specify the conditions where the safety requirements of the ACC system are violated. The liveness requires that if the ACC system receives a signal from the brake subsystem indicating that the driver has applied the brakes then it will eventually disengage the cruise control system.

The invariant of the ACC system. We represent the invariant of the ACC system in terms of the constraint I_{ACC} in OCM, where

$$\begin{aligned} I_{ACC} = \{s : (&target(s) \Rightarrow ((closing(s) \vee coasting(s) \vee matching(s) \vee alarm(s) \vee disengaged(s)) \wedge \\ &(safeyZone \leq distance(s)) \ \wedge ((\neg target(s) \wedge cruise(s)) \Rightarrow resume(s)) \wedge \\ &(Brakes(s) \Rightarrow disengaged(s)) \wedge (v(s) > v_{max} \Rightarrow alarm(s))\} \end{aligned}$$

Notation. $var(s)$ denotes the value of a variable var in a state s.

The constraint I_{ACC} is specified in terms of a set of variables that represent system modes and parameters. These variables will be allocated to different objects of the ACC system in the object analysis phase. The constraint I_{ACC} specifies a set of states, where (i) if a target vehicle has been detected then the ACC system should be in *closing*, *coasting*, *matching*, *alarm*, or *disengaged* mode and the car must not be in the safety zone; (ii) the distance with the target vehicle should always be greater than the safety zone distance; (iii) if the ACC system is in the *cruise* mode and the target is lost then ACC will go to the *resume* mode; (iv) if the driver applies the brakes then the ACC system must be in the *disengage* mode, and (v) if the closing speed of the car is greater than a maximum speed, v_{max}, then the system must alarm the driver of a potential collision.

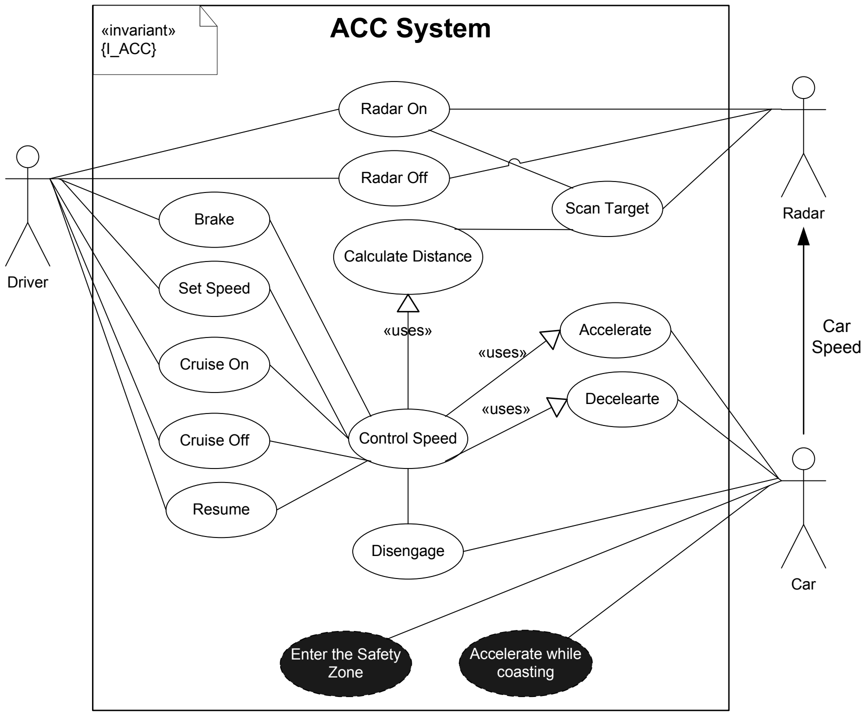


Figure 3: The use case diagram of the ACC system.

Faults and fault-tolerance in the ACC system. Two types of faults may perturb the state of the ACC system (see Figure 4). First, the fault-type f_1 may perturb the state of the car to the accelerating state. Second, the fault-type f_2 may arbitrarily change the mode of the ACC system from disengaged to NOT disengaged. We have modeled each fault-type as an actor. Also, we associate each fault-type with its impact on the system that is modeled as a faulty use case (the use cases with the gray spectrum in Figure 4).

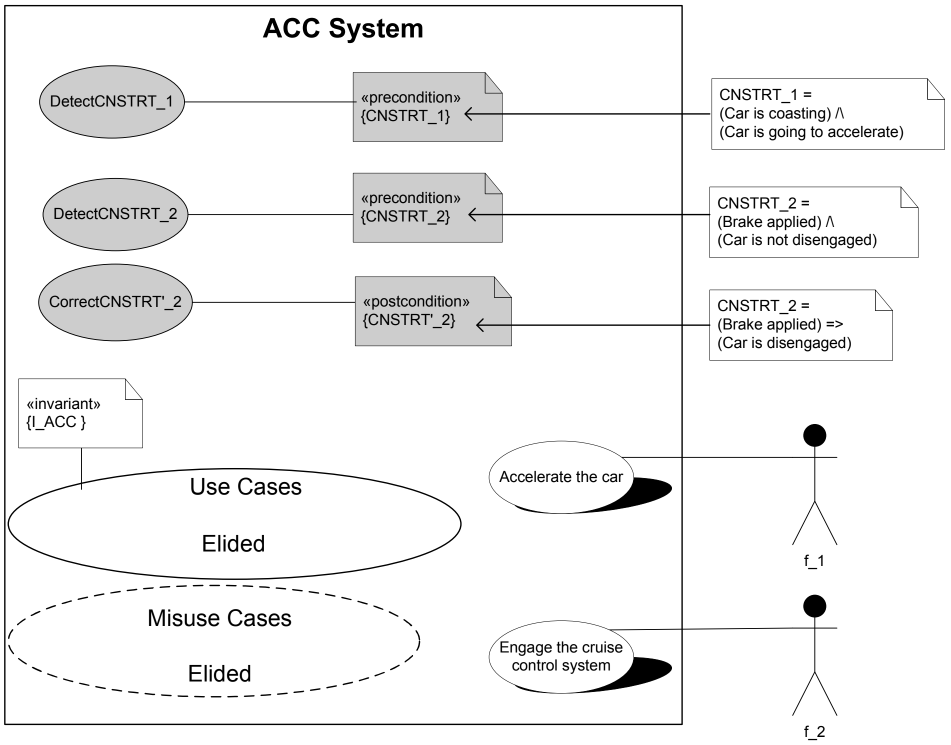

ACC System
DetectCNSTRT_1
«precondition» {CNSTRT_1}
CNSTRT_1 = (Car is coasting) /\ (Car is going to accelerate)
DetectCNSTRT_2
«precondition» {CNSTRT_2}
CNSTRT_2 = (Brake applied) /\ (Car is not disengaged)
CorrectCNSTRT'_2
«postcondition» {CNSTRT'_2}
CNSTRT_2 = (Brake applied) => (Car is disengaged)
«invariant» {I_ACC }
Use Cases Elided
Misuse Cases Elided
Accelerate the car
Engage the cruise control system
f_1
f_2

Figure 4: The use case diagram of the ACC system including faults and fault-tolerance use cases.

In order to model failsafe f_1-tolerance, we first explore conditions under which fault f_1 may directly violate the safety requirements; i.e., f_1 may directly cause the execution of a use case that is behaviorally similar to some misuse case. Since perturbing the state of the car to the accelerating state does not directly enter the car to the safety zone, f_1 will not cause the execution of the misuse case Enter the safety zone (see Figure 4). Also, being in the accelerating state does not mean that the car has accelerated. Thus, f_1 does not directly execute the misuse case Accelerate while coasting. Based on the above discussion, f_1 will not *directly* violate the safety requirements. Second, we need to investigate cases where f_1 perturbs the system to states from where the ACC system itself executes some misuse case. Such a scenario occurs when the ACC system is in coasting mode and f_1 occurs. In this case, fault f_1 perturbs the state of the ACC system to the accelerating state while the system is in coasting mode. Now, if the ACC system accelerates then the misuse case Accelerate while coasting will be executed; i.e., the safety requirements will be violated by the actions of the ACC system after the occurrence of f_1. Therefore, before acceleration, the ACC system must detect whether or not it is in the coasting mode, which results in the identification of the detection constraint CNSTRT_1 in Figure 4.

In order to model nonmasking f_2-tolerance, we need to identify the correction constraint that must be met in the presence f_2. The fault f_2 may arbitrarily change the mode of the ACC system from disengaged to not disengaged. Since the invariant of the ACC system stipulates that *if the brake is applied then the system must be disengaged*, the correction constraint that must be met will be equal to (*Brake applied*) $\Rightarrow$ (*Car is disengaged*) (see CNSTRT'_2 in Figure 4).

4 Modeling Faults in the Object Model

In this section, we illustrate how to model faults in (structural and behavioral) UML models. First, we explain how we formally represent UML class diagrams. We use such formal representation to simplify our presentation in the rest of the paper. Second, we demonstrate how to add faults to UML models. Third, we define what we mean by an UML model in the presence of faults. Finally, we demonstrate our modeling approach in the context of the ACC system.

Fault-intolerant UML models. We represent the class diagram of an UML model M as a tuple $\langle \mathcal{V}, \mathcal{G} \rangle$, where $\mathcal{V}$ is a set of classes $\langle c_1, \cdots, c_n \rangle$ in M, and $\mathcal{G} = \{\langle c_i, c_j \rangle\}$ is a relation that represents the association between classes c_i and c_j $(1 \leq i, j \leq n)$. We denote the state transitions diagram of each class c_i by S_{d_i} $(1 \leq i \leq n)$.

Adding faults to UML models. We add faults to UML models by adding fault transitions to state diagrams. Given a fault-type f, we model the effect of f on the state transition diagram S_{d_i} of each class c_i by introducing a new set of fault transitions in S_{d_i}. In fact, we define the impact of f on c_i as a new fault-type f_i that perturbs the state of the state transition diagram of c_i (see Figure 5). The difference between fault transitions and the regular transitions in an UML state diagram is that an object of c_i does not have control over the occurrence of faults f_i (see dotted arrows in Figure 5), whereas the execution of regular transitions (see solid arrows in Figure 5) is controlled by the thread of execution in that object. When faults perturb the state of an object outside its invariant, that object may reach a cycle (see ErrorState_1 and ErrorState_2 in Figure 5) or a deadlock state (see ErrorState_3 in Figure 5).

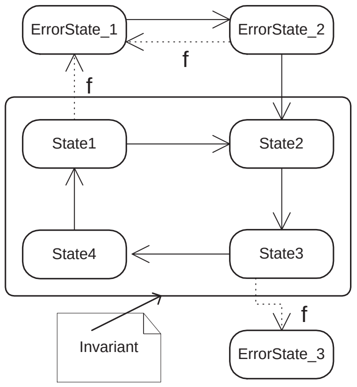


Figure 5: Modeling conditional faults.

Depending on the occurrence of faults, we classify faults into two types of *conditional* and *arbitrary* faults. A *conditional fault-type* is a fault-type that may occur only in particular states of the state transition diagram of an object (see Figure 5). The object does not have control over the occurrence of f, however, the occurrence of f depends on the current state of an object. For example, in an electronic circuit board, fault transitions may change the value of some registers if the voltage level falls below a certain threshold. An *arbitrary fault-type* may occur at any state and there exist no preconditions for its occurrence. For example, environmental noise may arbitrarily change the value of a register in its domain. We model an arbitrary fault-type as a separate fault state transition diagram (e.g., F_1 in Figure 6) that executes concurrently with an object state machine (e.g., S_1 in Figure 6). In Figure 6, the transitions of the arbitrary faults in F_1 may trigger at any state of the state diagram S_1.

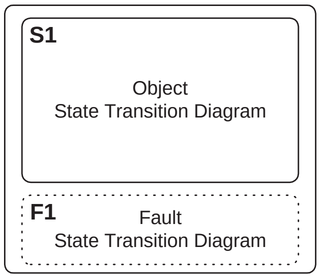


Figure 6: Modeling arbitrary faults.

In UML class diagrams, we model each fault-type as a method (see Figure 7). Fault methods may be executed arbitrarily and perturb the state of an object. One approach to distinguish fault methods from regular methods is to use a specific prefix F for fault methods.

Figure 7: The effect of fault-type f on a class c_i.

UML models in the presence of faults. An UML model M_f in the presence of fault f is the resulting model after modeling the effect of f on all state diagrams of the model M. The modeling of fault-type f at the state diagram level also affects existing sequence diagrams of M. Specifically, a state diagram S_{d_i} that is affected by faults will correspondingly influence all sequence diagrams where S_{d_i} is involved. The new sequence diagrams in M_f represent requirements scenarios in the presence of f. Hereafter, we use such scenarios in modeling fault-tolerance in M_f.

Example: Modeling fault-type f_1 in the object model of the ACC system. In the ACC system, we demonstrate how we model fault-type f_1 that perturbs the state of the car to the accelerating state. For reasons of space, we omit the modeling of fault-type f_2 introduced in Section 3.1. First, we present an excerpted class diagram of the ACC system in Figure 8. The main classes in the ACC system are (i) the Control class that models all the controlling activities of the ACC system (hereafter we interchangeably use the ACC system and the control), (ii) the Car class that models the attributes and the functionalities of the car, and (iii) the Radar class that models the attributes and the functionalities of the radar. Each instance of the car associates with one control object and one radar object (see Figure 8).

We model fault f_1 both at the structural and the behavioral levels. Specifically, in order to model fault f_1 at the class diagram level, we add a fault method $F\_f_1()$ to the Car class (similar to Figure 7). At the state diagram level, we add a fault transition that perturbs the state of the car from the state of calculating the real speed of the car to the Accelerating state (see Figure 9). To demonstrate the effect of the fault f_1 on sequence diagrams, we illustrate the coasting scenario in the presence of f_1 in Figure 10. In the closing mode, the control continuously asks for the distance between

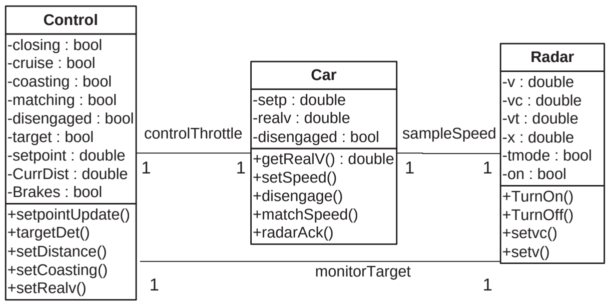


Figure 8: The class diagram of the ACC system.

the car and the target vehicle (see Figure 10). If the distance is less than or equal to the coasting distance then the control object changes its mode to the *coasting mode*. Subsequently, control asks the car to match its speed with the target vehicle. The fault-type f_1 may perturb the state of the car to the Accelerating state, where the car may increase its speed in the *coasting* mode and may violate the safety requirement.

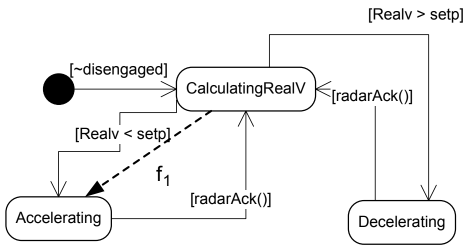


Figure 9: Modeling faults in the state transition diagram of the car.

5 Detector Pattern

In this section, we introduce the detector analysis pattern that is used in modeling failsafe and masking fault-tolerance. The detector pattern separates the concern of detecting a condition from the rest of the analysis model. We first present a formal definition of the detector pattern in OCM. Then we present the detector pattern template and explain how to apply it to UML models.

5.1 Formal Specification of the Detector Pattern

In this section, we extend the definition of detectors from Arora and Kulkarni [14,15] in the context of the OCM. We use detectors for modeling detection in the presence of faults. Such detection is necessary in specifying failsafe and masking fault-tolerant programs.

Detector. A detector is an OCM $\mathcal{D}$ comprising (i) a set of objects $d_1, \cdots, d_n$ $(n \geq 1)$, (ii) a relation $\mathcal{T}$ that represents the topology of the interconnection of $d_1, \cdots, d_n$, and (iii) a set $\{X, Z, I_\mathcal{D}\}$, where X and Z are state predicates, and $I_\mathcal{D}$ is an invariant of $\mathcal{D}$. The detector $\mathcal{D}$ validates the requirement $'Z$ detects X' from $I_\mathcal{D}$ if the following conditions are satisfied in all computations $\langle s_0, s_1, \cdots \rangle$ of $\mathcal{D}$ starting from $I_\mathcal{D}$ (from [14,15]):

- **Safeness.** If Z is true in a state s_j $(0 \leq j)$ then X must be true in s_j as well; i.e., $Z \Rightarrow X$.

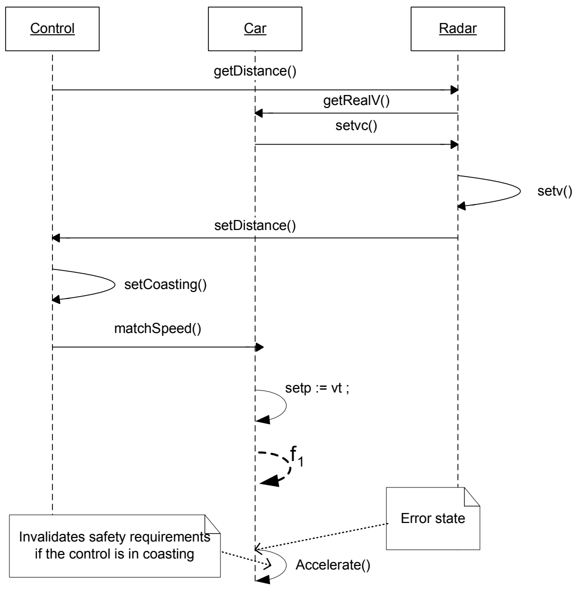


Figure 10: Coasting scenario in the presence of faults.

- **Progress.** If X is true in s_j then Z will eventually become true in a state s_k, where $j \leq k$.
- **Stability.** If Z is true in s_j then Z remains true in subsequent states as long as X is true. □

Intuitively, the safeness guarantees that the state predicate Z is never *true* when X is *false*; i.e., $\mathcal{D}$ never lies. The progress property ensures that if X is true then Z will eventually hold. In other words, the state predicate Z will eventually witness that the detection has occurred. Also, $\mathcal{D}$ should guarantee that once Z becomes *true*, it will remain *true* (i.e., Z remains *stable*) as long as predicate X is *true*. Note that safeness and stability can be classified as *safety* requirements and progress as *liveness* requirement of $\mathcal{D}$.

5.2 Detector Pattern Template

In this section, we present the template of the detector pattern. Depending on the problem at hand, each field of the detector pattern template should be instantiated. We use the ACC system to demonstrate such an instantiation, where we use the detector pattern to add failsafe f_1-tolerance to the ACC system.

Detection Problem. In order to guarantee safety in the presence of faults, a fault-tolerant system should be able to detect whether or not it is in an error state. An error state is a state from where the (safety and liveness) requirements may be violated by fault or system actions. A failsafe (respectively, masking) fault-tolerance should be aware of two classes of error states: (1) bad states from where a sequence of fault transitions *alone* may violate the safety requirements, and (ii) at-risk states from where the actions of the system itself *may* result in violating safety requirements. To detect error states, existing fault-tolerance techniques rely on a variety of error detection mechanisms such as error detection codes, watchdogs, snapshot procedures [44], acceptance tests [1], and exception conditions. Such mechanisms are different solutions for the recurring problem of detecting a condition. The detector pattern presents an abstract and generic method for formulating the detection problem. Of course, there may be other approaches for formulating the detection problem, but our approach provides a systematic way for (i) composing an instance of the detector pattern with the conceptual model of a system, and (ii) verifying the correctness of the resulting composition (see the Interference-Freedom Constraints field of the template).

Example: Adaptive cruise control. In order to preserve safety requirements in the presence of fault f_1, the ACC system should detect if it is in the coasting mode before accelerating the car. Thus, the detection problem amounts to detecting error states where the car is about to accelerate in the coasting mode.

Intent. The detector pattern (i) formulates the detection problem, and (ii) provides an abstract decomposition strategy for the detection problem.

Applicability. The detector pattern could be used for modeling conditional requirements wherever the truth-value of some condition should be evaluated before some actions take place.

Detection Predicate. A detection predicate, say X, is a state predicate whose truth-value should be examined. In other words, a detection predicate identifies a set of states that should be detected by a system. A detection predicate could be either a global state predicate or a local state predicate.

Example: Adaptive cruise control. An example detection predicate in the ACC system is the predicate $X_{ACC} \equiv$ ((car is in accelerating state) $\wedge$ (control is in coasting)). The predicate X_{ACC} is the same as the detection constraint identified in the use case analysis in Section 3.1.

Witness Predicate. A witness predicate, say Z, is a local state predicate whose truth-value of *true* implies that the detection predicate holds. If the detector pattern meets its requirements (specified in Section 5.1) then we say $'Z$ detects X'.

Detector Elements (Participants). In a distributed system, it is difficult for a component to detect a global detection predicate X in an atomic step [45]. Thus, we decompose the detection predicate X into a set of local detection predicates $X_1, \cdots, X_n$. Subsequently, we use detector elements d_i, $1 \leq i \leq n$, such that each d_i is responsible for detecting X_i. Each detector element d_i, for $1 \leq i \leq n$, is indeed a participant of the detector pattern and has its own detection predicate X_i and witness predicate Z_i, where $'Z_i$ detects X_i'. The detectors d_i, for $1 \leq i \leq n$, collaborate with each other in detecting X.

Example: Adaptive cruise control. We use the detector pattern for detecting the predicate X_{ACC} in the ACC system. The detector pattern is applied to the control and the car objects (see Section 4). Thus, in this case, the detector pattern comprises two elements $d_{control}$ and d_{car}. The element $d_{control}$ is composed with the control object to detect $X_{control} \equiv$ (control is in coasting). The element d_{car} is composed with the car object to detect $X_{car} \equiv$ (car is in accelerating state). Thus, the detection predicate X_{ACC} is equal to $X_{ocntrol} \wedge X_{car}$. The element $d_{control}$ (respectively, d_{car}) sets its witness predicate $Z_{control}$ (respectively, Z_{car}) to *true* when $X_{control}$ (respectively, X_{car}) holds.

Distinguished Element. In the collaboration of detector elements $d_1, \cdots, d_n$ towards detecting the truth-value of the detection predicate X, one element d_{index} is responsible for concluding the detection of X. The element d_{index} is called the *distinguished element.* The distinguished element d_{index} is the owner of the witness predicate Z and responsible for witnessing that the detection predicate X has become *true.*

Example: Adaptive cruise control. The distinguished element of the detector pattern applied to the

ACC system is the element d_{car}.

Structure. The topology of a distributed system determines how the detection of local state predicates associated with each object will result in the detection of the global predicate X. Such collaborative detection of X can be done either (i) sequentially, where each element d_i detects X_i after all its predecessors have witnessed their detection predicates, or (ii) in parallel, where all elements d_i, $1 \leq i \leq n$, detect their detection predicates concurrently. For example, if the underlying communication topology of a system is a ring (respectively, tree) then a sequential (respectively, parallel) detector pattern can be used. We respectively illustrate the structure of sequential and parallel detectors in Figures 11 and 12. (The shadowed objects represent the detector elements.) Note that, each object could be composed with more than one detector elements for the detection of different predicates.

The structure in Figure 11 provides a strategy for detecting a predicate $X \equiv (X_1 \wedge X_2 \wedge \cdots \wedge X_n)$ in a sequential fashion. Each detector element d_i is responsible for the detection of X_i $(1 \leq i \leq n)$. If X_i holds and all elements $d_1, \cdots, d_{i-1}$ have already witnessed their detection predicates $X_1, \cdots, X_{i-1}$ then the element d_i will eventually witness the truth-value of X_i (by setting Z_i to *true*). In other words, if Z_i holds then $X_1 \wedge \cdots \wedge X_{i-1} \wedge X_i$ must hold as well.

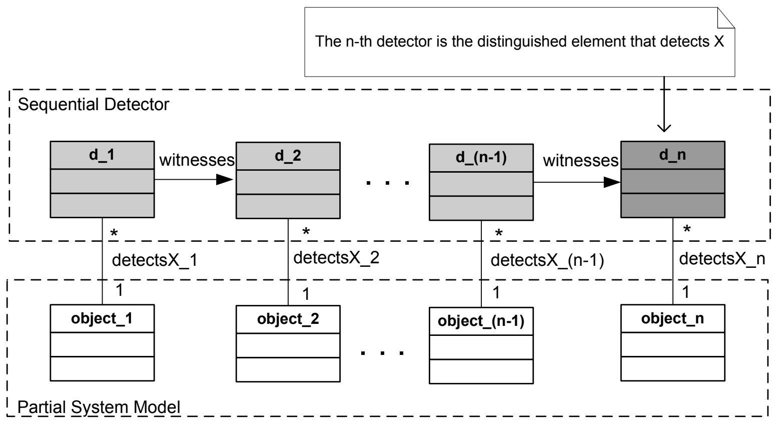


Figure 11: Sequential Detector.

The structure in Figure 12 provides a strategy for detecting a predicate $X \equiv (X_1 \wedge X_2 \wedge \cdots \wedge X_n)$ in a parallel fashion. Each detector element d_i is responsible for the detection of X_i $(1 \leq i \leq n)$. If X_i holds and all its children have already witnessed their detection predicates then the element d_i will eventually witness the truth-value of X_i (by setting Z_i to *true*). In other words, if Z holds then $Z_1 \wedge \cdots \wedge Z_n$ must hold as well.

Also, depending on the topology of the functional object model, a combination of sequential and parallel detectors may be used. For example, the structure in Figure 13 demonstrate a combination of sequential and parallel detectors in a hierarchical system. The detection of X_2 by itself requires a parallel detector. Thus, if d_1 has witnessed that X_1 holds and the parallel detector has also witnessed that X_2 holds then the element d_2 of the sequential detector will witness. Likewise, one can use a sequential detector for the detection of one of the elements in a parallel detector. In general, the composition of sequential and parallel detectors depends on the interconnection of the objects in the object model.

Example: Adaptive cruise control. We use a sequential detector in the case of the ACC system (see Figure 14). The element d_{car} witnesses if $d_{control}$ has already witnessed and X_{car} is *true*.

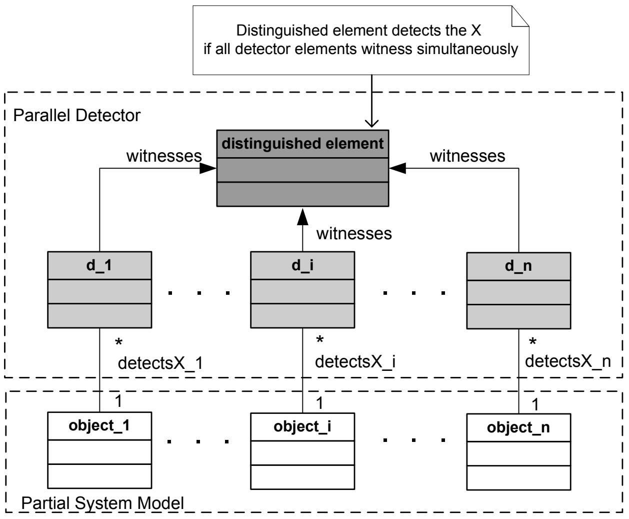


Figure 12: Parallel Detector.

Invariant. The invariant constraint for the sequential detector pattern states that if an element d_j, for $1 \leq j \leq n$, witnesses then all its predecessors $d_1, \cdots, d_{j-1}$ witness as well. Also, the invariant of the parallel detector pattern stipulates that if an element d_j witnesses then all its children witness as well.

Example: Adaptive cruise control. The invariant of the sequential detector applied to the object model of the ACC system is equal to $Z_{car} \Rightarrow Z_{control}$. Also, in this case, since we are using the sequential detector for modeling failsafe fault-tolerance, the sequential detector should be failsafe fault-tolerant against the effect of f_1 on the detector. Thus, if a fault-type F models the effect of f_1 on the detector then the detector should be failsafe F-tolerant from $Z_{car} \Rightarrow Z_{control}$. In other words, the detector should meet its safeness and stability requirements even in the presence of f_1. In this case, F may set the witness predicates Z_{car} and $Z_{control}$ to *false*, and the detector is failsafe F-tolerant.

Behavior. The behavior of the detector pattern (see Figures 15 and 16) illustrates a solution strategy for the detection problem. We represent the behavior of a sequential detector in Figure 15. To detect a global condition $X \equiv X_1 \wedge \cdots \wedge X_n$, the detector elements $d_1, \cdots, d_n$ detect their local detection predicates sequentially until the global detection predicate X is detected by the last element d_n (see Figure 15). More specifically, each element d_i is responsible for detecting its local detection predicate X_i after all its predecessors $d_1, \cdots, d_{i-1}$ have witnessed their local detection predicates ($2 \leq i \leq n$). As a result, when d_n witnesses, it follows that the detection predicate X holds.

The elements of the parallel detector, $d_1, \cdots, d_n$ detect their local detection predicates concurrently (see Figure 16). The global detection predicate is detected when all detector elements have witnessed their local detection predicates simultaneously (see Figure 16). More specifically, the distinguished element can witness when $Z_1 \wedge \cdots \wedge Z_n$ is true.

In the (sequential and parallel) detector pattern, each element d_i ($1 \leq i \leq n$) is concurrently composed with $object_i$ in the object model. In other words, the state diagram of the resulting composition will be a concurrent state comprising the state diagram of the detector element d_i and the state diagram of $object_i$. We respectively demonstrate the state diagrams of d_1 and d_i

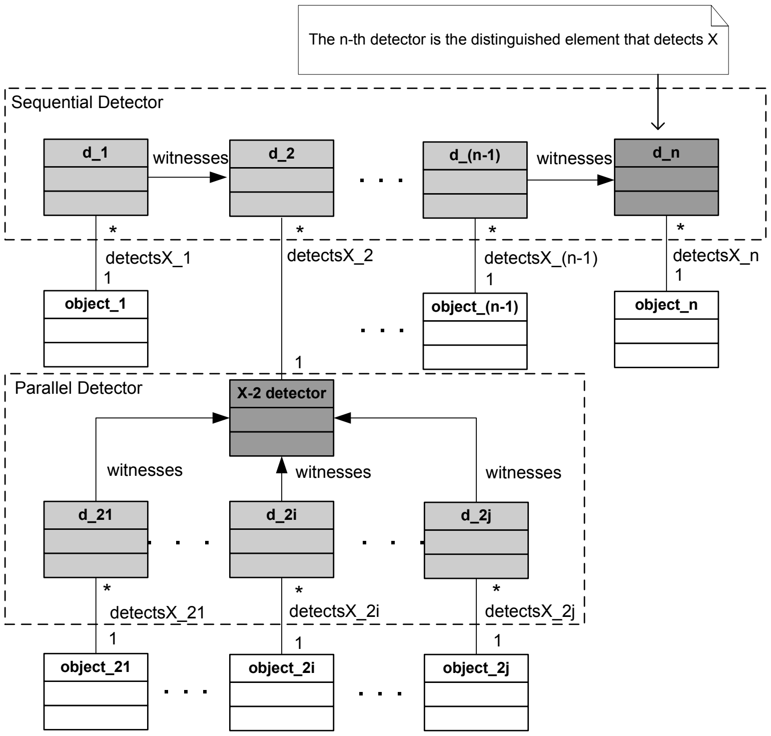


Figure 13: Composition of sequential and parallel detectors.

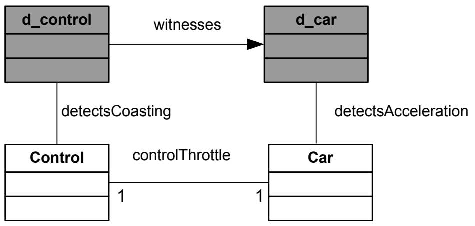


Figure 14: Composition of the sequential detector pattern with the ACC system.

$(1 < i \leq n)$ in a sequential detector in Figures 17 and 18. The element d_1 will witness if its local detection predicate X_1 holds (see Figure 17), whereas d_i $(1 < i \leq n)$ will witness if not only its local detection predicate X_i holds, but also its predecessor has already witnessed (see Figure 18), i.e., $Z_{(i-1)}$ is *true*. The detector element d_i, $1 \leq i \leq n$, stays in its *Witness* state as long as X_i is *true*, i.e., stability (see Figures 17 and 18). More specifically, d_i must not be in the *Witness* state if its detection predicate X_i has been falsified (by fault or system actions). The fault transitions may directly set the truth-value of Z_i to *false*. In such cases, the detector d_i transitions to the *CheckLocalDetectionPredicate* state and the safeness of the d_i is not violated. In cases where $object_i$ updates its variables in such a way that X_i becomes *false*, the value of Z_i should be set to *false* simultaneously. Otherwise, the safeness of d_i will be violated. Thus, $object_i$ should atomically set the value of Z_i to *false* whenever the actions of $object_i$ falsify X_i. In the case of detector element d_n, the value of Z_n should be atomically set to *false* if some X_i $(1 \leq i \leq n)$ is being falsified.

We depict the state diagram of the distinguished element (i.e., the root) of the parallel detector in Figure 19. In a parallel detector, the root witnesses its detection predicate if its local detection predicate X_{root} holds and all its children have already witnessed their detection predicate. Recursively, this rule applies to other nodes of the parallel detector; i.e., d_i will witness if its children

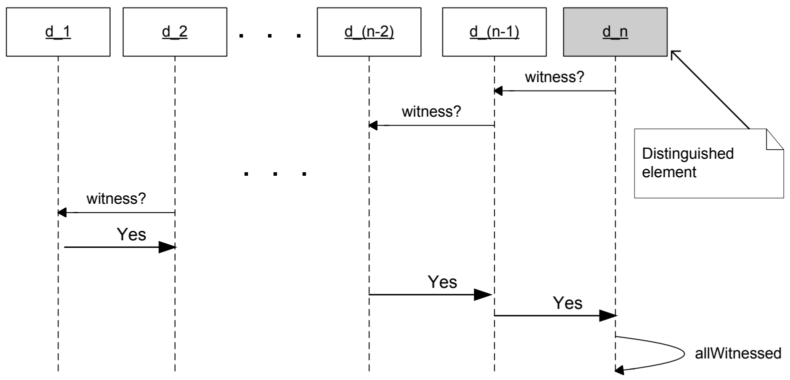


Figure 15: The behavior of a sequential detector.

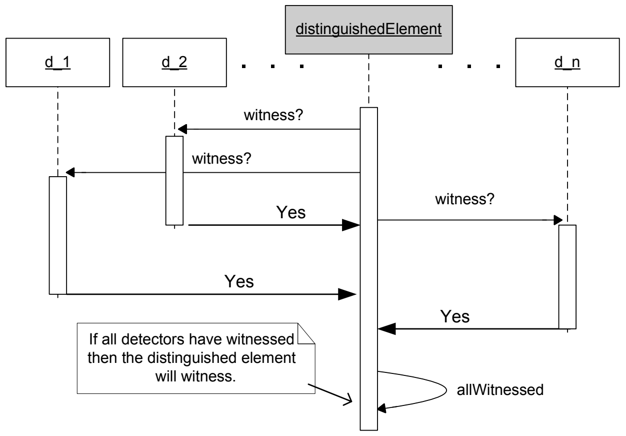


Figure 16: The behavior of a parallel detector.

have already witnessed and its local detection predicate X_i holds. Thus, the state diagram of each element d_i of the parallel detector is structurally similar to that of the root (see Figure 19).

Example: The state diagram of the detector pattern for the ACC system. We illustrate the state diagrams of the detector pattern used in the ACC example in Figures 20 and 21. In this section, we applied a sequential detector pattern to the object model of the ACC system to model failsafe f_1-tolerance. The detector element that is composed with the Car monitors the state of the Car object to check the truth-value of the state predicate X_{Car} (that represents whether or not the Car is accelerating). This detector element (see Figure 20) will witness X_{Car} if the detector element composed with the Control object (see Figure 21) witnesses its detection predicate $X_{Control}$ (that represents whether or not the ACC system is in the coasting mode).

We illustrate the behavior of the sequential detector pattern used in the ACC system in Figure 22. We demonstrate how the control (respectively, car) object interacts with the detector element $d_{control}$ (respectively, d_{car}). Also, Figure 22 illustrates the behavior of the elements $d_{control}$ and d_{car} in detecting the detection predicate X_{ACC}.

Collaborations. The client of the detector pattern is any component of the system model in which the truth-value of a condition should be checked. The client only checks the witness predicate Z by communicating with the distinguished element of the detector pattern.

Example: Adaptive cruise control. The client of the sequential detector added to the ACC system is the car object (see Figures 22 and 23). The car object checks whether the detection predicate

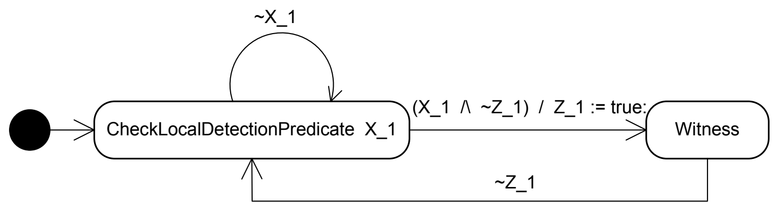

~X_1
CheckLocalDetectionPredicate X_1
(X_1 /\ ~Z_1) / Z_1 := true;
Witness
~Z_1

Figure 17: The state diagram of the element d_1 in the sequential detector pattern.

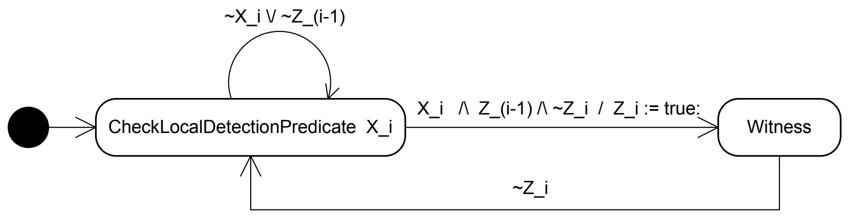

~X_i \/ ~Z_(i-1)
CheckLocalDetectionPredicate X_i
X_i /\ Z_(i-1) /\ ~Z_i / Z_i := true;
Witness
~Z_i

Figure 18: The state diagram of the element d_i of the sequential detector pattern.

X_{ACC} holds before accelerating the speed of the car. In order to illustrate this, we present the excerpted state diagram of the car object after applying the sequential pattern in Figure 23.

Interference-Freedom Constraints. The detector pattern should not *interfere* with the fault-intolerant object model. Intuitively, after the composition of the detector pattern with the fault-intolerant object model of a system, the resulting compositional conceptual model should meet (i) the functional requirements in the absence of faults, and (ii) the fault-tolerance requirements in the presence of faults. We say the detector pattern interferes with the object model iff in the absence of faults the compositional model violates (safety or liveness of) functional requirements. Also, we say the object model interferes with the detector pattern iff in the presence of faults the compositional model violates safeness, stability, or the progress of the detector pattern. More specifically, since we use detectors for modeling failsafe and masking fault-tolerance, the compositional model should meet the following requirements:

- *In the absence of faults*, the functional requirements must be met by the compositional conceptual model.
- *In the presence of faults*, depending on the level of fault-tolerance for which the detector pattern is applied, the following requirements must be met:
  - For failsafe fault-tolerance, the detector pattern should meet (i) safeness, where if the detector witnesses its detection predicate X then X must be *true*; i.e., the detector never lies, and (ii) stability, where if the detector witnesses then Z remains *true* as long as X is *true*.
  - For masking fault-tolerance, in addition to *safeness* and *stability*, the detector pattern must meet the progress property, where the detector guarantees to eventually witness if its detection predicate holds.

The safeness and stability are safety properties that can be specified in Linear Temporal Logic (LTL) [46] using (i) the universal operator $\Box$, where $\Box Y$ means that the state predicate Y always holds; (ii) the next state operator $Next$, where $Next(Y)$ means that in the next state Y holds, and (ii) the eventuality $\Diamond$, where $\Diamond Y$ means that the state predicate Y eventually holds.

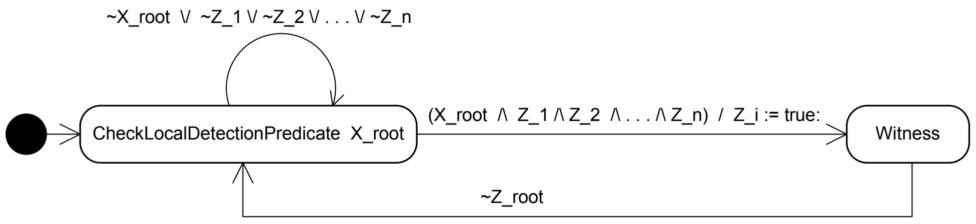

~X_root ∨ ~Z_1 ∨ ~Z_2 ∨ . . . ∨ ~Z_n
CheckLocalDetectionPredicate X_root
(X_root ∧ Z_1 ∧ Z_2 ∧ . . . ∧ Z_n) / Z_i := true:
Witness
~Z_root

Figure 19: The state diagram of the root of the parallel detector pattern.

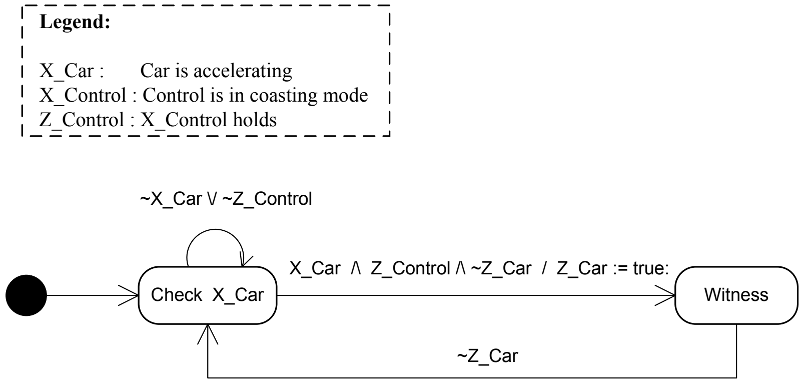

Legend:
X_Car : Car is accelerating
X_Control : Control is in coasting mode
Z_Control : X_Control holds
~X_Car ∨ ~Z_Control
Check X_Car
X_Car ∧ Z_Control ∧ ~Z_Car / Z_Car := true:
Witness
~Z_Car

Figure 20: The state diagram of the detector element composed with the Car object in the ACC system.

We respectively specify *safeness* and *stability* as $\Box(Z \Rightarrow X)$ and $\Box(Z \Rightarrow (Next(Z \vee \neg X)))$. Also, we specify *progress* as the following LTL expression: $\Box(X \Rightarrow \Diamond Z)$.

Example: Adaptive cruise control. The verification of the interference-freedom constraints can be done by the substitution of the predicates $X_{car}, Z_{car}, X_{control}$, and $Z_{control}$ in the above formulas. In Section 7, we illustrate how to generate formal specification of the detector pattern in Promela, where the SPIN model checker can be used for the verification of the interference-freedom constraints. To ensure the interference-freedom of the sequential detector and the ACC system, we also need to verify that the safety and liveness requirements of the ACC system (specified in Section 3) hold in the absence of faults.

6 Corrector Pattern

In this section, we introduce the corrector analysis pattern that is used in modeling nonmasking and masking fault-tolerance. The corrector pattern separates the concern of *correcting* a condition from the rest of the analysis model. We first present a formal definition of the corrector pattern. Then we present the template of the corrector pattern and explain how to apply the pattern to UML object models.

6.1 Formal Specification of the Corrector Pattern

In this section, we extend the formal definition of correctors from [14, 15] in the context of OCM. We use correctors for modeling recovery in the presence of faults. Such recovery is necessary in specifying nonmasking and masking fault-tolerant programs.

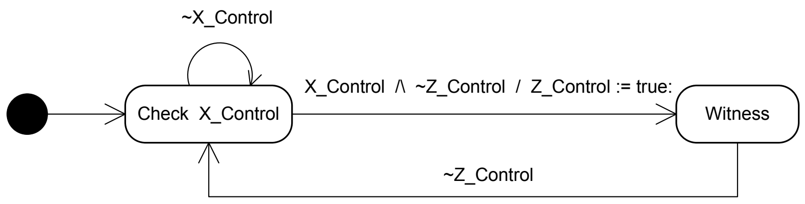


Figure 21: The state diagram of the detector element composed with the Control object in the ACC system.

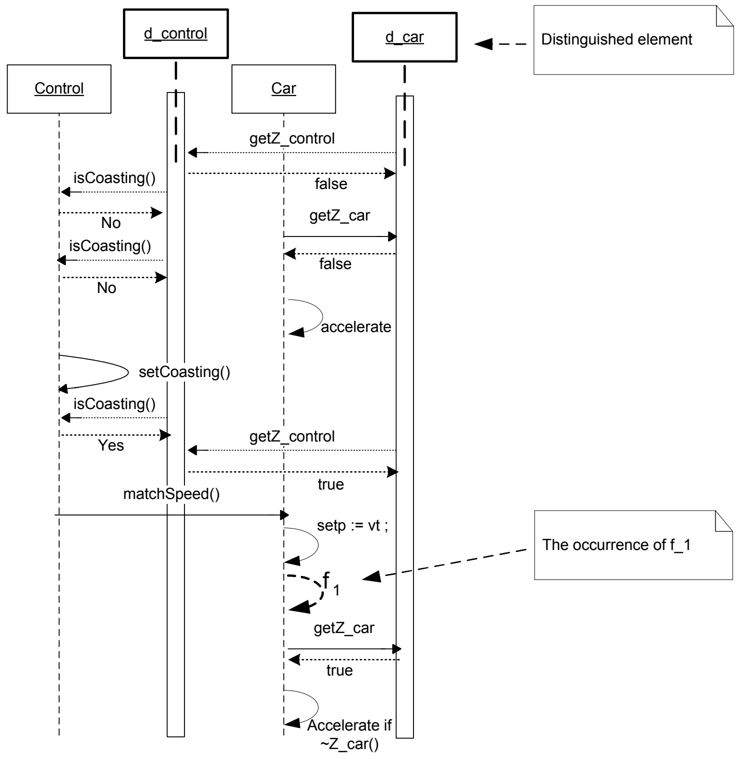


Figure 22: The behavior of the sequential detector applied to the ACC system.

Corrector. A corrector is an OCM $\mathcal{C}$ comprising (i) a set of objects $c_1, \cdots, c_n$ $(n \geq 1)$, (ii) a relation $\mathcal{T}$ that represents the topology of the interconnection of $c_1, \cdots, c_n$, and (iii) a set $\{X, Z, I_\mathcal{C}\}$, where X and Z are state predicates, and $I_\mathcal{C}$ is an invariant of $\mathcal{C}$. The corrector $\mathcal{C}$ validates the requirement $'Z$ corrects X' from $I_\mathcal{C}$ if the following conditions are satisfied in all computations $\langle s_0, s_1, \cdots \rangle$ of $\mathcal{C}$ starting from $I_\mathcal{C}$ (from [14, 15]):

- **Convergence.** There exists i, $i \geq 0$, such that for all j, $j \geq i$, X holds at s_j. Also, for all k, $k \geq 0$, if X holds at s_k, then X also holds at s_{k+1}.
- **Safeness.** If Z is true in a state s_j, where $j \geq 0$, then X must be true in s_j as well; i.e., $Z \Rightarrow X$; i.e., Z never witnesses incorrectly.
- **Progress.** If X is true in s_j then Z will eventually become true in a state s_k, where $j \leq k$; i.e., if X is true then Z will eventually hold.
- **Stability.** If Z is true in s_j then Z remains true in subsequent states as long as X is true.

□

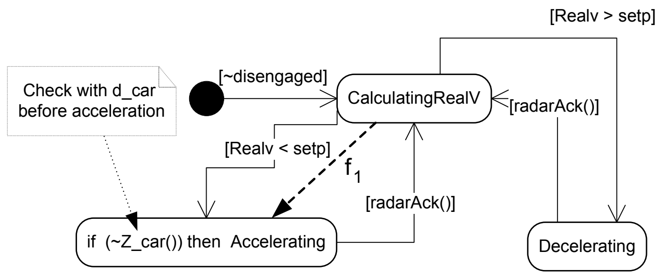


Figure 23: The excerpted state diagram of the car after applying the sequential detector pattern.

Intuitively, the convergence property guarantees that if X is *false*, then it will eventually become *true* and will stay *true* thereafter. In other words, a corrector $\mathcal{C}$ guarantees to correct the truth-value of the predicate X whenever X becomes *false*. Also, the state predicate Z witnesses that such correction has happened. Therefore, if Z is *true*, then X must have been corrected. If X has become *true*, then Z will eventually become *true*. Note that, if $'Z$ corrects X', then $'Z$ detects X' as well because $\mathcal{C}$ meets safeness, progress, and stability.

6.2 Corrector Pattern Template

In this section, we present the template of the corrector pattern. Also, in the context of the ACC example, we illustrate how correctors can be used for providing nonmasking f_2-tolerance.

Correction Problem. In order to provide a desired behavior in the presence of faults, a fault-tolerant system should guarantee some post conditions after it finds itself in an error state. The satisfaction of such post conditions prevents the occurrence of behavioral failures that may originate at error states. Extant fault-tolerance techniques such as error correction codes, recovery blocks [1], replicated state machines [4], and roll-back (respectively, roll-forward) recovery [6] are different mechanisms for correcting the state of computing systems. The corrector pattern generalizes the recurring problem of correcting the state of a computing system after the detection of error states.

Example: Adaptive cruise control. In the ACC system, we need to correct the invariant whenever it is falsified due to the occurrence of f_2. Specifically, the occurrence of fault-type f_2 (introduced in Section 3.1) may perturb the ACC system to a state s where the condition $Brakes(s) \Rightarrow disengaged(s)$ does not hold; i.e., the driver has applied the brakes, but the car has not disengaged from the cruise mode. The falsification of $Brakes(s) \Rightarrow disengaged(s)$ results in the falsification of the invariant I_{ACC}.

Intent. The corrector pattern (i) formulates instances of the correction problem, and (ii) provides an abstract decomposition strategy for the correction problem.

Applicability. The corrector pattern can be used in cases where one needs to ensure that the state of a computing system will eventually satisfy a particular condition. Thus, in the presence of faults, if the state of a computing system is perturbed outside its invariant then the corrector pattern can be used to ensure the recovery of the system to its invariant.

Correction Predicate. A correction predicate X represents the condition whose truth-value should be set to *true* once it becomes *false*. In a distributed system, it may not be possible to correct the truth-value of a state predicate in an atomic step. Thus, the correction of a *global* state predicate should be decomposed to a sequence of local corrective actions. In other words, the problem of correcting a global state predicate will be decomposed to the correction of a set of local predicates.

Example: Adaptive cruise control. The occurrence of f_2 may take the ACC system to a state s, where the car is not disengaged even though brakes are applied. We represent this situation by the correction predicate $Y \Rightarrow W$, where $Y \equiv Brakes(s)$ and $W \equiv disengaged(s)$. To provide nonmasking f_2-tolerance, we must ensure that the constraint $Y \Rightarrow W$ will eventually hold after f_2 stops occurring.

Witness Predicate. A witness predicate Z is a local state predicate whose truth-value of *true* implies that the correction predicate X holds.

Corrector Elements (Participants). Likewise the detector pattern, the corrector pattern comprises a set of *corrector elements* (denoted c_i, for $1 \leq i \leq n$) corresponding to each local correction predicate X_i. Each corrector element c_i is indeed a participant of the corrector pattern and has its own correction predicate X_i and witness predicate Z_i. The collaboration of corrector elements $c_1, \cdots, c_n$ results in the correction of the predicate X.

Distinguished Element. In the collaboration of corrector elements $c_1, \cdots, c_n$ towards correcting the truth-value of the corrector predicate X, one element c_{index} is responsible for concluding the correction of X. This element is called the *distinguished element*.

Structure. Likewise the detector pattern, the corrector pattern also provides two basic structures for modeling the collaborative correction of a state predicate X. Such collaborative correction of X can be done either (i) sequentially, where each element c_i corrects X_i after all its predecessors have witnessed their correction predicates, or (ii) in parallel, where all elements c_i, $1 \leq i \leq n$, correct their correction predicates concurrently. We omit the pictorial presentation of the structure of the corrector pattern since it is similar to the structure of the detector pattern. Instead, we illustrate the structure of a specific subclass of correctors composed of a (sequential/parallel) detector and a local corrector. (We apply an instance of this specific class of the corrector pattern to the ACC system.) In particular, in Figure 24, we demonstrate the structure of a corrector that comprises a parallel detector and a local corrector. First, the detector $\mathcal{D}$ detects $\neg X$ to determine whether the correction predicate X has become *false*. Then a corrective action, which is located at the distinguished element of $\mathcal{D}$, changes the value of X to *true* by updating the local state of a specific object O_i $(1 \leq i \leq n)$. Alternatively, such composition may take place by first performing the corrective action and then detecting if X holds.

Example: Adaptive cruise control. In the case of the ACC system, we use a corrector that comprises a sequential detector pattern and a local corrector (see Figure 25). The detector pattern comprises two elements d_1 and d_2 that are respectively composed with the control and the car objects. The sequential detector pattern is responsible for detecting that $Y \Rightarrow W$ has become *false*. Thus, the detection predicate of the detector pattern is equal to $Y \wedge \neg W$; i.e., brakes are applied, but the car is not disengaged yet. The local corrector is responsible for setting the disengaged flag in the car object if the sequential detector has witnessed that the predicate $Y \Rightarrow W$ does not hold.

Invariant. The invariant constraint for the sequential corrector pattern states that if an element c_j $(1 \leq j \leq n)$, witnesses then all its predecessors $c_1, \cdots, c_{j-1}$ witness as well. Also, the invariant of the parallel corrector pattern stipulates that if an element c_j witnesses then all its children witness as well. In case of the corrector in Figure 24, the invariant of the corrector pattern is equal to $(Z_{\mathcal{D}} \Rightarrow \neg X) \wedge (Z_{\mathcal{C}} \Rightarrow X)$, where $Z_{\mathcal{D}}$ is the witness predicate of the parallel detector and $Z_{\mathcal{C}}$ is the

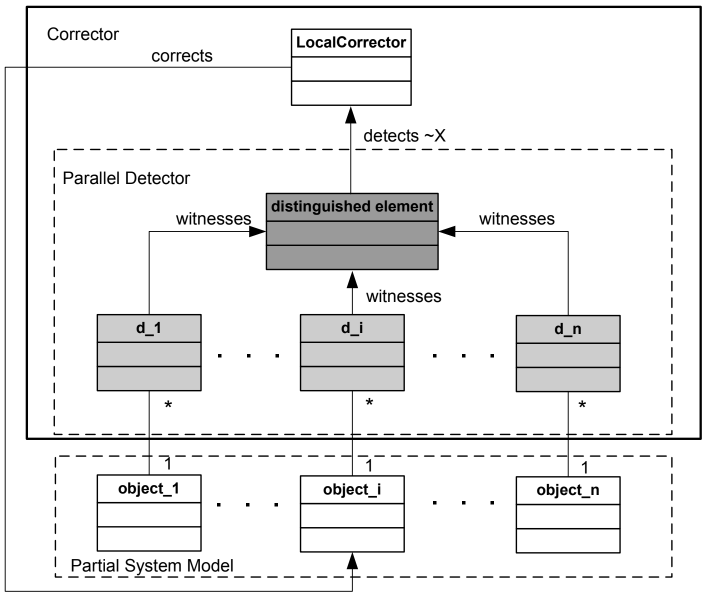


Figure 24: The structure of a detector composed with a local corrector.

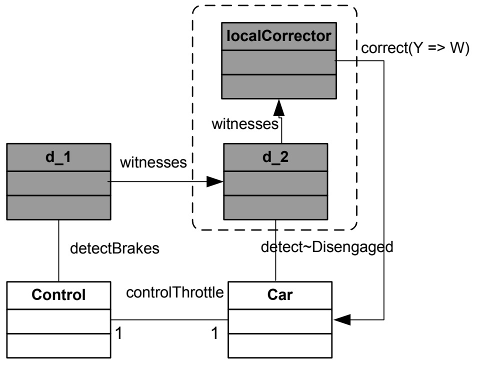


Figure 25: The structure of the corrector composed with the object model of the ACC system.

witness predicate of the local corrector.

Example: Adaptive cruise control. The invariant of the corrector used to add nonmasking f_2-tolerance to the ACC system is equal to $(Z_{d_2} \Rightarrow (Y \wedge \neg W)) \ \wedge \ (Z_{\mathcal{C}} \Rightarrow (Y \Rightarrow W))$, where $Z_{\mathcal{C}}$ is the witness predicate of the local corrector.

Behavior. In Figure 26, we illustrate the behavior of the corrector pattern of Figure 24. First, the parallel detector detects that the state predicate X does not hold. Then the local corrector updates the state of some object $object_i$, which in turn results in the correction of X.

Example: Adaptive cruise control. We illustrate the behavior of the corrector pattern composed with the ACC object model in Figure 27. This Figure depicts (i) the collaborations between the sequential detector pattern and the local corrector, and (ii) the collaborations between the elements of the detector pattern and the car and control objects.

Collaborations. The client of the corrector pattern is any component of the system object model in which the correction predicate X is set to *true* after it becomes *false*. The client only checks the witness predicate Z to verify whether or not X holds.

Interference-Freedom Constraints. To ensure that there is no conflict between fault-tolerance

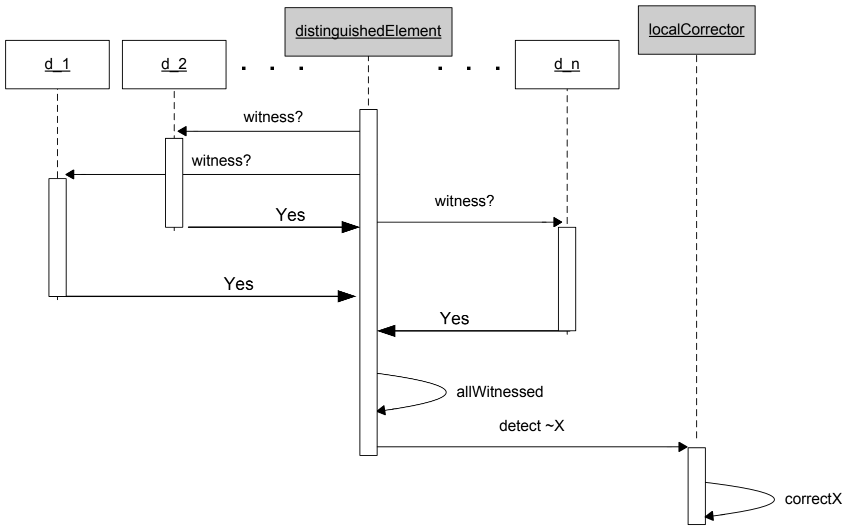


Figure 26: The behavior of the corrector of Figure 24.

requirements and functional requirements of a system, we need to provide constraints for verifying that the corrector pattern and the system object model do not interfere. Towards this end, we present the following constraints for the conceptual model resulting from the composition of the corrector pattern and the system object model:

- *In the absence of faults*, the functional requirements must be met by the compositional conceptual model.
- *In the presence of faults*, depending on the level of fault-tolerance for which the corrector pattern is applied, the following requirements must be met:
  - For nonmasking fault-tolerance, the corrector pattern should meet (i) convergence, where X will eventually become *true* and will continuously remain *true* thereafter, and (ii) progress, where the witness predicate Z will eventually become *true* if X becomes *true.*
  - For masking fault-tolerance, in addition to *convergence* and *progress*, the corrector pattern must meet the safeness and stability properties.

  The LTL expression $\Diamond(\Box X)$ represents the convergence constraint of the corrector pattern; i.e., X will eventually become *true* and will remain *true.* (See the detector pattern for LTL specification of *progress, safeness*, and *stability*).

The verification of the interference-freedom constraints ensures that the composition of the corrector pattern with the fault-intolerant object model meets both the functional and fault-tolerance. Also, it guarantees that the corrector pattern itself is nonmasking (respectively, masking) fault-tolerance if it is used for the addition of nonmasking (respectively, masking) fault-tolerance.

Example: Adaptive cruise control. In order to ensure that the conceptual model of the ACC system captures the requirements of nonmasking f_2-tolerance, we verify the interference-freedom constraints. Such a verification can be achieved by (i) generating the formal specification of the compositional conceptual model; (ii) substituting the appropriate state predicates in the above LTL formulas, and (iii) using a model checker to verify the interference-freedom constraints. In the next section, we concentrate on how to carry out such verification.

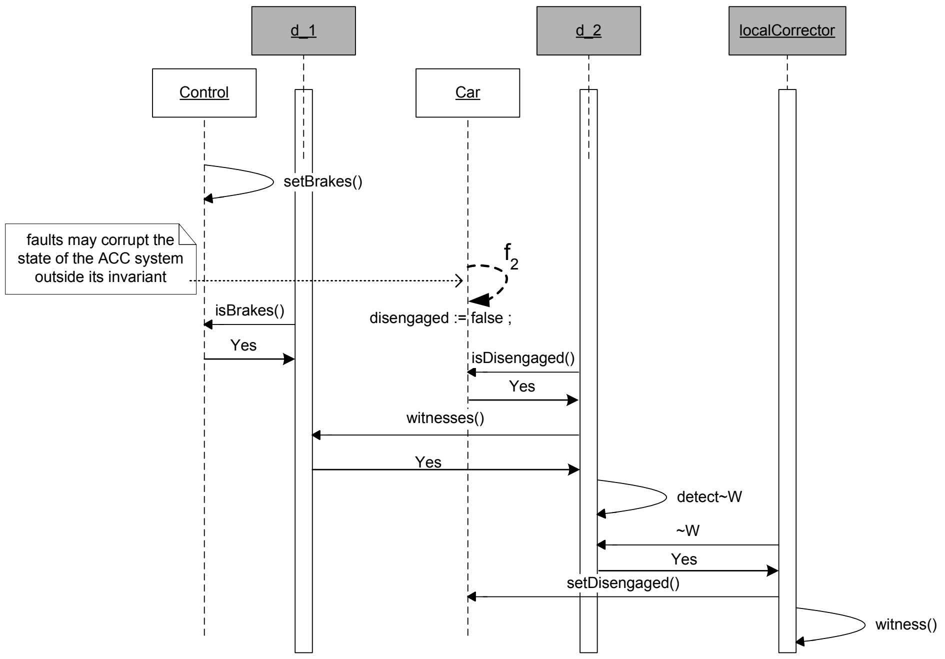


Figure 27: The behavior of the corrector pattern composed with the ACC system.

7 Formal Analysis of Fault-Tolerant Systems

In this section, we demonstrate how to generate formal specification of the detector and corrector patterns in Promela modeling language [16]. Also, we present a method for formal specification of faults in Promela. The formalization presented in this section enables us to use the SPIN model checker [17] in analyzing the inconsistencies of fault-tolerance requirements with functional requirements (if any). More importantly, the detector and corrector patterns provide a level of modularity that helps us in (i) pinpointing the source of inconsistencies, and (ii) resolving the inconsistencies between functional and fault-tolerance requirements.

7.1 Formalizing Detector and Corrector Patterns

In this section, we present a formal template for the detector pattern in Promela. We omit the presentation of the template of the corrector pattern as it is similar to the detector pattern. We use Hydra formalization framework [9,10] to generate the template. Also, we use Hydra to generate the Promela specification of the functional object model that is concurrently run with the detector and corrector patterns. Before presenting the Promela code of the detector pattern, we briefly review Promela and Hydra.

The Promela modeling language. Promela is a language for modeling concurrent and distributed programs in the model checker SPIN [17]. Using SPIN, developers can verify the model of a computing system (specified in Promela) with respect to a set of desired properties specified in Linear Temporal Logic (LTL) [46]. Also, SPIN enables simulation of the system model for validation purposes. The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called `proctype`, and (3) a set of asynchronous and synchronous channels for inter-process communications. Semantically, Promela is a descendent of Dijkstra's guarded commands [47] and CSP [48] except that there exist no mechanisms for inter-process communications in guarded commands language. Also, Promela is more general than CSP in that Promela allows the use of buffered channels, whereas in CSP processes communicate only via rendezvous channels; i.e., synchronized unbuffered channels.

The Hydra formalization framework. Hydra [9, 10] is a generic framework for generating formal specification from textual representation of UML (structural/behavioral) diagrams. Specifically, Hydra uses a set of mapping rules for translating entities in UML meta-model to entities in the meta-model of a target specification language (e.g., Promela). For example, Hydra translates two concurrent states specified in UML state diagrams to two concurrent proctypes specified in Promela. Using Hydra, we translate UML models to Promela specification language, which in turn enables us to verify (respectively, validate) UML models against properties of interest.

Generating Promela code for the detector pattern. Hydra generates the following proctype for an element d_i in the sequential detector pattern, which is concurrently run with the proctype of the composed object $object_i$ (see Figures 11 and 12). The structure of the Promela code generated for the parallel detector pattern is also similar to the following template.

```
proctype  d_i() {
do
  :: atomic{ X_i && Z_(i-1) && !Z_i -> Z_i = true; }
od;
}

proctype  object_i() {
 /* The actions of object_i */
  }
```

The state predicates X_i, Z_i, and $Z_(i-1)$ in the above Promela code are defined in terms of the variables in the locality of $object_i$, which is composed with d_i. Thus, although the above template could be used for different object models, the state predicates should be instantiated based on the state variables of $object_i$ in the object model. Also, the communication between different detector elements takes place using their shared variables. For example, in the above code, the element d_i reads the witness predicate of the element d_{i-1}, i.e., Z_{i-1}.

Example: The Promela code for the detector pattern of the ACC system. The formalization of the detector pattern for the ACC system results in the following Promela code, which is an instantiation of the template introduced above.

```
proctype  d_Car() {
do
  :: atomic{ (CarState == Accelerating) && Z_Control && !Z_Car -> Z_Car = true; }
od;
}

proctype  Car() {

 /* The actions of the  Car object.  */

  }
```

The X_{car} has been substituted with the predicate $(CarState == Accelerating)$, where $CarState$ is a variable that represents the current state of the car object. Since the predecessor of the detector element d_{car} (see Figure 14) is $d_{control}$, the predicate Z_{i-1} has been substituted with $Z_{control}$.

7.2 Formalizing the Fault Model

In this section, we illustrate how to model faults in Promela. Also, we discuss issues related to executing the fault model in SPIN. As mentioned in Section 4, we model a fault-type as a set of transitions in UML object models. In order to model faults in Promela, Hydra integrates all fault transitions in a Fault process that could be executed concurrently with all other processes in the executable model. For example, in the case of the ACC system, Hydra generates the following Promela code to represent all types of faults that perturb the cruise control system.

```
proctype  Fault() {
 if
  :: atomic{ Car. CarState == CalculatingRealV -> Car.CarState = Accelerating; }
  :: atomic{ Car.disengaged = true -> Car.disengaged = false; }
  :: atomic{ Control.closing = true -> Control.closing = false; }
  :: atomic{ Control.matching = true -> Control.matching = false; }
  :: atomic{ Control.coasting =true -> Control.coasting = false; }
 fi
}
```

Each action of the above Promela code represents a set of transitions in the entire state space of the object model. (For simplicity, we have attached the name of each variable to the name of the object that owns that variable.) Also, note that, in the Promela model, all actions are executed in an atomic step.

The execution of the Fault proctype must conform with the formal model of faults defined in OCM (see Section 2.5). More specifically, the Fault proctype should represent a process with terminating computations; i.e., the Fault process will terminate in a finite amount of time. This is a realization of the assumption made in Section 2.5 that faults will eventually stop occurring so that recovery can be provided by the corrector pattern.

The semantics of the execution of the Fault proctype in SPIN depends on the model of fairness selected while simulating (respectively, verifying) the fault-tolerant object model. Since, by default, SPIN does not guarantee fair execution of all processes [49], there exists a possibility that the Fault proctype will never get an execution chance during the simulation (respectively, verification) of the fault-tolerant model. Thus, during simulation (respectively, verification) of fault-tolerance requirements, the fairness option in SPIN should be set so that the Fault process is executed along other processes. Since the Fault process will eventually terminate, the fault-tolerant model should eventually meet its liveness requirements (after recovery to the invariant).

7.3 Analyzing the Inconsistencies of Fault-Tolerance and Functional Requirements

In this section, we present a strategy for analyzing fault-tolerance and functional requirements using the SPIN model checker [17]. Specifically, we specify the steps that must be taken in creating a consistent conceptual model for fault-tolerant systems (after the generation of the Promela code).

In this section, our focus is on the conceptual model of fault-tolerant systems that is the composition of the functional object model and the detector/corrector patterns.

In order to analyze the consistency of the fault-tolerance requirements with the functional requirements, we first exclude the Fault proctype from the Promela code of the object model. Then we verify (respectively, validate) the safety and liveness requirements. This step ensures that in the absence of faults the failsafe fault-tolerant model meets its functional requirements. To analyze the fault-tolerant object model in the presence of faults, we include the Fault proctype in the Promela code of the object model, and we select one of the following steps:

- To verify (respectively, validate) the failsafe fault-tolerance requirements, we ensure that the *safety* requirements hold.
- To verify (respectively, validate) the nonmasking fault-tolerance requirements, we ensure that the object model will eventually recover to its invariant.
- To verify (respectively, validate) the masking fault-tolerance requirements, we ensure that (i) the *safety* requirements hold, and (ii) the object model will eventually recover to its invariant.
- In addition to the above steps, we have to verify that (i) each instance of the detector/corrector pattern that is used for providing failsafe (respectively, nonmasking or masking) fault-tolerance is itself failsafe (respectively, nonmasking or masking) fault-tolerant. This can be achieved by verifying the interference-freedom constraints specified (in LTL) in Sections 5 and 6.

The counterexamples generated during the above verification steps represent the inconsistencies of fault-tolerance and functional requirements. Specifically, if SPIN generates counterexample during the verification of interference-freedom constraints then those counterexamples represent scenarios where fault-tolerance requirements are inconsistent with the functional requirements. Using such counterexamples, the analysts can revise the conceptual model of a fault-tolerant system at the early stages of development. For example, in the conceptual model of the ACC fault-tolerant system, if after d_{car} witnesses that its detection predicate $X_{car} \equiv$ (car is accelerating) holds (i.e., d_{car} sets the value of Z_{car} to *true*) the state of the car object is change to another state without falsifying Z_{car} then the safeness of d_{car} will be violated. To remedy this inconsistency, we revise the object model in that any transition originating at the Accelerating state should be accompanied with the simultaneous falsification of Z_{car}.

8 Analysis Method

In this section, we give an overview of our analysis method, where we (i) emphasize the challenges that exist in the development of fault-tolerant systems, and (ii) motivate the analysis method that we introduce in this paper. Also, we define what we mean by (detector/corrector) *patterns* in the analysis phase (since one may think of the term pattern as a design concept).

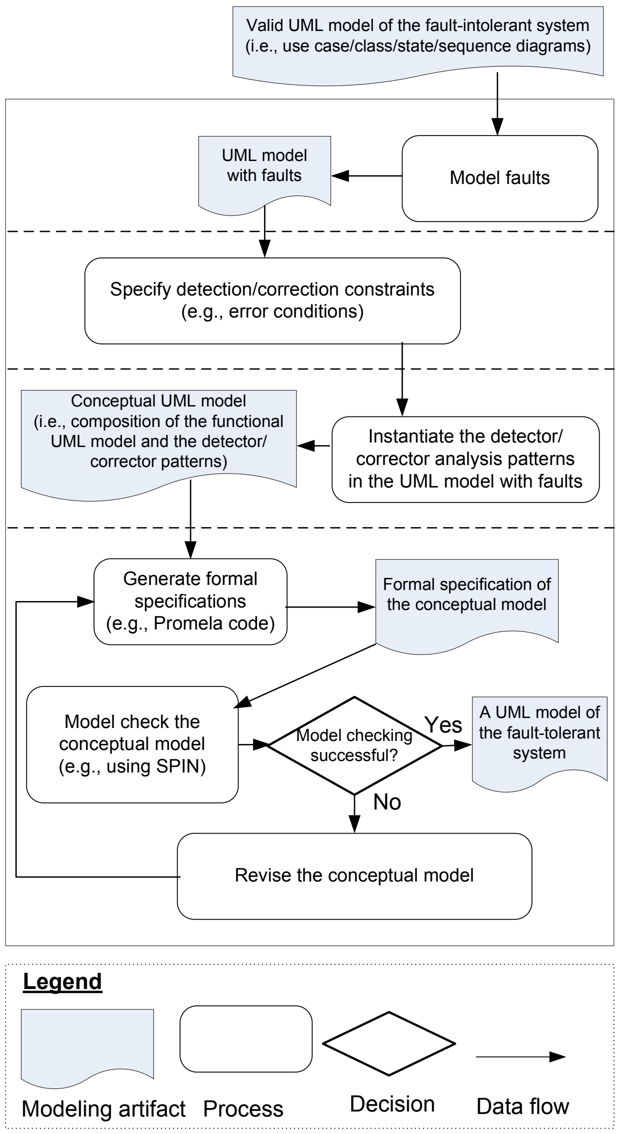
At the early stages of system development, it is difficult (if not impossible) to anticipate all types of faults that may perturb a system. Thus, as developers encounter new types of faults, they have to add desired level of fault-tolerance to an existing system. Clearly, such an addition of fault-tolerance should (i) preserve the functionalities of the existing system in the absence of faults, and

(ii) provide the required level of fault-tolerance in the presence of faults. Moreover, adding fault-tolerance against a newly discovered fault-type should preserve fault-tolerance properties that have already been added against other types of faults. Due to such complex mixture of functional and fault-tolerance requirements, we need systematic (and preferably automatic) methods for modeling and analyzing fault-tolerance requirements before entering design and implementation phases.

In order to provide a systematic approach for the development of fault-tolerant systems, we present an incremental modeling and analysis method that includes two major steps: *elicitation and specification of fault-tolerance requirements*, and *building conceptual models for automated analysis*. To specify fault-tolerance requirements, we start with identifying problems (i.e., constraints) that must be solved towards providing fault-tolerance. Based on necessity and sufficiency of detectors and correctors for designing fault-tolerance requirements [14,15], we focus on identifying detection and correction constraints. The use of detection/correction constraints (i) separates fault-tolerance and functional concerns; (ii) separates fault-tolerance concerns added for different types of faults, and (iii) provides traceability throughout the entire software development lifecycle.

To build the conceptual model of fault-tolerant systems, we present fault-tolerance analysis patterns for problem decomposition. In this paper, an analysis pattern is an abstract strategy for decomposing a problem into subproblems. Likewise Jackson [50], our focus is on identifying problems that should be solved towards providing fault-tolerance. Moreover, we move one step further in presenting techniques for decomposing a problem to smaller problems using our analysis patterns. Our notion of analysis patterns differs from Fowler's analysis patterns [51] in that (1) Fowler's analysis patterns represent recurring ideas for business modeling, whereas our fault-tolerance analysis patterns are recurring problems that must be solved for supporting fault-tolerance, and (2) Fowler's analysis patterns are the outcome of the modeler's experience in a particular domain, and as a result, they may be an incomplete set of patterns for the subject domain, whereas our fault-tolerance analysis patterns comprise a canonical set of patterns that are necessary and sufficient for specifying fault-tolerance.

We present an overview of our analysis method in Figure 28. When developers encounter a new fault-type that perturbs an existing system, the first step is to model faults and fault-tolerance requirements in the use case model of the existing fault-intolerant system (see Section 3). The specification of the fault-tolerance requirements comprises of specifying detection and correction constraints. Then, at the object analysis level, developers have to use detector and corrector patterns (see Sections 5 and 6) for the refinement of fault-tolerance constraints. The resulting artifact would be a conceptual model that contains the conceptual model of the fault-intolerant system composed with detector and corrector patterns. Finally, we generate formal specification from the fault-tolerant conceptual model for automated analysis. Specifically, developers have to verify the interference-freedom of fault-tolerance analysis patterns and the fault-intolerant model; i.e., the detector and corrector patterns and the fault-intolerant model meet their requirements in the presence of each other. If the fault-tolerance analysis patterns and the fault-intolerant model interfere then developers should modify the fault-tolerant conceptual model towards resolving the inconsistencies. Nonetheless, ensuring interference-freedom is out of the scope of this paper since our goal is to create the necessary framework for automated analysis of such inconsistencies between functional and fault-tolerance requirements.


Valid UML model of the fault-intolerant system (i.e., use case/class/state/sequence diagrams)
UML model with faults
Model faults
Specify detection/correction constraints (e.g., error conditions)
Conceptual UML model (i.e., composition of the functional UML model and the detector/corrector patterns)
Instantiate the detector/corrector analysis patterns in the UML model with faults
Generate formal specifications (e.g., Promela code)
Formal specification of the conceptual model
Model check the conceptual model (e.g., using SPIN)
Model checking successful?
Yes
A UML model of the fault-tolerant system
No
Revise the conceptual model
Legend
Modeling artifact
Process
Decision
Data flow
Figure 28: An overview of the analysis method.

9 Diesel Filter System

In this section, we illustrate how to use an instance of the detector pattern to model failsafe fault-tolerance in the object model of a Diesel Filter System (DFS) introduced in [52]. In the context of this case study, we are interested in modeling and analyzing failsafe fault-tolerance using the detector pattern. Thus, we focus on the aspects of the DFS system related to our analysis. The DFS is an embedded computing system used to reduce soot from diesel truck exhaust (see Figure 29). The DFS system comprises a canister containing a filter placed into the exhaust pipe, where the exhaust gas flows through the filter. A filter has several tubes consisting of ceramic fibers wrapped around a metallic cylindrical grid. Due to the exhaust gas flow, the filter becomes clog up after a while and it needs to be cleaned. The cleaning is achieved by heating up the grid (using heaters mounted around the metallic grid) and burning off the trapped soot. The DFS controller initiates the cleaning cycle whenever the differential pressure across the canister is beyond a pre-determined threshold. A pressure sensor provides the current differential pressure to the DFS controller. We illustrate an excerpted class diagram of the DFS system in Figure 30.

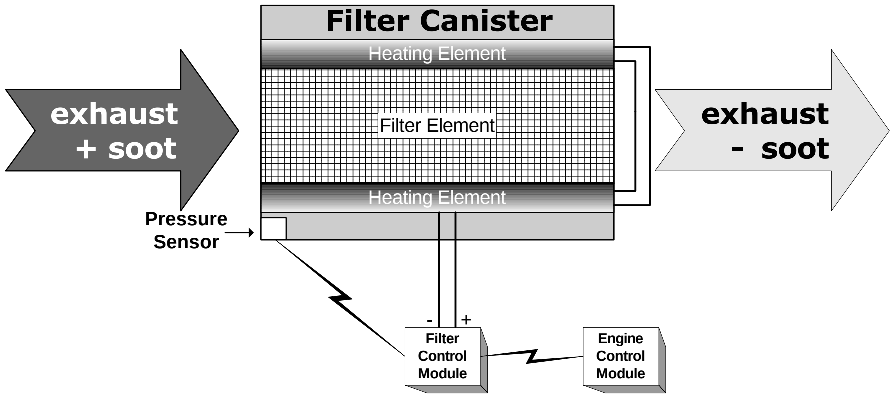


Figure 29: Physical structure of a diesel filter system.

- The PressureSensor class models the pressure sensor, which only provides the differential pressure across the filter canister to the FilterController class (i.e., the pressure sensor is a passive element and does not processing).

- The FilterController class models the controlling functionalities of the DFS. The FilterController class has three operating modes. The DFS is in the *idle* mode if the current pressure measured by the pressure sensor is between 0 and 8000 *Pascals* (Pa). In the idle mode, the FilterController remains inactive. If the pressure is above 8000 Pa and less than or equal to 10000 Pa, then the FilterController enters the *cleaning* mode, where it starts a cleaning cycle by electrifying the heaters. The FilterController enters the *safety* mode if the sensed pressure is above 10000 Pa, where the FilterController must not initiate the heating process.

- The EngineController class models the engine control module that measures the current engine speed. The EngineController.totalRPM variable keeps the total number of engine revolutions since last cleaning cycle, and the EngineController.currRPM variable represents the current speed of the engine.

- The HeatRegulator class models a heater installed in the filter canister, and the HeatRegulator.cleanupValue represent a value set by the FilterController that determines how fast the heating should be done.

- The HeaterCurrentSensor class models the passive sensor used to measure the value of the electrical current in the heater regulator.

Safety requirements of the DFS system. The cleaning cycle should not be started if less than 10000 engine revolutions have occurred since the last cleaning cycle or if the current engine revolutions per minute (RPM) is less than 700. (The engine control module (see Figure 29) measures the current RPM.) Moreover, the heating elements must not be defected (represented by a negative value of HeaterCurrentSensor.electricalCurrent) if the DFS controller is to activate the heaters.

We present the state diagram of the FilterController in Figure 31. The controller starts by checking the pressure of the filter canister. Depending on the value of the pressure received form the pressure sensor, the controller transitions to the Idle, Ready, or Safety states. In the Idle state, the controller

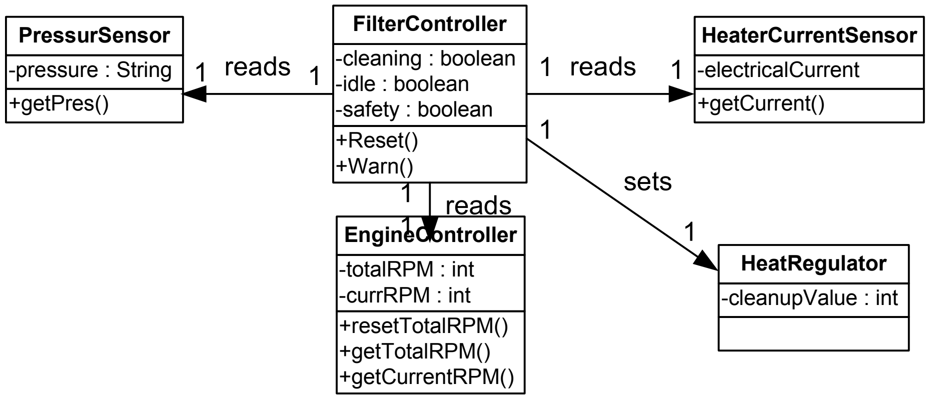


Figure 30: Excerpted class diagram of the diesel filter system.

again measures the pressure and moves to other states based on the pressure value. If the greater than 8000 and less than or equal to 10000, then the controller moves to the Ready state, where it checks other conditions before transitioning to the Cleaning state. Specifically, in the Ready state, the controller ensures that the total RPM is at least 10000, the current RPM is at least 700, and the heater is not defected. After the cleaning is over, the controller resets the value of the total RPM for the next cleaning cycle.

Faults. The DFS system is subject to faults that may perturb the state of the FilterController to the Cleaning state from any other state (see Figure 31). When such faults occur, the controller may issue command signals for the heater in an inappropriate time, which could lead to the violation of the safety requirements of the DFS. Next, we illustrate how to use an instance of the parallel detector pattern to guarantee safety even when faults occur.

Applying the Detector Pattern. We compose an instance of a parallel detector with the object model of the DFS (see Figure 32). The parallel detector (encapsulated in a dashed box in Figure 32) detects whether or not the preconditions of heating up hold in three different components of the DFS, namely, the PressureSensor, the EngineController, and the HeaterCurrentSensor. The distinguished element queries the detector elements and if all elements simultaneously witness their detection predicates then the distinguished element will also witness. We present a subset of the fields of the detector pattern that are related to modeling and analyzing failsafe fault-tolerance in the object model of the DFS.

Detection Predicate. Since faults may take the state of the FilterController to the cleaning state, we have to ensure that the FilterController issues command signals for the heater only if the condition X_{DFS} holds, where $X_{DFS} \equiv X_{pressure} \ \wedge \ X_{engine} \ \wedge \ X_{heater}$, and

$$\begin{array}{ll} X_{pressure} & \equiv ((8000 < PressureSensor.pressure \leq 10000) \\ X_{engine} & \equiv (EngineController.totalRPM \geq 10000) \ \wedge \ (EngineController.currRPM \geq 700) \\ X_{heater} & \equiv (HeaterCurrentSensor.electricalCurrent \geq 0)) \end{array}$$

Detector Participants. The parallel detector pattern comprises three elements $d_{pressure}$, d_{engine}, and d_{heater}, where $d_{pressure}$ (respectively, d_{engine} and d_{heater}) is responsible for detecting $X_{pressure}$ (respectively, are responsible for detecting X_{engine} and X_{heater}). Each element has its own witness predicate that implies its corresponding detection predicate (e.g., $Z_{pressure}$ implies $X_{pressure}$).

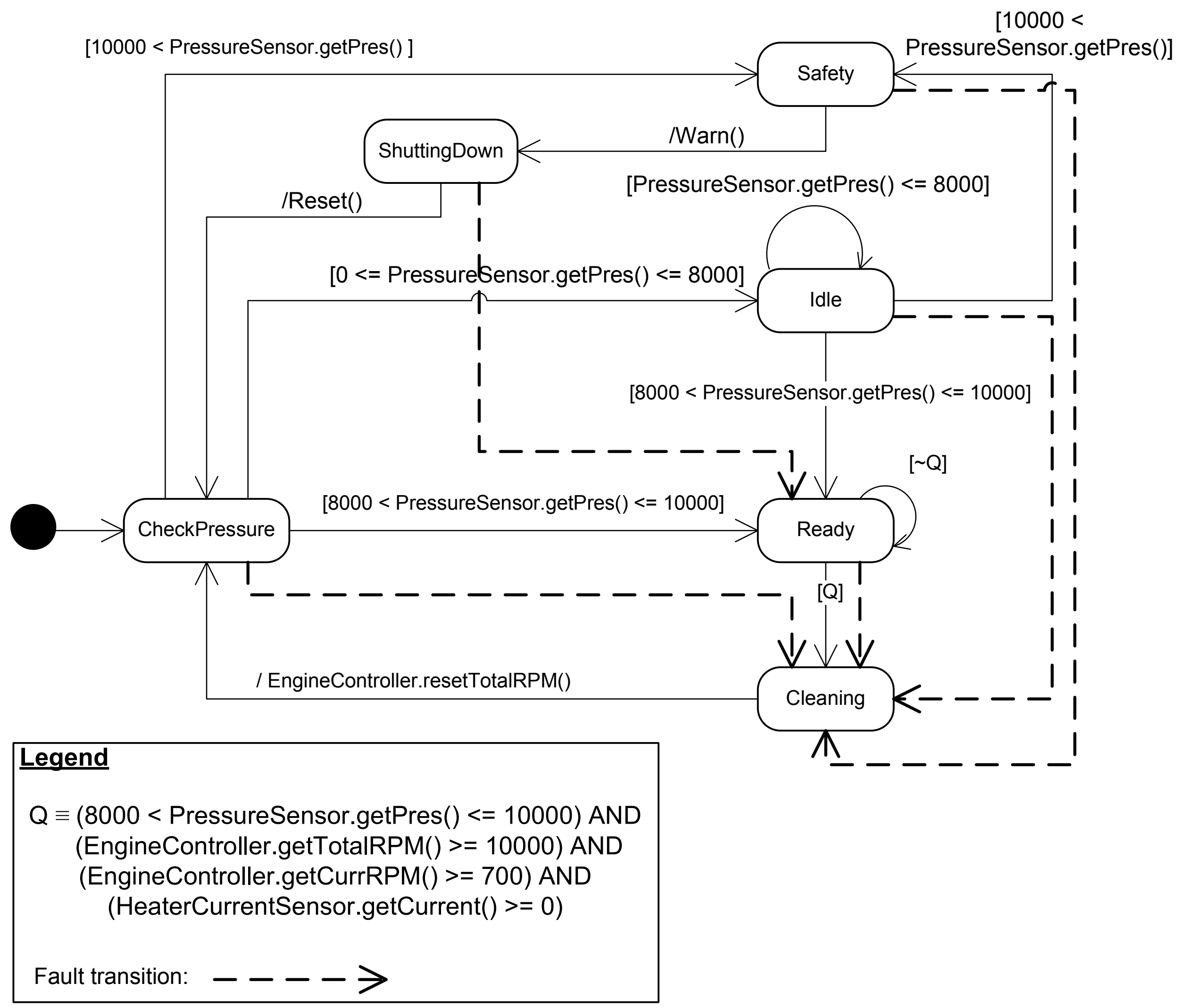

[10000 < PressureSensor.getPres()]
[10000 < PressureSensor.getPres()]
Safety
ShuttingDown
/Warn()
[PressureSensor.getPres() <= 8000]
/Reset()
[0 <= PressureSensor.getPres() <= 8000]
Idle
[8000 < PressureSensor.getPres() <= 10000]
[~Q]
CheckPressure
[8000 < PressureSensor.getPres() <= 10000]
Ready
[Q]
/ EngineController.resetTotalRPM()
Cleaning
Legend
Q ≡ (8000 < PressureSensor.getPres() <= 10000) AND (EngineController.getTotalRPM() >= 10000) AND (EngineController.getCurrRPM() >= 700) AND (HeaterCurrentSensor.getCurrent() >= 0)
Fault transition:

Figure 31: An abstract state diagram of the filter controller.

Distinguished element and the witness predicate. The distinguished element is an object that is the owner of the witness predicate Z_{DFS}, where Z_{DFS} implies that X_{DFS} holds and $Z_{DFS} \equiv (Z_{pressure} \wedge Z_{engine} \wedge Z_{heater})$.

Detection Requirements. Since we use the parallel detector to model failsafe fault-tolerance, only the safeness and the stability of the detector pattern should be met. Thus, the composition of the DFS object model and the parallel detector pattern should satisfy the following LTL properties:

- *Safeness.* The safeness of the parallel detector requires that $\Box(Z_{DFS} \Rightarrow \Diamond X_{DFS})$.
- *Stability.* The stability requires that $\Box(Z_{DFS} \Rightarrow (Next(Z_{DFS} \vee \neg X_{DFS})))$ always holds.

Also, all participants of the parallel detector pattern must meet the safeness and the stability requirements. For example, the participant $d_{pressure}$ should meet the following:

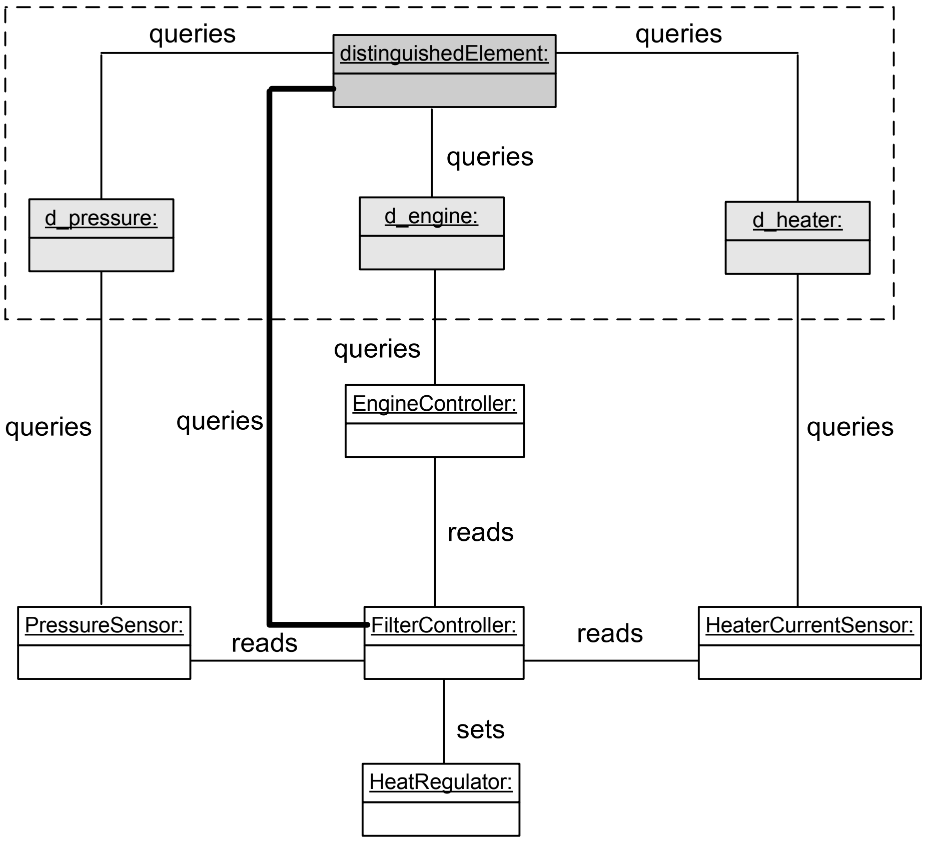


Figure 32: The composition of a parallel detector and the object model of the diesel filter system.

- *Safeness.* The safeness of $d_{pressure}$ requires that $\Box(Z_{pressure} \Rightarrow \Diamond X_{pressure})$.
- *Stability.* The stability requires that $\Box(Z_{pressure} \Rightarrow (Next(Z_{pressure} \vee \neg X_{pressure})))$ always holds.

Collaborations. The FilterController object is the client of the parallel detector pattern since it checks the truth-value of Z_{DFs} before it issues any command signals to the heater. We illustrate this collaborative relation between the parallel detector pattern and the FilterController object in the state diagram of Figure 33.

Interference-Freedom Constraints. In order to ensure that the object model of the DFS and the instance of the parallel detector pattern do not interfere, we verify the following requirements (using model checking techniques):

- *In the absence of faults*, we verify that if the FilterController object is in the Cleaning state, then X_{DFS} must hold. Also, we verify that the FilterController object never deadlocks.
- *In the presence of faults*, we verify the safeness and the stability of the parallel detector pattern. Moreover, we verify that the FilterController object never violates the safety requirements of the DFS.

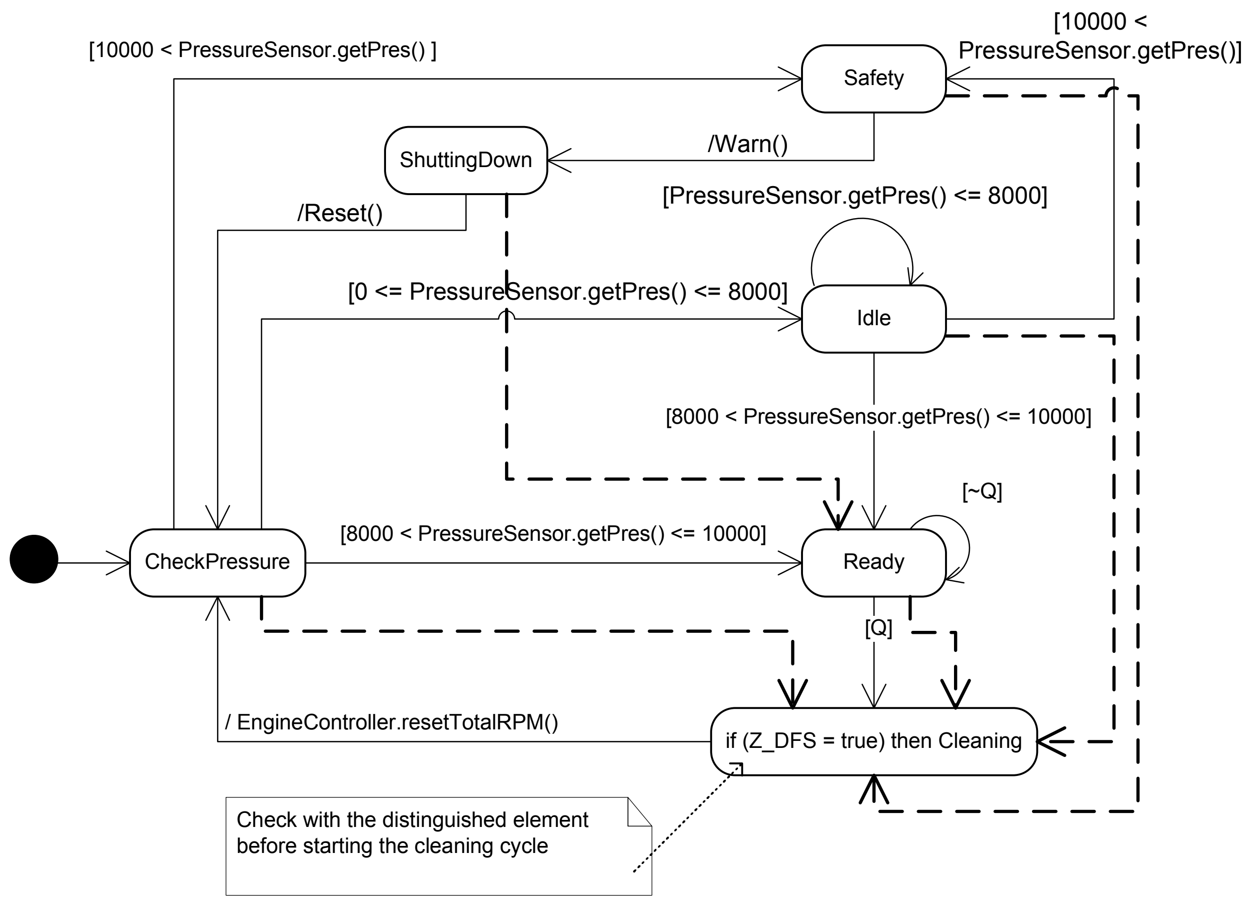

[10000 < PressureSensor.getPres()]
Safety
[10000 < PressureSensor.getPres()]
ShuttingDown
/Warn()
[PressureSensor.getPres() <= 8000]
/Reset()
[0 <= PressureSensor.getPres() <= 8000]
Idle
[8000 < PressureSensor.getPres() <= 10000]
[~Q]
CheckPressure
[8000 < PressureSensor.getPres() <= 10000]
Ready
[Q]
/ EngineController.resetTotalRPM()
if (Z_DFS = true) then Cleaning
Check with the distinguished element before starting the cleaning cycle

Figure 33: The state diagram of the filter controller after the addition of the parallel detector.

10 Related Work

Several approaches [19, 53, 54] focus on modeling and analyzing fault-tolerance in UML, which differ from our approach in the way they model faults and fault-tolerance concerns. Dondossola and Botti [19] present a methodology for the specification of fault-tolerance requirements in UML, where they model Faults, Errors, and Failures (FEF) as a class hierarchy and associate the elements of FEF hierarchy to UML class diagrams of the fault-intolerant system. Shui *et al.* [54] model the occurrence of faults as exceptional conditions and fault-tolerance concerns as exception handlers, where the system execution must stop when exceptions occur so that exception handlers become active. Stoller and Schneider [53] present a method for automated analysis of distributed fault-tolerant systems with terminating computations based on message flow graphs, where they use graph-theoretic methods to analyze the system behaviors in the presence of failure scenarios. Using their approach, it is difficult to capture the non-terminating behavior of embedded systems. All the above approaches focus on a limited notion of faults (e.g., component failure) and a particular fault-tolerance mechanism (e.g., replication).

While the detector and corrector patterns are used to create and analyze the conceptual model of fault-tolerant systems during the analysis phase, various approaches [5, 7, 55] exist that can be used for the refinement of the UML model of fault-tolerant systems to design and implementation phases. For example, Beder *et al.* [5] use design patterns to implement Coordinated Atomic Actions (CAA) in the design and implementation of error-handling mechanisms in distributed O-O systems. Also, Tichy *et al.* [55] present a pattern-based approach for the design of service-based systems in UML, based on the redundancy [2] of system modules. They also use architectural design and deployment restrictions to develop linear programming constraints in the design of fault-tolerant systems, however, their approach could only be applied for detectable faults; i.e., faults whose occurrence can be detected by the subject system.

In summary, our approach aims to bridge the gap between the theory of fault-tolerance presented in [13, 14, 56] and the development of fault-tolerant systems. Specifically, our goal is to develop a roundtrip engineering process in UML, where we expect to benefit from (1) the generality of the existing theoretical results [13, 14] in modeling faults and fault-tolerance, and (2) the techniques for automatic addition of fault-tolerance [56] in providing automation in our roundtrip engineering process.

11 Conclusions

In this paper, we presented a method for modeling faults and fault-tolerance in UML for embedded systems. Specifically, we introduced two object analysis pattern, called the *detector* and *corrector* patterns, for modeling and analyzing different levels of fault-tolerance, where instances of the detector and corrector patterns are added to the UML model of existing systems to create the UML model of their fault-tolerant version. The detector and corrector patterns also provides a set of constraints for verifying the consistency of functional and fault-tolerance requirements. We extended McUmber and Cheng's formalization framework [9, 10] to generate formal specifications of the UML model of fault-tolerant systems in Promela [16]. Subsequently, we used the SPIN model checker [17] to detect the inconsistencies of fault-tolerance and functional requirements. To facilitate the automated analysis of fault-tolerance, we developed the visualization tool Theseus that animates counterexample traces and generates corresponding sequence diagrams at the UML level. The automated analysis with the SPIN model checker coupled with Theseus create a roundtrip engineering process of modeling fault-tolerant systems.

The use of the detector and corrector patterns simplifies the analysis and design of fault-tolerant systems as these patterns provide modularity, traceability, and separation of concerns. Such separation of concerns is especially helpful in analysis and design of *multitolerant* systems [56, 57], where a multitolerant system is subject to n fault-types $f_1, \cdots, f_n$ and provides different levels of fault-tolerance corresponding to each type of faults. One approach for creating the UML model of a multitolerant system is to incrementally model fault-tolerance corresponding to each f_i and use the resulting UML model of each step in subsequent steps for modeling f_j-tolerance, where $j > i$. The use of the detector pattern will facilitate reasoning about the fault-tolerance provided against each fault-type f_i, and the interference-freedom of failsafe f_i-tolerance and failsafe f_j-tolerance, where $(1 \leq i, j \leq n) \wedge (i \neq j)$. Also, using the detector pattern, it would be easier to trace the concern of failsafe f_i-tolerance from analysis to design and implementation by focusing on the instances of the detector pattern that have been added for providing failsafe f_i-tolerance. The use of the detector and corrector patterns can also decrease the complexity of managing UML models since different instances of the detector (respectively, corrector) pattern are associated with different detection (respectively, correction) constraints. Thus, while developers analyze the detection (respectively, correction) requirements of one instance of the detector (respectively, corrector) pattern, other instances of the detector (respectively, corrector) pattern can be concealed from developers' view, which makes the UML models modular and more manageable.

As an extension of this work, we are investigating the application of automatic synthesis techniques [56] in generating the UML model of fault-tolerant embedded systems once they have been analyzed for errors.

References

[1] B. Randall. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, pages 220–232, 1975.

[2] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[3] F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, 11(1), 1985.

[4] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[5] D.M. Beder, B. Randall A. Romanovsky, C.R. Snow, and R.J. Stroud. An application of fault-tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *ACM SIGOPS Operating System Review*, 34(4), 2000.

[6] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[7] Titos Saridakis. A system of patterns for fault-tolerance. *The 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 535–582, 2002.

[8] UML profile for modeling quality of service and fault tolerance characteristics and mechanisms. www.omg.org/docs/ptc/04-06-01.pdf, 2002.

[9] W.E. McUmber and B.H.C. Cheng. A general framework for formalizing UML with formal languages. *In the proceedings of 23rd International Conference of Software Engineering*, pages 433 – 442, 2001.

[10] W.E. McUmber. *A Generic Framework for Formalizing Object-Oriented Modeling Notations for Embedded Systems Development.* PhD thesis, Michigan State University, 2000.

[11] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[12] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

[13] A. Arora. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, 1992.

[14] A. Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 436–443, May 1998.

[15] S. S. Kulkarni. *Component-based design of fault-tolerance.* PhD thesis, Ohio State University, 1999.

[16] Spin language reference. http://spinroot.com/spin/Man/promela.html.

[17] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.

[18] Ali Ebnenasir and Sandeep S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. http://www.cse.msu.edu/ sandeep/software.

[19] G. Dondossola and O. Botti. System fault tolerance specification:Proposal of a method combining semi-formal and formal approaches. *Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering (FASE), LNCS*, 1783:82–96, 2000.

[20] H.H. Ammar, S.M. Yacoub, and A. Ibrahim. A fault model for fault injection analysis of dynamic UML specifications. *International Symposium on Software Reliability Engineering, LNCS*, 2001:74 – 83, 2001.

[21] Sandeep S. Kulkarni and Ali Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *To appear in IEEE Transactions on Dependable and Secure Computing*, 2(3), July-September 2005.

[22] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[23] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-oriented Software Engineering: A use case driven approach.* Addison-Wesley, 1992.

[24] A. Ebnenasir, B.H.C. Cheng, and S.S. Kulkarni. Modeling faults and failsafe fault-tolerance in UML. *Submitted to the 28th International Conference on Software Engineering*, 2006.

[25] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Transaction on Software Engineering*, 28(9):889 – 903, 2002.

[26] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[27] Martín Abadi and Gordon D. Plotkin. A logical view of composition and refinement. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 323–332, Orlando, Florida, January 1991.

[28] R.J.R. Back and K. Sere. Stepwise refinement of parallel programs. *ACM Transactions on Software Engineering and Methodology*, 3(4):133–180, 1994.

[29] M. J. Fischer, N. .A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):373–382, 1985.

[30] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *In Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, 2000.

[31] M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, 2002.

[32] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[33] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[34] A. Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998. A preliminary version appears in the Proceedings of the Fourteenth Symposium on Reliable Distributed Systems, Bad Neuenahr, 174–185, 1995.

[35] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[36] G. Sindre and A. L. Opdahl. Eliciting security requirements by misuse cases. *In Proceedings of TOOLS Pacific*, pages 120 – 131, November 2000.

[37] I. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 2(1):58–66, January - February 2003.

[38] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. *6th International Symposium on the Foundations of Software Engineering (FSE-6), Orlando FL*, pages 56–69, November 1998.

[39] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99 – 123, 2001.

[40] S. Sendall and A. Strohmeier. From use cases to system operation specifications. *The Unified Modeling Language. Advancing the Standard: Third International Conference, Lecture Notes in Computer Science*, 1939, 2000.

[41] V. Mencl. Deriving behavior specifications from textual use cases. *in Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04), Linz, Austria*, pages 331–341, September 2004.

[42] V. Mencl, F. Plasil, and J. Adamek. Behavior assembly and composition of use cases - uml 2.0 perspective. *in Proceedings of the Software Engineering (SE) 2005 conference, Innsbruck, Austria*, pages 193–201, February 2005.

[43] L.A. Campbell. *Enabling Integrative Analyses and Refinement of Object-Oriented Models with Special Emphasis on High-Assurance Embedded Systems*. PhD thesis, Michigan State University, 2004.

[44] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[45] N. Mittal and V. K. Garg. On detecting global predicates in distributed computations. *In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA*, pages 3–10, April 2001.

[46] E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[47] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1990.

[48] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 8:666 – 677, August 1978.

[49] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

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

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

[52] S. Konrad, B. H.C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970 – 992, 2004.

[53] S.D. Stoller and F.B. Schneider. Automated analysis of fault-tolerance in distributed systems. *Formal Methods in System Design*, 26(2):183–196, March 2005.

[54] A. Shui, S. Mustafiz, J. Kienzle, and C. Dony. Exceptional use cases. *In the Proceedings of 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS*, 3713:568–583, 2005.

[55] M. Tichy, D. Schilling, and H. Giese. Design of self-managing dependable systems with uml and fault tolerance patterns. *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS), Newport Beach, CA*, pages 105 – 109, 2004.

[56] Ali Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, 2005.

[57] Sandeep S. Kulkarni and Ali Ebnenasir. Automated synthesis of multitolerance. *In Proceedings of International Conference on Dependable Systems and Networks (DSN), Palazzo dei Congressi, Florence, Italy*, pages 209–218, July 2004.