Pattern-Based Modeling and Analysis of Failsafe Fault-Tolerance in UML[*]

Ali Ebnenasir[†]
Department of Computer Science
Michigan Technological University
Houghton MI 49931 U.S.A.
aebnenas@mtu.edu

Betty H.C. Cheng
Computer Science and Engineering Department
Michigan State University
East Lansing MI 48824 U.S.A.
chengb@cse.msu.edu

Abstract

In order to facilitate incremental modeling and analysis of fault-tolerant embedded systems, we introduce an object analysis pattern, called the detector pattern, that provides a reusable *strategy for capturing the requirements of failsafe fault-tolerance in an existing conceptual model, where a failsafe system satisfies its safety requirements even when faults occur. We also present a method that (i) uses the detector pattern to help create a behavioral model of a failsafe fault-tolerant system in UML, (ii) generates and model checks formal models of UML state diagrams of the fault-tolerant system, and (iii) visualizes the model checking results in terms of the UML diagrams to facilitate model refinement. We demonstrate our analysis method in the context of an industrial automotive application.*

Keywords: Requirements Analysis, Fault-Tolerance, Formal Methods, Detector, UML

1 Introduction

The complexity of developing fault-tolerant systems is, in part, due to the crosscutting and evolving nature of fault-tolerance requirements. Since it is difficult to anticipate all types of faults[2] at early stages of development, it is better to have techniques that enable existing analysis artifacts to be revised once a new type of fault is detected. Such a revision would potentially crosscut all components of the conceptual model of an existing system. As such, developers need rigorous techniques and reusable artifacts for adding fault-tolerance concerns to conceptual models. This paper introduces a pattern-based approach for adding failsafe fault-tolerance to an existing conceptual model (in UML [7]), where a failsafe fault-tolerant system is expected to meet its safety requirements even when faults occur.

Numerous approaches exist to support fault-tolerance [1, 5, 14, 30, 32, 33] and to analyze system safety in the presence of failures [6, 23, 27], most of which (i) rely on a specific fault-tolerance design/implementation mechanism; (ii) do not emphasize on reuse in analysis phases, and (iii) lack sufficient support for formal analysis of the potential interferences between fault-tolerance and functional concerns. For example, Saridakis [32] presents a set of design patterns based on existing recovery mechanisms [14,30]. Bondavalli *et al.* [6] translate UML structural and behavioral diagrams to Stochastic Petri Nets in order to provide quantitative predictions for system dependability/reliability. Leveson and Stolzy [27] present a formal framework based on Timed Petri Nets to model and analyze fault-tolerance in real-time systems. The UML profile for fault-tolerance [1] and several aspect-oriented approaches [18, 34] use redundancy of services [33] to *mask* faults, which is sometimes impractical and costly [2]. Approaches based on safety cases [23] present a framework for goal-based failure analysis and systematic specification of lessons and recommendations for post-failure corrections of safety-critical systems. Furthermore, some existing analysis methods for fault-tolerance [22, 31] assume that a specific fault-tolerance design mechanism (e.g., exception handling, redundancy) will be used and specify analysis requirements within those design constraints, which may overly constrain and preclude useful fault-tolerance solutions. (For example, it is difficult to specify and model self-stabilization [9] *solely* based on exception handling.) While all aforementioned approaches present useful and important techniques for modeling and analyzing fault-tolerance concerns, three important features

[*]This work was partially sponsored by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, CNS-0551622, CCF-0541131, NSF CAREER CCR-0092724, ONR grant N00014-011-0744, DARPA Grant OSURS01-C-1901, Siemens Corporate Research, a grant from the Michigan State University's Quality Fund, and a grant from Michigan Technological University.

[†]The work presented here was performed largely while this author was a postdoctoral researcher at Michigan State University with support from NSF and ONR.

[1]Email: aebnenas@mtu.edu, chengb@cse.msu.edu
Tel: +1-906-487-4372, Fax: +1-906-487-2283

[2]A fault-type is a representation of the hypothesized cause of an error. An error is characterized by a (set of) state(s) from where failures may occur. A failure is a behavioral deviation from system specifications [4].

distinguish our approach from existing work: (1) providing *reusable* modeling artifacts for incremental modeling of fault-tolerance; (2) ensuring the fault-tolerance of the modeling elements added for fault-tolerance purposes, and (3) facilitating automated reasoning (coupled with visualization) in analyzing the mutual impact of fault-tolerance and functional concerns.

We introduce an object analysis pattern, called the *detector* pattern, that provides a reusable strategy for eliciting and specifying the requirements of error detection in UML object models for embedded systems. Our focus on capturing error detection requirements is an extension of Arora and Kulkarni's [3] results where they show that detection is *necessary and sufficient* for developing a rich class of failsafe fault-tolerant systems. As such, the detector pattern is intended to provide a generic artifact in model-driven development of failsafe systems. Object analysis patterns apply a similar approach to that used by design patterns [19], but instead of focusing on design they address the construction of the conceptual model of a system [10]. Patterns for the analysis stage of software development are not new (see [17,24]). For example, Fowler [17] presents a method for characterizing recurring ideas in business modeling as reusable analysis patterns. Konrad *et al.* [24] present domain-specific object analysis patterns for analyzing the conceptual models of embedded systems. The proposed detector pattern in this paper provides a reusable building block for the construction of the conceptual models of failsafe systems.

Our pattern-based method comprises fault modeling, fault-tolerance modeling, and automated analysis of the UML models of fault-tolerant embedded systems. Specifically, to construct the UML model of a Fault-Tolerant System (FTS), we start with the UML model of its fault-intolerant version, where a Fault-Intolerant System (FIS) meets its functional requirements in the absence of faults (i.e., when no faults occur) and provides no guarantees in the presence of faults (i.e., when faults occur). Then we model faults in the UML model of the FIS to produce a *model with faults*. We use the notion of state perturbation to model different types of faults in UML state diagrams [2,9]. Next, we specify error states that are reached due to the occurrence of faults. Subsequently, we add instances of the detector pattern to the model with faults to capture the requirements of error detection and to generate a *candidate* UML model of a failsafe FTS. To create a valid UML model of the failsafe FTS, we have to ensure that the candidate UML model is *interference-free*. That is, in the absence of faults, the candidate model meets all functional requirements of the FIS, and in the presence of faults, the candidate model at least meets the safety requirements of the FIS and the instances of the detector pattern. To ensure interference-freedom, we extend McUmber and Cheng's formalization framework [28] to generate formal specifications of faults, fault-tolerance and functional concerns in the Promela modeling language [21]. Subsequently, we use the Spin model checker [21] to detect inconsistencies between the detector pattern instances and the functional model of the FIS. The automated analysis with the Spin model checker coupled with a new visualization tool, called Theseus [20], that animates counterexample traces enables a roundtrip engineering process for modeling and analyzing failsafe FTSs.

We demonstrate our approach by modeling and analyzing an adaptive cruise control (ACC) system in UML. We have also validated our approach for several other examples from industry [12], including a diesel filter system for reducing soot from diesel truck exhaust. The remainder of this paper is organized as follows. Section 2 introduces an approach to modeling faults in UML state diagrams. Section 3 discusses the relation between failsafe fault-tolerance and error detection. Section 4 presents the detector pattern. Section 5 focuses on formal analysis of UML models of FTSs using Spin [21]. Finally, Section 6 gives concluding remarks and discusses future work.

2 Modeling

In this section, we present the basic concepts of modeling FISs, faults, and failsafe fault-tolerance in UML. We reiterate the definitions of faults and fault-tolerance from [2]. The motivation behind using UML is two-fold. First, UML is the *de facto* standard for object-oriented modeling. Second, UML state diagrams enable us to capture any form of fault-tolerance that can be expressed in a state machine-based formalism.

UML Models. We use conventional UML notations [7] to represent the UML-based conceptual models of FISs (respectively, FTSs). Since our focus is on modeling and analyzing fault-tolerant embedded systems, we follow Douglass [10] in using UML class diagrams to model structural constraints of (software and hardware components of) embedded systems during the object analysis phase. We use state diagrams to capture high-level *behavioral* information of UML object models. The combination of class and state diagrams yields an *object analysis model*.

Depending on the semantics of object interactions, the complexity of automatic analysis of a FTS varies from polynomial (in a shared memory model [25]) to undecidable (in an asynchronous message-passing model [16]). To facilitate an automated analysis method with a manageable complexity, in this paper, we consider a *high atomicity* model where transitions of UML state diagrams are executed atomically, and any instance of message passing between two objects takes place in an atomic step. Another motivation behind this assumption is that modeling fault-tolerance in a high atomicity model provides an *impossibility test* in early stages of development (which could potentially reduce development costs). That is, if a conceptual model of an FTS cannot be derived from the conceptual model of its fault-intolerant version in the high atomicity model, then it would be impossible to derive a model of the FTS in a lower atomicity level.[3]

Underlying Computational Model. In an UML object model $M = \langle O_1, \cdots, O_n \rangle$, where each O_i ($1 \leq i \leq n$) is an object, we denote the state transition diagram of each object O_i by $SD_i = < S_i, \delta_i >$, where S_i is the set of states in the state diagram SD_i and δ_i denotes the set of transitions of SD_i. A state of an object O_i is a valuation of its state variables (i.e., attributes). A transition of an object O_i is of the form $(a, evt[grd]/act, b)$, by which O_i transitions from state a to state b if a triggering event evt occurs and a condition grd holds. During such a transition, O_i executes an action act. A *global state predicate* is a Boolean expression defined over a set of states of multiple objects. A *local state predicate* is a Boolean expression specified over the set of states of only one object O_i (i.e., S_i). A *computation* of an object O_j ($1 \leq j \leq n$) is a sequence $\langle s_0, s_1, \cdots \rangle$ such that $\forall s_i : i \geq 0 : (s_i \in S_j) \wedge (s_i, s_{i+1}) \in \delta_j$. A *computation* of an UML object model $M = \langle O_1, \cdots, O_n \rangle$ is a sequence of states $\langle s_0, s_1, \cdots \rangle$, where $\forall s_i : i \geq 0 : (\exists O_j : 1 \leq j \leq n : (s_i \in S_j) \wedge ((s_i, s_{i+1}) \in \delta_j))$.

Modeling Functional Requirements. We consider the set of *functional requirements* as a conjunction of a set of safety requirements and a set of liveness requirements. Intuitively, *safety requirements* stipulate that nothing bad ever happens, and the *liveness requirements* specify that something good will eventually occur in a finite amount of time. For example, in a cruise control system, the actual speed of the car must not exceed 1% of the desired speed set by the driver (i.e., safety), and when the driver applies the brakes, the cruise control system will eventually be deactivated (i.e., liveness). We represent safety requirements by a set of transitions, say $\mathcal{B}$, that must not occur in the computations of any object. We do not explicitly specify liveness requirements, instead, since we want to derive a model of an FTS from a valid model of its fault-intolerant version, we stipulate that during such transformations no deadlock states (states with no outgoing transitions) should be introduced in the absence of faults. The deadlock freedom requirement captures the fact that, in the absence of faults, fault-tolerant embedded systems have nonterminating computations and always react to their environment. We say a computation $\langle s_0, s_1, \cdots \rangle$ *meets safety requirements* iff (if and only if) $\forall i : i \geq 0 : (s_i, s_{i+1}) \notin \mathcal{B}$. A computation $\sigma = \langle s_0, s_1, \cdots \rangle$ *meets functional requirements* if and only if σ meets safety requirements and does not deadlock. A computation of an UML model M that meets the functional requirements of M is a formal representation of a *functional scenario*.

[3]A theoretical investigation of this claim can be found in [26]. Also, in cases such atomicity assumptions do not hold (e.g., distributed systems), one can use existing fault-tolerance-preserving refinements (e.g., [8]) to generate a refined model from a high atomicity model developed using our proposed approach.

Running Example: Adaptive Cruise Control (ACC). The ACC system comprises a standard cruise control system and a radar system to control the distance between the car and the front vehicle, called the *target vehicle*. The ACC system has different modes of operation (see Figure 1), namely *closing*, *coasting* and *matching*. When the radar detects a target vehicle, the ACC system enters the *closing* mode. 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 the target vehicle travels in a fixed amount of time (e.g., 2 seconds). The distance to the target vehicle must not be less than a *safety zone*, which is 90% of the *trail distance*. 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; i.e., the car enters the *coasting* zone. When the car reaches the trail distance, the relative speed of the car should be zero; i.e., the ACC system is in the *matching* mode. The safety requirements of the ACC system state that *the car is never in the safety zone and it will never accelerate in the coasting or matching modes.*

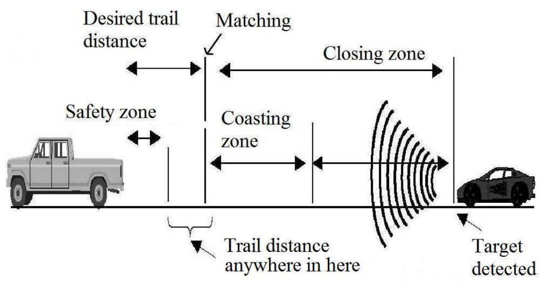

Figure 1. The adaptive cruise control system.

2.1 Modeling Faults in State Diagrams

We systematically model a fault-type as *a set of transitions* in UML state diagrams (see Figure 2). Representing faults as a set of transitions has already appeared in previous work [2, 9], and it is known that state perturbation is sufficiently expressive to represent different types of faults, such as crash, input-corruption and Byzantine, from different behavioral categories; i.e., transient, intermittent, permanent [2, 9]. To model a fault-type f in a UML model $M = \langle O_1, \cdots, O_n \rangle$, we model the effect of f on the state diagram SD_i of each object O_i by introducing a new set of transitions in SD_i denoted f_i, for $1 \leq i \leq n$ (see dashed arrows in Figure 2). We denote the set of *transitions of SD_i in the presence of faults f_i* by $\delta_i \cup f_i$. An object O_i does not have control over the execution of faults f_i, whereas the execution of regular transitions is controlled by the thread of execution in O_i (see solid arrows in Figure 2). Modeling a fault-type f in all state diagrams of the UML model

M creates a *model with faults* f. In a *computation with faults* $\langle s_0, s_1, \cdots\rangle$, there exists a fault transition (s_i, s_{i+1}), for some $i \geq 0$. A computation with faults is a formal representation of a *scenario with faults*.

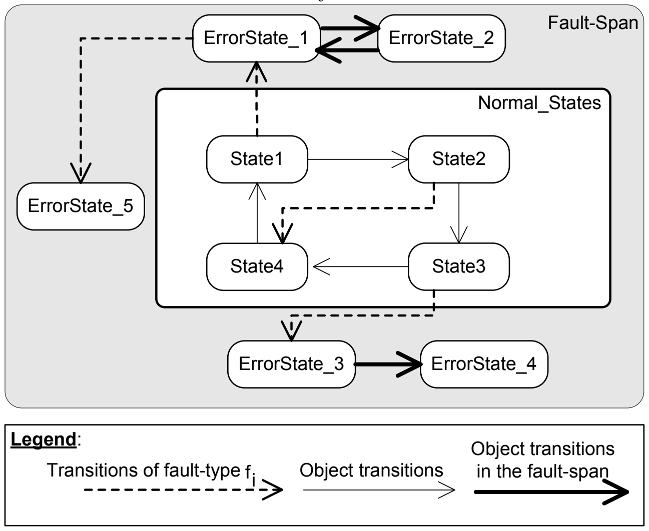

Figure 2. Modeling faults in UML state diagrams.

When modeling a fault type f_i in a state diagram SD_i of the UML model of the FIS, modelers should identify the effect of f_i on the behavior of O_i. In the absence of faults, an object O_i of the FIS is in a set of *normal states* from where it meets its functional requirements. When f_i occurs, O_i may reach error states that are outside of the set of normal states. The set of states reachable from the normal states by a combination of fault and regular transitions is the *fault-span* of O_i for fault f_i , denoted f_i-span of O_i [25]. For example, in Figure 2, all error states are only reachable when faults occur. Thus, modeling faults in a state diagram may introduce new states and transitions in that state diagram. A computation of an object that starts in its fault-span outside the set of normal states may lead to a failure scenario in which it may (i) violate safety requirements, (ii) fall into non-progress cycles, or (iii) reach a deadlock state. For example, in Figure 2, if the object is in State1 then the faults f_i may non-deterministically transition to ErrorState_1 from where the object may either be trapped in a non-progress cycle (comprising ErrorState_1 and ErrorState_2) or be deadlocked in ErrorState_5. Since manual identification of f_i-span is a tedious task, we use our previously develop Fault-Tolerance Synthesizer (FTSyn) [13] to automatically generate the fault-span.

ACC Example: *The object model of the ACC system includes three main objects, namely* Control, Car *and* Radar *(see Figure 5). (We use* Sans Serif *font to denote system variables, objects and states.) The* Control *object captures the controlling activities that set the mode of the ACC system. The* Car *object models the engine management functionalities such as acceleration and deceleration. The* Radar *object samples the speed and the distance of the car to the target vehicle. Figure 3 illustrates an excerpted state diagram of the* Car *object in the ACC system. The* Car *object continuously compares the real speed of the car, saved in the variable* RealV*, with respect to the* setpoint*, which is the desired speed determined by the driver. If the setpoint is less than the real speed of the car, then the* Car *object transitions to the* Decelerating *state. If the setpoint is greater than the real speed of the car, then the* Car *object transitions to the* Accelerating *state. The ACC system is subject to a transient fault-type f_{ACC} that causes the* Car *object to transition to the* Accelerating *state non-deterministically. Hence, we model the effect of f_{ACC} on the* Car *object as a set of transitions f_{car} that perturb the state of* Car *to the* Accelerating *state. In this case, the fault-type f_{car} does not introduce any new states.*

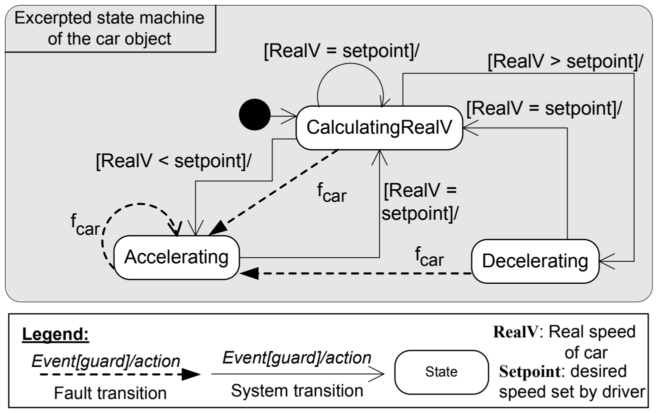

Figure 3. Modeling faults in the state transition diagram of the car.

2.2 Failsafe Fault-Tolerance

Intuitively, *failsafe* fault-tolerance requires that nothing bad ever happens even in the presence of faults [2]. For a fault-type f and a UML model M of an FIS that is subject to f, we want to derive a UML model M' from M such that M' satisfies the following conditions: (1) in the absence of f, the set of computations of M' is non-empty and is a subset of the set of computations of M, and (2) all computations of M', including the set of computations with faults f, meet safety requirements.

3 Necessity and Sufficiency of Detection

In this section, we intuitively explain the relation between error detection in the presence of faults and failsafe fault-tolerance. Specifically, to preserve safety requirements even if faults occur, a failsafe fault-tolerant system should ensure that none of its computations would reach a state from where faults may directly violate safety. Moreover, after faults perturb the state of a failsafe system outside its set of normal states, the system must not execute transitions that violate safety. To enable such functionalities, a failsafe system should be able to *detect* its current state and take necessary actions that will not lead to the violation of safety requirements. Hence, we define two categories of states that should be detected by a failsafe system, namely *fault-unsafe* and *at-risk* states. The set of fault-unsafe states, represented by a state predicate S_{unsafe}, captures the set of states in the fault-span from where a sequence of fault transitions alone may violate safety requirements. A failsafe FTS must never reach a state in S_{unsafe}. In the set of at-risk states, denoted as a state predicate $S_{at-risk}$, a FIS itself may execute actions that violate safety requirements. (FTSyn [13] automatically identifies these state predicates.) For example, in the ACC system, a global state where the control is in the coasting mode and the car is in the Accelerating state is an at-risk state since the car may accelerate and violate the requirement of *no acceleration while coasting*. (A soundness proof of the above argument about necessity and sufficiency of detection for failsafe fault-tolerance can be found in [3, 25].)

4 Detector Pattern

In this section, we introduce the detector pattern that we use to augment the FIS conceptual model to derive a conceptual model of a failsafe FTS while preserving the safety and liveness requirements of the FIS in the absence of faults. In order to facilitate its use, we define a template for the detector pattern based on the fields used in the design patterns presented by Gamma *et al.* [19], with modifications to reflect analysis-level information. For example, we do not use the Implementation and Sample Code fields. The Structure field captures structural constraints of the detector pattern represented by UML class diagrams. The detector pattern also includes several new fields that are added for the purpose of specifying and analyzing fault-tolerance concerns. For example, the detector pattern includes the Detection Requirements field that specifies a set of requirements that must be met by the resulting UML model to guarantee that the detection occurs correctly. Next, we describe the fields of the detector pattern. Example application of each field to the ACC system is denoted in *italics*.

Detection Predicate. In a UML model $M = \langle O_1, \cdots, O_n \rangle$, a detection predicate is a global/local state predicate. In a distributed system, it is difficult for an object to detect a global detection predicate X in an atomic step [29]. Thus, we decompose X into a set of local predicates $X_1, \cdots, X_n$, and specify the detection of X based on the detection of $X_1, \cdots, X_n$, where each X_i is a local state predicate specified for object O_i. Since the state predicate S_{unsafe} captures the local unsafe states of each object, S_{unsafe} is often specified as a conjunction of a set of local state predicates. For the same reason, $S_{at-risk}$ most often has a conjunctive form. Hence, we limit the scope of the application of the detector pattern to the detection of conjunctive predicates. Moreover, since embedded systems frequently detect the conditions of their underlying physical system, the corresponding conditions remain stable relative to the processing speed of the embedded computing system until the embedded system detects them and takes necessary actions. Hence, in this section, we focus on stable global predicates that once hold remain true until detected.

ACC Example: In order to preserve safety requirements in the presence of fault f_{ACC}, the ACC system should detect if it is in the coasting or matching mode before accelerating the car. The corresponding detection predicate in the ACC system is the predicate $X_{ACC} \equiv X_{car} \wedge X_{control}$, where $X_{car} \equiv$ (Car is in the accelerating state) and $X_{control} \equiv$ ((Control is in the coasting mode) $\vee$ (Control is in the matching mode)).

Detector Elements (Participants). A detector element d_i, $1 \leq i \leq n$, captures the detection of a local state predicate X_i in a functional object O_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.[4]

Distinguished Element. An element d_{index} ($1 \leq index \leq n$) that finalizes the detection of X based on the detection of $X_1, \cdots, X_n$ is called the *distinguished element*.

Structure. The choice of the structure of the detector pattern depends on the inter-object associations in a UML object model. For example, if one needs to detect a global predicate over a set of functional objects that are associated with each other in a tree-like structure, then it is appropriate to use an instance of the detector pattern with a hierarchical (parallel) structure. Due to space constraints, we omit the presentation of sequential and compositional detector patterns (see [12] for details). In a parallel detector (see Figure 4), the detection of X can be done in parallel, where all elements d_i, $1 \leq i \leq n$, detect their detection predicates concurrently. The shadowed objects represent the elements of the detector pattern encapsulated in a dashed box that denotes an instance of the detector pattern. The distinguished element of the detector pattern is depicted by the dark shading. Each detector participant d_i is associated with an object O_i ($1 \leq i \leq n$). In Figure 4, the distinguished element is associated with all participants d_i.

ACC Example: Figure 5 depicts an excerpted class diagram of the ACC system in which an instance of the parallel detector pattern has been instantiated. Such an instantiation is performed manually as a modeling activity. The instance of the detector pattern applied to the ACC system comprises two elements $d_{control}$ and d_{car} modeled as two new objects in the class diagram of the ACC system. The

[4]In the design and implementation phases, the detector elements may be realized as independent software/hardware components that execute concurrently with other components of an embedded system. We conjecture that any additional execution overhead on system performance incurred by adding detector elements would not be worse than the use of conventional redundancy mechanisms.

element $d_{control}$ is responsible for detecting $X_{control}$ and the element d_{car} should detect X_{car}.

Witness Predicate. Since we decompose the global detection predicate X into a set of local detection predicates $X_1, \cdots, X_n$, we should specify what implies the truth value of X. Towards this end, we introduce the notion of a *witness* predicate Z that is a local condition belonging to the distinguished element, which implies that the detection is complete by just communicating with detector elements. We also consider a witness predicate Z_i for each element d_i. We say d_i *witnesses* X_i iff Z_i is *true*. The distinguished element d_{index} sets the value of Z to *true* if all d_i have witnessed their detection predicates.

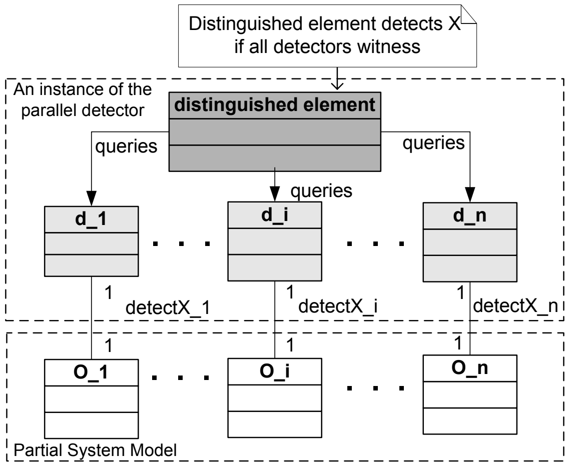

Figure 4. The structure of the parallel detector.

ACC Example:
The distinguished element should set Z_{ACC} (i.e., the witness predicate) to true if the elements $d_{control}$ and d_{car} witness their detection predicates $X_{control}$ and X_{car}, thus indicating that the global detection predicate X_{ACC} has become true. Such a detection functionality enables the FTS to accelerate only if X_{ACC} does not hold.

Detection Requirements. In order to ensure that the detection occurs correctly, the detector pattern should meet the following requirements (adapted from [3]): (1) *Safeness.* It is never the case that the witness predicate Z is *true* when the detection predicate X is *false*; i.e., the detector pattern never lies. (2) *Progress.* It is always the case that if X is true then Z will eventually hold. (3) *Stability.* It is always the case that once Z becomes *true*, it will remain *true* as long as predicate X is *true* (i.e., Z remains *stable*). Each participant d_i should also meet the above requirements for Z_i and X_i.

The safeness and stability are safety requirements that can be specified in Linear Temporal Logic (LTL) [15] using (i) the universal operator $\Box$, where $\Box Y$ means that the state predicate Y always holds, and (ii) the next state operator $\bigcirc$, where $\bigcirc Y$ means that in the next state Y holds. We respectively specify *safeness* and *stability* as $\Box(Z \Rightarrow X)$ and $\Box(Z \Rightarrow (\bigcirc(Z \vee \neg X)))$. Using the eventuality operator $\Diamond$, where $\Diamond Y$ means that the state predicate Y eventually holds, we specify *progress* as $\Box(X \Rightarrow \Diamond Z)$.[5]

Behavior. The state diagram of each detector element d_i in Figure 4 is composed with the state diagram of each object O_i in a concurrent fashion. The state machine of a detector element d_i monitors X_i by reading the state of O_i. The distinguished element can witness if all the detector participants d_i have already witnessed their detection predicates.

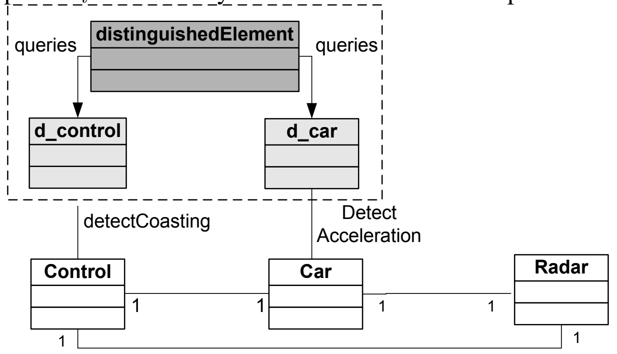

Figure 5. Composition of a parallel detector pattern with the ACC system.

ACC Example: Before accelerating, the Car *object communicates with the distinguished element to check whether the witness predicate Z_{ACC} holds (see Figure 6). Even though in the case of the ACC system, such detection can also be done by calling a method of the* Control *object to check its mode, the use of the detector pattern separates the concerns of fault-tolerance from functional concerns and enables developers to easily trace and reason about fault-tolerance. Moreover, in cases where more than two objects are involved in the detection of a global predicate, it is difficult to use functional method calls for providing fault-tolerance.*

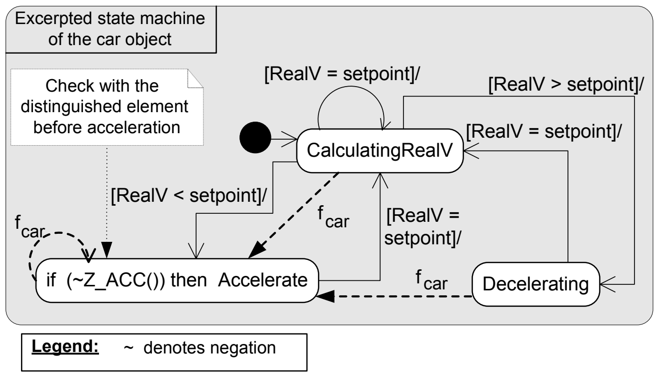
Excerpted state machine of the car object
Check with the distinguished element before acceleration
[RealV = setpoint]/
[RealV > setpoint]/
[RealV = setpoint]/
CalculatingRealV
[RealV < setpoint]/
f_car
f_car
[RealV = setpoint]/
if (~Z_ACC()) then Accelerate
f_car
Decelerating
Legend: ~ denotes negation

Figure 6. The excerpted state diagram of the Car after applying the parallel detector pattern.

Failsafe fault-tolerance of the Detector pattern. Since the instances of the detector pattern are also subject to

[5]Note that the detection requirements can also be represented in terms of Dwyer's specification patterns [11].

faults, we ensure that the detector pattern is itself failsafe fault-tolerant. To this end, we verify that the composition of the UML model M of an FIS, and an instance of the detector pattern meet the following requirements: (1) *In the absence of faults*, the functional requirements of M are met (i.e., safety is not violated and no deadlock is reached), and (2) *In the presence of faults*, the safeness and the stability of the detector pattern and the safety requirements of M are satisfied. Note that when faults occur, the only requirement for a failsafe FTS is to preserve its safety requirements. Thus, the detector pattern need not meet its progress requirement when faults occur. For example, in the ACC example, if X_{ACC} holds but Z_{ACC} never becomes true, then safety requirements are still met since the Car object never accelerates if Z_{ACC} is false. However, the liveness of the ACC system may be violated.

Remark. While the ACC example is small, the number of the elements of an instance of the detector pattern cannot go beyond the number of system components. Moreover, even though composing an instance of the detector pattern with the functional model of an FIS may add a layer of complexity, the modularity provided by the detector pattern facilitates the management of such complexity. (Other pattern-driven methods may also suffer from this additional layer of complexity introduced by pattern instantiation.)

5 Model Analysis

In order to verify whether composing an instance of the detector pattern with the conceptual model of a FIS generates a valid model of a failsafe system, we extend the Hydra formalization framework [28] to generate a Promela specification of a *candidate* UML model of an FTS. Subsequently, we use the Spin model checker [21] to detect potential inconsistencies between the instances of the detector pattern and the functional model of the FIS. The general UML-to-Promela formalization approach of Hydra is to map objects to processes in Spin that exchange messages via *channels*. Nested and concurrent states are also formalized as processes. Additional details on the modeling and analysis process, and the underlying formalization framework can be found in [28]. We use the current Hydra formalization directly to generate Promela specifications of the functional objects of the FIS and each d_i element as a separate process.

We extend Hydra to include a number of new formalization rules that treat the transitions of a fault-type f differently than other transitions. As we distinguish fault transitions from regular transitions by defining a <<Fault>> stereotype [7], the extended Hydra integrates the transitions of f modeled in different state diagrams in a separate process Fault_f in Promela that is concurrently run with all other processes. Such a formalization is advantageous in that the resulting Promela model separates faults from the functional part of the Promela specifications so that the effect of faults on system behaviors can easily be simulated and analyzed.

We use Spin to simulate and verify the Promela specifications generated by Hydra. For example, while verifying the UML model of a FTS against detection requirements, we may find counterexamples that represent the inconsistencies of the detector pattern and the functional objects. To analyze such inconsistencies, we use Theseus [20] to visualize each step of the Spin simulation in UML state diagrams. Such a visualization of counterexamples facilitates the analysis and refinement of the UML models.

ACC Example: In the UML model of the ACC system, we verified the safeness of the detector pattern to ensure that the detector pattern is itself failsafe $f_{detector}$-tolerant, where $f_{detector}$ represents the effect of f_{ACC} on d_{car}, which is the resetting of Z_{car}. We specified the safeness requirements as the LTL property $\Box(Z_{car} \Rightarrow X_{car})$; i.e., it is always the case that if d_{car} has witnessed then the Car *object is in the* Accelerating *state. A sample counterexample that we found was for the case where d_{car} witnessed that its detection predicate X_{car} holds, but the state of the* Car *object had been changed to another state without resetting Z_{car} (i.e., X_{car} was no longer true). In this case, the safeness of d_{car} was violated. The Theseus visualization tool highlighted all transitions that leave the* Accelerating *state of the* Car *object as the transitions that would violate the safeness of the detector pattern. To remedy this inconsistency, we manually revised the object model so that any transition originating from the* Accelerating *state would be accompanied with the simultaneous reset of Z_{car} (i.e., $Z_{car} := false$). Notice that, in this case, resolving the inconsistencies of the detector pattern and the functional model required a change in the functional model. We note that this change does not affect the computations of the functional UML model in the absence of faults.*

6 Conclusions and Future Work

In this paper, we introduced an object analysis pattern, called the *detector* pattern, for modeling and analyzing failsafe fault-tolerance, where instances of the detector pattern are added to the UML model of a system to create the UML model of its failsafe fault-tolerant version. The detector pattern also provides a set of constraints for verifying the consistency of functional and fault-tolerance requirements and the fault-tolerance of the detector pattern itself. We extended McUmber and Cheng's formalization framework [28] to generate formal specifications of the UML model of fault-tolerant systems in Promela [21]. Subsequently, we used the Spin model checker [21] to detect the inconsistencies between fault-tolerance and functional requirements. To facilitate the automated analysis of failsafe fault-tolerance, we employed the Theseus visualization tool [20] that animates counterexample traces and generates corresponding diagrams at the UML level. As an extension of this work, we are currently investigating the incorporation of timing issues in the detector pattern.

References

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

[2] 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.

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

[4] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January – March 2004.

[5] D. Beder, B. R. A. Romanovsky, C. Snow, and R. 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] A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analysis for supporting design decisions in UML. *IEEE International Symposium on High-Assurance Systems Engineering*, pages 64–71, 1999.

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

[8] M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, pages 589 – 597, 2002.

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

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

[11] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. *In Proceedings of the 21st International Conference on Software Engineering (ICSE99), Los Angeles, CA, USA*, pages 411–420, 1999.

[12] A. Ebnenasir and B. H. C. Cheng. A framework for modeling and analyzing fault-tolerance. Technical Report MSU-CSE-06-5, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, January 2006.

[13] A. Ebnenasir and S. S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. http://www.cs.mtu.edu/ aebnenas/research/tools/ftsyn.htm.

[14] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

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

[16] 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.

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

[18] R. France and G. Georg. An aspect-based approach to modeling fault-tolerance concerns. *Technical Report 02-102, Computer Science Department, Colorado State University*, 2002.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[20] H. Goldsby, S. Konrad, B. H. Cheng, and S. Kamdoum. Enabling a roundtrip engineering process for the modeling and analysis of embedded systems. *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), Genova, Italy*, October 2006.

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

[22] D. Ilic and E. Troubitsyna. Modeling fault tolerance of transient faults. *Proceedings of Rigorous Engineering of Fault-Tolerant Systems*, pages 84 – 92, 2005.

[23] T. P. Kelly and J. A. McDermid. A systematic approach to safety case maintenance. In *SAFECOMP*, pages 13–26, 1999.

[24] 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.

[25] 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.

[26] S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 441–449, 2003.

[27] N. G. Leveson and J. L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, 1987.

[28] W. 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.

[29] 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.

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

[31] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. *Software Practice and Experience*, 35:195–236, 2005.

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

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

[34] M. Tkatchenko and G. Kiczales. Uniform support for modeling crosscutting structure. *Appeared in AOM Workshop held in conjunction with AOSD*, 2005.