A Pattern-Based Approach for Modeling and Analysis of Error Recovery*

Ali Ebnenasir[1] and Betty H.C. Cheng[2]

[1] Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931, USA
aebnenas@mtu.edu
http://www.cs.mtu.edu/~aebnenas
[2] Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824, USA
chengb@cse.msu.edu
http://www.cse.msu.edu/~chengb

Abstract. Several approaches exist for modeling recovery of fault-tolerant systems during the requirements analysis phase. Most of these approaches are inclined by design techniques for recovery. Such design-biased analysis methods unnecessarily constrain an analyst when specifying recovery requirements. To remedy such restrictions, we present an object analysis pattern, called the *corrector* pattern, that provides a generic reusable strategy for modeling error recovery requirements in the presence of faults. In addition to templates for constructing structural and behavioral models of recovery requirements, the corrector pattern also contains templates for specifying properties that can be formally verified to ensure the consistency between recovery and functional requirements. Additional property templates can be instantiated and verified to ensure the fault-tolerance of the corrector pattern and the system to which the corrector pattern has been applied. We validate our analysis method in terms of UML diagrams, where we (1) use the corrector pattern to model recovery in UML behavioral models, (2) generate and model check formal models of the resulting UML models, and (3) visualize 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, Error Recovery, Corrector, UML

* 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.

1 Introduction

High costs and complexity of developing fault-tolerant distributed systems are largely due to the crosscutting nature of fault-tolerance concerns and the requirement for coordinated recovery by system components [1]. For example, in a distributed embedded system, it is difficult to preserve the safety requirements while providing recovery from transient faults. Late characterization of such inconsistencies between functional and fault-tolerance requirements may potentially increase the development costs because it would be more expensive to address such inconsistencies in design and implementation phases rather than the requirements analysis phase. As such, systematic modeling and analysis of error recovery is even more important. To facilitate such early modeling and analysis, this paper presents a pattern-based approach for modeling and analyzing *nonmasking* fault-tolerance in embedded software systems, where, in the ideal case, a nonmasking fault-tolerant system guarantees error recovery[3].

Numerous approaches exist for the *design and implementation* of recovery from error conditions in sequential [2, 3] and concurrent (respectively, distributed) programs [1, 4, 5]. For example, Randell [2] presents the concept of recovery blocks for implementing recovery in sequential programs and uses atomic actions for the design of error recovery in asynchronous concurrent programs [1]. Cristian [3] focuses on the concept of exceptional conditions and systematic handling thereof. Schneider [4] presents a replication-based method for recovery from failures in client-server distributed systems. Saridakis [6] presents a set of design patterns based on existing recovery mechanisms [5]. The UML profile for fault-tolerance [7] and several aspect-oriented approaches [8–10] use redundancy of services to *mask* faults, which is sometimes impractical and costly [11]. Moreover, most existing *analysis* methods for fault-tolerance [12–15] assume that a specific fault-tolerance design mechanism will be used (e.g., exception handling, redundancy) and specify analysis requirements within those design constraints. As such, error recovery requirements may be overly constrained and preclude useful solutions or even finding a solution. For example, it is difficult to specify and model self-stabilization [16] solely based on exception handling. While a specific error recovery mechanism should certainly be considered at design time based on the constraints of the problem at hand, we believe that, at the requirements analysis level, an abstract specification of error containment and state restoration in distributed systems helps developers to detect the inconsistencies between recovery and functional requirements independent of the design and implementation techniques.

In order to specify and analyze recovery for embedded systems, we introduce an object analysis pattern, called the *corrector* pattern, that provides a reusable strategy for eliciting and specifying error correction constraints in UML object models. Object analysis patterns apply a similar approach to that used by design

[3] We emphasize that our proposed approach facilitates the creation and analysis of the conceptual models of nonmasking fault-tolerant systems. For such models to be realized in practice, one has to use fault-tolerance preserving refinements to develop design and implementation artifacts.

patterns [17], but instead of focusing on design their focus is on late requirements analysis where a conceptual model of a system is built. Patterns for the analysis stage of software development are not new (see [18, 19]). For example, Fowler [18] presents a method for characterizing recurring ideas in business modeling as reusable analysis patterns. Konrad *et al.* [19] present domain-specific object analysis patterns for analyzing the conceptual models of embedded systems. We introduce the corrector pattern that serves to modularize the requirements of error recovery, thereby facilitating tracing and reasoning about recovery in different stages of system development.

Our method comprises fault modeling, recovery modeling, and automated analysis of the UML models of fault-tolerant embedded systems. Specifically, to construct the UML model of a nonmasking 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*. Instead of focusing on a specific fault-type (e.g., fail-stop, crash, Byzantine, etc.), we use the notion of state perturbation to model different types of faults in UML state/sequence diagrams [11, 16]. Next, we add instances of the corrector pattern to the model with faults to generate a *candidate* UML model of a nonmasking FTS. To generate a valid UML model of the nonmasking 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 meets recovery requirements; i.e., when faults stop occurring, the system will eventually recover from error conditions. To ensure interference-freedom, we extend McUmber and Cheng's UML formalization framework [20] to generate formal specifications of the candidate model in the Promela modeling language [21]. Subsequently, we use the SPIN model checker [21] to detect inconsistencies between the corrector pattern and the functional UML model. The automated analysis with the SPIN model checker coupled with a new visualization tool, called Theseus [22], that animates counterexample traces and generates corresponding sequence diagrams enables a roundtrip engineering process for modeling and analyzing recovery requirements.

We demonstrate our approach by modeling and analyzing an adaptive cruise control (ACC) system in UML. We have also validated the corrector pattern for several other examples [23] including a diffusing computation program for a hierarchical distributed system [24]. The remainder of this paper is organized as follows. Section 2 presents an overview of the proposed approach. Section 3 introduces an approach to modeling faults and nonmasking fault-tolerance in terms of UML state and sequence diagrams. Section 4 presents a systematic method for eliciting and specifying error conditions. Section 5 presents the corrector pattern. Section 6 focuses on formal analysis of the UML model of FTSs using the model checker SPIN [21]. Section 7 discusses related work. Finally, Section 8 gives concluding remarks and discusses future work.

2 Overview

In this section, we present an overview (see Figure 1) of our pattern-based modeling approach. Figure 1 illustrates the steps of our approach (including modeling faults, specifying error conditions, instantiating the corrector pattern, and automated analysis) annotated with the relevant paper section number on the lower left corner of each step. For a given FIS $\mathcal{S}$ and a fault-type f, we start from a *valid* UML model of $\mathcal{S}$ that captures all global properties of $\mathcal{S}$ that should hold in the absence of faults f. Subsequently, we model the effect of f on each component of $\mathcal{S}$ modeled as an object in UML. Then we specify the error conditions that denote the set of states from where recovery should be provided. Subsequently, to specify the requirements of detecting and correcting error conditions, we compose instances of the proposed corrector pattern with the UML model of $\mathcal{S}$, which results in creating a candidate UML model of a nonmasking version of $\mathcal{S}$. To ensure the correctness of such a composition, we first employ an extended version of the Hydra [20] formalization tool to generate the Promela specifications of the candidate UML model. Then we use the SPIN model checker to verify the correctness of the composition. If the model checking is successful, then the candidate model is indeed a UML model of a nonmasking fault-tolerant version of $\mathcal{S}$. Otherwise, using the Theseus visualization tool [22], we animate the analysis errors (illustrated as counterexamples) in the state and sequence diagrams. Using such a visualization, we help developers to revise the candidate model to eliminate the inconsistencies between functional and recovery requirements. The revised model can again be model checked until the model checking is successful or an upper bound is reached in the number of model checking attempts.

In our approach, we separate the functional concerns from fault-tolerance concerns and start with a valid UML model of the FIS. The motivation behind such a separation of concerns is two-fold. First, fault-tolerance is added only for dealing with faults, and the added fault-tolerance concerns should not conflict with functional requirements in the absence of faults. Second, fault-tolerance requirements evolve as we encounter new types of faults. Thus, there exist two options: either (1) develop from scratch a system that meets its functional requirements in the absence of f and provide desired functionalities in the presence of f, or (2) incrementally add new fault-tolerance functionalities while preserving the existing functionalities in the absence of f. In this paper, we adopt the latter as it seems to be less expensive than the former approach and better handles legacy systems.

3 Modeling

In this section, we present the basic concepts of modeling FISs, faults, and nonmasking fault-tolerance in UML. The motivation behind using UML is two-fold. First, UML is a modeling language well-accepted in both academia and industry. Second, since our model is based on the notion of finite state machines, UML state diagrams enable us to capture any form of recovery that can be expressed in a state machine-based formalism.

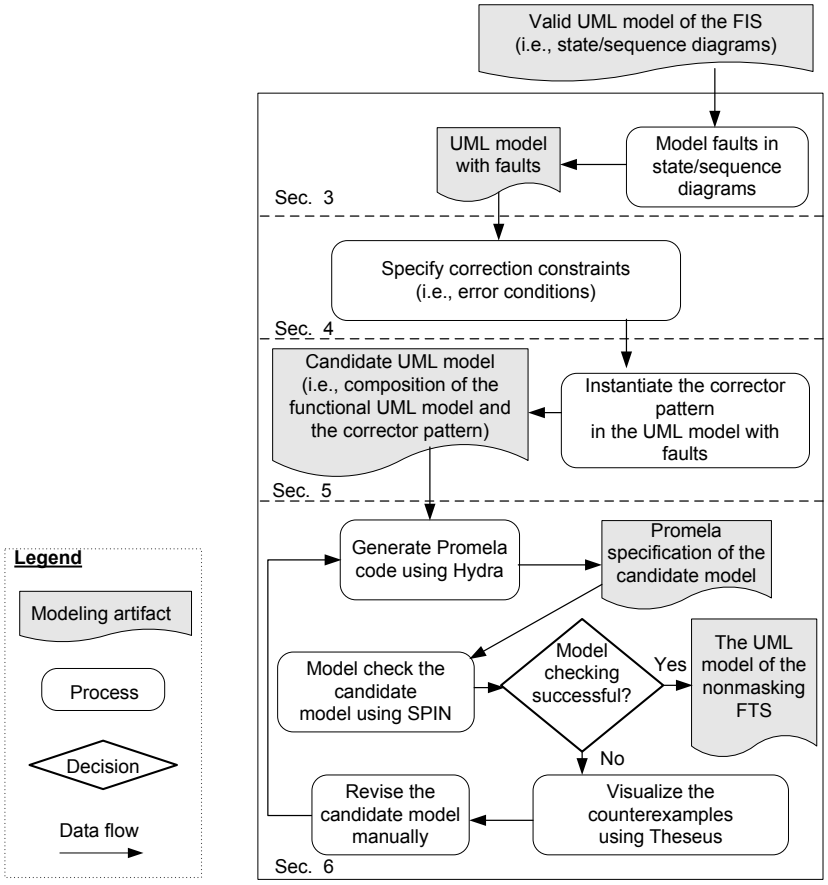

Fig. 1. An overview of the proposed approach.

3.1 UML Models

We use conventional UML notations [25] to represent the UML-based conceptual models of FISs (respectively, FTSs). Since our focus is on modeling and analyzing nonmasking fault-tolerant embedded systems, we follow Douglass [26] in using state/sequence diagrams to capture high-level *behavioral* information of UML object models. In a UML object model M with n objects $O_1, \cdots, O_n$, 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 $(1 \leq i \leq n)$. A state of an object O_i is a valuation of its state variables (i.e., attributes). A transition is of the form $(a, evt[grd]/act, b)$, where a and b are states, evt denotes a triggering event, grd represents a guard condition and act denotes an action that should be taken when a transition from a to b takes place. A *global state predicate* is defined over a set of states of multiple objects. A *local state predicate* is specified over the set of states of only one object O_i (i.e., S_i). A *scenario* is a sequence of states $\langle s_0, s_1, \cdots \rangle$, where each s_i is a state of some object O_j $(1 \leq j \leq n)$. We use UML sequence diagrams to represent

scenarios. A *behavior* of an object O_j $(1 \leq j \leq n)$ is a scenario $\langle s_0, s_1, \cdots \rangle$ such that $\forall s_i : i \geq 0 : s_i \in S_j$.

Underlying Computational Model. Depending on the semantics of object interactions, the complexity of automatic analysis of an FTS varies from polynomial (in a shared memory model [27]) to undecidable (in an asynchronous message-passing model [28]). For example, in our previous work [29–31], we have shown that automated analysis of fault-tolerance for models of distributed systems has an exponential complexity (in the size of the model). 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. In cases such atomicity assumptions do not hold (e.g., distributed systems), one can use our proposed approach for developing a high atomicity conceptual model of the fault-tolerant system, and then use existing tolerance-preserving refinement techniques (e.g., [32]) to generate a refined model. (Such refined models can in turn be realized in the implementation phase using existing mechanisms such as coordinated atomic actions [33].) More importantly, if the inconsistencies of fault-tolerance and functional concerns cannot be resolved in a high atomicity model (with atomic actions), then deriving a refined (concrete) model of a fault-tolerant system from the conceptual model of its fault-intolerant version would be impossible [30].

Modeling Functional Requirements. In order to model functional requirements, we extend Gouda and Arora's [34] notion of *closure*, where a fault-tolerant system remains in a set of legitimate states as long as no faults have occurred. A set of legitimate states (also called an *invariant*) can be computed by identifying the set of states that are reachable by system actions from a given set of initial states. (Techniques of how to extract an invariant from user requirements are beyond the scope of this paper. Examples can be found in [35,36].) In the invariant, no system action violates its *safety requirements* (i.e., the system takes no bad actions (defined by user requirements)) and *liveness requirements* are satisfied. Intuitively, safety requirements stipulates that nothing bad ever happens and the liveness requirements specify that something good will eventually occur. 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 actions, say $\mathcal{B}$, that must not occur in the behaviors of any object. Since we start with a functional model of an FIS system that meets its safety and liveness requirements in the absence of faults (i.e., when no faults occur) and incrementally model fault-tolerance, we do not explicitly specify liveness requirements. Nonetheless, we require that while modeling fault-tolerance concerns, no deadlock states (states with no outgoing transitions) should be introduced in the invariant. The deadlock freedom requirement captures the fact that, in the absence of faults, embedded systems have non-terminating computations and always react to their environment. We say a scenario $\langle s_0, s_1, \cdots \rangle$ *meets safety*

requirements iff (if and only if) $\forall i : i \geq 0 : (s_i, s_{i+1}) \notin \mathcal{B}$. A scenario $\langle s_0, s_1, \cdots \rangle$ *meets liveness requirements* iff every state s_i, for $i \geq 0$, has a next state and some desired conditions eventually become true in that scenario. Thus, an invariant has two properties: (1) starting from any state in the invariant, the subsequent states are also in the invariant (i.e., *closure*), and (2) from every state in the invariant, all scenarios meet safety and liveness requirements. A UML model M *meets its functional requirements* iff there exists a non-empty invariant $\mathcal{I}$ for M. A *functional scenario* is a scenario whose states all belong to the invariant. A *recovery scenario* $\langle s_0, s_1, \cdots \rangle$ satisfies the following condition: $\exists i : i \geq 0 : s_i$ is in the invariant.

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 (i.e., *target vehicle*) for collision avoidance. The ACC system has different modes of operation (see Figure 2), namely *closing*, *coasting*, *matching*, *alarm*, *disengaged* or *resume* mode. When the radar detects a target vehicle, i.e., *target* mode, 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* mode. 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. In cases where the speed of the car is so fast (greater than a maximum speed v_{max}) that a collision is unavoidable, the ACC system must raise an alarm for the driver, i.e., the *alarm* mode, and must deactivate the cruise control system, i.e., the *disengaged* mode. When the radar loses the target vehicle and the cruise control system is active, the system is in the *resume* mode.

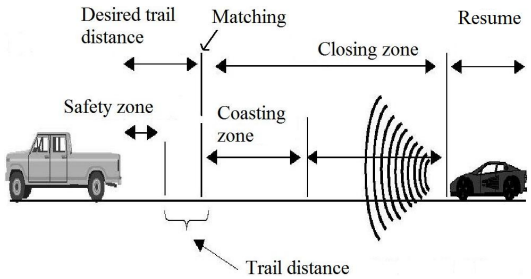

Fig. 2. The adaptive cruise control system.

The ACC system comprises three main classes, namely Control, Car, and Radar (see Figure 3). (We use Sans Serif font to denote state variables, methods and classes.) (i) The Control class has a set of Boolean state variables that represent

different modes of the ACC system. The Brakes state variable is set when the control receives a signal from the brakes subsystem indicating that the brakes have been applied. The ACC system must be disengaged when the Control receives a Brakes signal. The method setpUpdate() updates the *setpoint*, which is the desired speed determined by the driver. The Radar sets Control.target to true by invoking the targetDet() method. Depending on the computed trail distance, the Control object also calculates the safetyZone such that the distance to the target vehicle never becomes the safetyZone value. (ii) The Car class models the engine management functionalities (e.g., acceleration, deceleration). A Car object matches the real speed of the car (denoted by realv) with the setpoint (using the method matchSpeed()). The car calculates its real speed using the data received from the speed sensors located in the car. The getRealV() method may be invoked by the Control to receive the real speed of the car. (iii) The Radar measures the distance of the car to the target vehicle, kept in the state variable currDist, which is also used by the Control (by invoking Radar.getDistance()).The Radar also measures the speed of the target vehicle (kept in Radar.targetSpeed) that can be accessed using the Radar.getTargetSpeed() method.

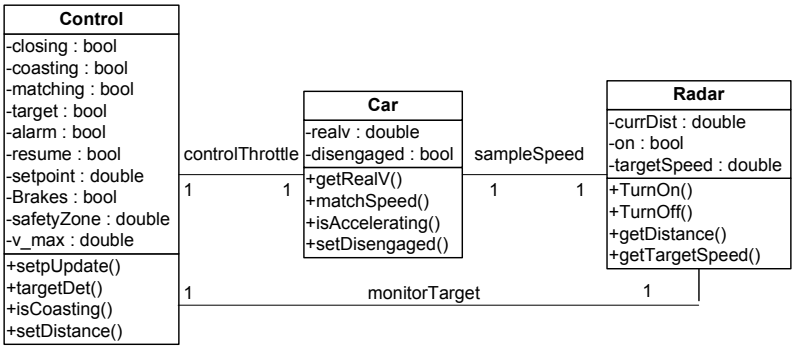

Fig. 3. Excerpted class diagram of the ACC system.

An invariant of the ACC system (denoted $\mathcal{I}_{ACC}$) is a global state predicate that specifies a set of states, in which (i) if a target vehicle has been detected then the ACC system is in one of the following modes: *closing*, *coasting*, *matching*, *alarm*, or *disengaged*; (ii) the distance with the target vehicle is 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 the maximum speed v_{max}, then the ACC system must alarm the driver of a potential collision and must disengage. Therefore, the invariant $\mathcal{I}_{ACC}$ is equal to the following set of states:

$$\{s : (target(s) \Rightarrow (closing(s) \vee coasting(s) \vee matching(s) \vee disengaged(s))) \wedge$$
$$(safeyZone(s) < currDist(s)) \ \wedge ((\neg target(s) \wedge cruise(s)) \Rightarrow resume(s)) \wedge$$
$$(Brakes(s) \Rightarrow disengaged(s)) \wedge$$
$$(realv(s) > v_{max}(s) \Rightarrow (alarm(s) \wedge disengaged(s)))\}$$

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

Note that the above predicate is specified in terms of the variables of all three objects. The variables target, closing, coasting, matching, resume, alarm, Brakes, v_{max} and safetyZone belong to the Control object. disengaged and realv are state variables of the Car object, and currDist is in the Radar object.

3.2 Modeling Faults in UML

In this section, we illustrate how to model faults in UML state and sequence diagrams in the context of the ACC system. Since our focus is on the behavioral object models, we omit the fault modeling at the class diagram level (see [23] for details).

Modeling Faults in State Diagrams. We systematically model a fault-type as *a set of transitions* in UML state diagrams (see Figure 4). Representing faults as a set of transitions has already appeared in previous work [11, 16], and it is known that state perturbation is sufficiently expressive to represent different types of faults (e.g., crash, input-corruption, Byzantine) from different behavioral categories (e.g., transient, intermittent, permanent) [11, 16]. Moreover, we assume that faults stop occurring in a finite amount of time so that eventually recovery can occur [16, 37]. Depending on the occurrence of faults, we classify faults into two categories 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. An *arbitrary* fault-type has no precondition and may occur at any state (e.g., environmental noise). Given a conditional fault-type f, 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 Figure 4). We denote the set of *transitions of* SD_i *in the presence of faults* f_i by $\delta_i \cup f_i$. We model an arbitrary fault-type as a separate fault state transition diagram (e.g., FD_1 in Figure 5) that executes concurrently with an object state machine (e.g., SD_1 in Figure 5). In Figure 5, the transitions of the arbitrary faults in FD_1 may trigger at any state of the state diagram SD_1. The key difference is in the semantics of fault transitions in that an object O_i does not have control over the execution of faults f_i (see dashed arrows in Figure 4), whereas the execution of regular transitions (see solid arrows in Figure 4) is controlled by the thread of execution in O_i.

When modeling a fault type f_i in a state diagram SD_i, modelers should identify the scope of the states reachable by a combination of fault and regular transitions, which is called the *fault-span* of O_i for fault f_i (denoted f_i-span of O_i) [27]. For example, in Figure 4, all error states are only reachable when faults occur. Thus, introducing faults in a state diagram may require new states and transitions to be added to that state diagram. In fact, given a UML model M, its invariant $\mathcal{I}$ and a fault-type f, starting from $\mathcal{I}$, the set of states reachable by a combination of fault and system transitions comprises the *global fault-span* of M, denoted f-span of M. More precisely, the f-span of M has two properties: (1) the f-span of M contains $\mathcal{I}$, and (2) starting from every state in the f-span of M, any fault or system action will result in another state in the f-span; i.e., *closure*

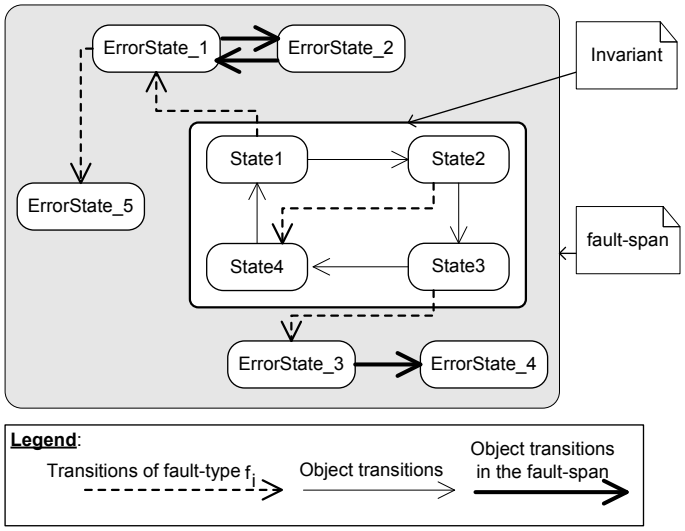

Fig. 4. Modeling conditional faults in UML state diagrams.

of the f-span of M in the set of system and fault actions. In Figure 5, the fault-span is identified by calculating the asynchronous automata-theoretic product of the two state machines SD_1 and FD_1, which results in a new state diagram that simultaneously includes fault and regular transitions. A behavior of an object that originates in its fault-span outside its invariant may lead to failures (i.e., violate safety requirements, fall into non-progress cycles, or reach a deadlock state). For example, in Figure 4, 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.

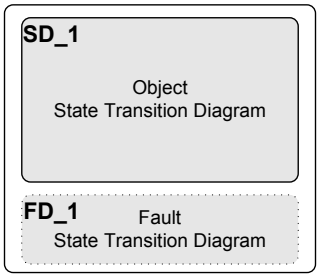

Fig. 5. Modeling arbitrary faults.

ACC Example: The ACC system is subject to an arbitrary fault-type f_{ACC} that may non-deterministically set the disengaged signal in the Car *object to false. Hence, we model the effect of f_{ACC} on the* Car *object as a state machine concurrent with the car state machine (see Figure 6). In this case, faults f_{ACC}*

may set the value of the Car.disengaged *variable to false at any state of the* Car *state diagram. Note that in the f_{ACC} state machine in Figure 6, once the fault transition from* State1 *to* State2 *sets* Car.disengaged *to false,* Car.disengaged *remains false until a system recovery action resets it back to true.*

Modeling Faults in Sequence Diagrams. In UML sequence diagrams, we model the effect of a fault-type f_i on an object O_i as a self message to O_i that may occur non-deterministically (see Figure 7). Such a representation of faults in sequence diagrams is based on how faults are modeled in the state diagram of O_i. Thus, modeling faults in SD_i affects all sequence diagrams in which O_i is involved. Such sequence diagrams represent *scenarios with faults*. Formally, a scenario with fault f is a sequence of states $\langle s_0, s_1, \cdots \rangle$, where $\forall s_i, s_{i+1} : i \geq 0 :$ $(((s_i, s_{i+1}) \in f) \ \vee \ ((s_i, s_{i+1})$ belongs to some object $O_j))$, for $1 \leq j \leq n$. To identify scenarios with faults, modelers should update every scenario in which O_i is involved, and should discover new scenarios that take place due to the occurrence of faults. The identification of scenarios with faults is important in modeling the system behaviors in the presence of faults.

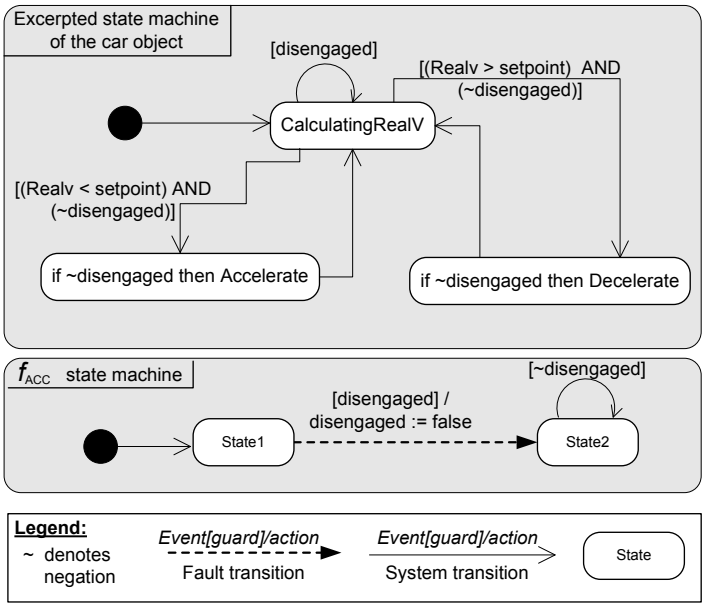

Fig. 6. Modeling faults in the state transition diagram of the car.

ACC Example: In Figure 7, once the Control *detects a* Brakes *signal, it invokes* Car.setDisengaged() *illustrating that the engine controller should be deactivated (i.e., disengaged). However, if faults f_{ACC} occur, the* disengaged *flag will be reset to false, which in turn results in the reactivation of the engine controller, possibly resulting in acceleration while brakes are applied. This is a scenario with faults that must be corrected.*

UML models with faults. Modeling a fault-type f in the state and sequence diagrams of a UML model M creates a UML object model M_f that has been augmented with fault f. We call M_f a UML *model with faults* f.

Comment on the complexity of modeling faults, fault-span and scenarios with faults. The proposed modeling approach in this section includes three main tasks, namely modeling (conditional or arbitrary) faults in state diagrams, modeling fault-spans in state diagrams and modeling scenarios with faults. The fault modeling task should be done manually and the other two tasks can be automated. While modeling (conditional and arbitrary) fault transitions in state diagrams may seem to be a tedious task for large systems, we argue that (1) the scale of such a modeling activity does not go beyond the complexity of modeling regular transitions in the state diagrams of all functional objects, and (2) techniques that facilitate the modeling of regular transitions can directly be reused to facilitate fault modeling. In cases where more than one type of faults should be modeled, UML extension techniques (e.g., stereotyping) would help developers to distinguish the transitions of different fault-types and their corresponding fault-spans. Since it is difficult to manually identify all scenarios with faults (respectively, model the fault-span of an object) and the number of such scenarios may increase exponentially, we are currently investigating the integration of a software tool we have previously developed [38,39], called Fault-Tolerance Synthesizer (FTSyn), in UML as FTSyn automatically generates fault-span and scenarios with faults from state diagrams.

3.3 Modeling Nonmasking Fault-Tolerance

In this section, we extend the definition of nonmasking fault-tolerance from Arora [11] in the context of UML models. Intuitively, *nonmasking* fault-tolerance requires recovery to the invariant after faults stop occurring [11]. More precisely, let $\mathcal{S}$ be an FIS, $\mathcal{I}$ be an invariant of S, and f be a given fault-type perturbing the state of $\mathcal{S}$. (Note that the FIS $\mathcal{S}$ provides no guarantees about its behavior when f occurs.) A system $\mathcal{S}'$ is a nonmasking f-tolerant (i.e., nonmasking fault-tolerant against f) version of $\mathcal{S}$ if and only if the following conditions are satisfied: (1) in the absence of f, the FTS $\mathcal{S}'$ meets the functional requirements specified for $\mathcal{S}$, and (2) in the presence of f, the FTS $\mathcal{S}'$ guarantees recovery to $\mathcal{I}$.

Before defining what we mean by a nonmasking fault-tolerant UML model, we define *recovery scenarios*. Let M be a UML model of $\mathcal{S}$ and $\mathcal{I}$ be an invariant of $\mathcal{S}$ defined in M. We say a scenario $\sigma = \langle s_0, s_1, \cdots \rangle$ in M *recovers to the invariant* $\mathcal{I}$ iff $\exists i : i \geq 0 : (s_i \in \mathcal{I})$. Note that once a state in the invariant is reached, the closure property guarantees that the system remains in the invariant as long as there are no faults. We say a scenario σ in the UML model M *violates recovery requirements* iff σ does not recover to the invariant of M. Violation scenarios could take place if a deadlock state or a non-progress cycle is reached due to the occurrence of f. We say a UML model M *recovers to the invariant* $\mathcal{I}$ iff all scenarios of M recover to $\mathcal{I}$. Accordingly, a UML model M *violates recovery requirements* iff there exists a scenario that violates recovery requirements. We say a UML model M' (derived from M) is nonmasking f-tolerant if M' satisfies the following conditions: (1) the set of functional scenarios of M' is a non-empty

subset of the set of functional scenarios of M starting in a subset of $\mathcal{I}$, and (2) all scenarios with fault f recover to $\mathcal{I}$.

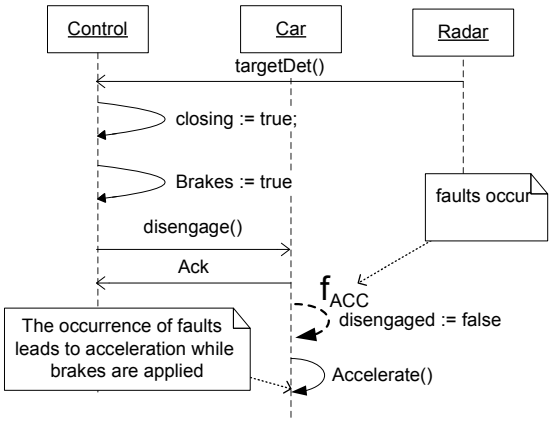

Fig. 7. Coasting scenario in the presence of faults f_{ACC}.

4 Specifying Error Conditions

In order to model recovery, we need to specify the set of error states from where recovery should be provided. The notion of invariant simplifies the task of specifying error states as it characterizes the set of states from where functional requirements are guaranteed to be met in the absence of faults. The occurrence of faults may falsify the invariant, thereby reaching states from where failures may occur. Thus, for a given UML model M and its invariant $\mathcal{I}$, the weakest set of error states is $\neg\mathcal{I}$. However, some states in $\neg\mathcal{I}$ may be unreachable by either system or fault actions. In fact, for a specific fault-type f, the set of reachable error states is equal to the intersection of $\neg\mathcal{I}$ and the f-span of M. In other words, the set of reachable error states is equal to $\mathcal{FS} - \mathcal{I}$, where $\mathcal{FS}$ denotes the f-span of M. A fault-intolerant system may stay in $\mathcal{FS} - \mathcal{I}$ forever for two reasons: reaching a deadlock state or falling in a cycle whose states all belong to $\mathcal{FS} - \mathcal{I}$ (called a *non-progress* cycle).

In order to ensure recovery, we have to resolve deadlock states and non-progress cycles. For programs whose processes can read and write all program variables in an atomic step, resolving deadlock states amounts to the addition of actions that establish the truth value of $\mathcal{I}$ once it is falsified. Such actions are called *convergence* actions [34] as they guarantee the convergence of system behaviors to its invariant. Likewise, non-progress cycles can be resolved by breaking cycles and adding convergence actions. In concurrent and distributed programs, resolving deadlock states and non-progress cycles is a non-trivial task. To illustrate the complexity of providing recovery, consider a distributed program with two processes and an invariant $((x - y) = c) \wedge (y \geq z)$, where x, y, and z are program variables and c is a constant. In an error condition where the invariant does not hold, a process that cannot read z may decrease the value of y to establish

the equality ($x - y = c$) for the sake of recovery. This recovery action may potentially violate the second conjunct of the invariant due to decreasing the value of y. In such cases, convergence should be provided in a coordinated fashion. In the above example, if y is left unchanged and each process is allowed to modify only one of the variables x and z, then coordinated recovery is achievable. In the next section, we present the corrector pattern, which facilitates modeling and analysis of the recovery of concurrent and distributed programs and provides measures to verify the correctness of such recovery.

5 Corrector Pattern

In this section, we introduce a template for the corrector pattern that we use for modeling and analyzing nonmasking fault-tolerance. We have also developed a corresponding detector pattern [23] to specify error detection, but due to space constraints, we do not include it here. While design patterns are traditionally classified in terms of structural, behavioral, and creational patterns [17], the corrector pattern provides a reusable strategy (for decomposing error conditions) that can be refined to different design mechanisms. In order to facilitate its use, we define a template for the corrector pattern based on the fields used in the design patterns presented by Gamma *et al.* [17], 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 corrector pattern represented by UML class diagrams. The corrector pattern also includes several new fields that are added for the purpose of specifying and analyzing fault-tolerance concerns. For example, the corrector pattern includes the *Correction Requirements* field that specifies a set of requirements that must be met by the corrector pattern to ensure that the corrector pattern is itself nonmasking fault-tolerant. We also introduce the *Interference-Freedom Constraints* field to guarantee that the correction occurs correctly. We use the ACC system to demonstrate how to use the corrector pattern to add nonmasking f_{ACC}-tolerance to the ACC system. Next, we describe the fields of the corrector pattern.

Intent. The corrector pattern formulates the problem of *correcting error conditions*. More specifically, the corrector pattern captures the recurring problem of restoring the state of a computing system from one state predicate to another (e.g., from outside an invariant to the invariant).

Correction Predicate. A *correction predicate*, say X, is a condition whose truth value should be established (e.g., invariant). In a UML model M, a correction predicate is a state predicate that could be either local or global. In a distributed system, it is difficult for an object to atomically correct a global correction predicate X in an atomic step [40]. Thus, it is desirable to decompose X into a set of local predicates $X_1, \cdots, X_n$, and to specify the correction of X based on the correction of $X_1, \cdots, X_n$, where each X_i ($1 \leq i \leq n$) represents the local state of a system component. In the specification of global error conditions we often encounter predicates representing deadlock states and non-progress cycles. These error conditions often have a conjunctive form. Since a correction predicate is the negation of an error condition, in this paper, we limit the scope of the application of the corrector pattern to the correction of conjunctive error

predicates. (Conjunctive predicates comprise an important class of predicates in distributed systems [41].)

ACC Example: The occurrence of f_{ACC} may perturb the ACC system to a state s, where the cruise control system is engaged in engine management even though brakes have been applied. This introduces a deadlock state as long as the brakes are applied. We represent this error condition by the conjunctive predicate $(X_{control} \wedge \neg X_{car})$ that should be corrected. The correction predicate X_{ACC} is the negation of the above error condition, i.e., $X_{ACC} \equiv \neg(X_{control} \wedge \neg X_{car}) \equiv (X_{control} \Rightarrow X_{car})$, where $X_{control} \equiv$ Control.Brakes and $X_{car} \equiv$ Car.disengaged. Note that if X_{ACC} is false (i.e., error has occurred), then the invariant $\mathcal{I}_{ACC}$ (specified in Section 3) is violated. To provide nonmasking f_{ACC}-tolerance, we must ensure that the condition X_{ACC} will eventually hold after f_{ACC} stops occurring. Moreover, in the context of the ACC example, there is only one way to correct X_{ACC}; it is by setting the state variable Car.disengaged *to true (because the state variable* Control.Brakes *represents a signal input to the ACC system and cannot be changed).*

Corrector Elements (Participants). We use corrector elements c_i, $1 \leq i \leq n$, such that each c_i is responsible for correcting X_i. Each corrector element [4] c_i, for $1 \leq i \leq n$, is indeed a participant of the corrector pattern and has its own correction predicate X_i.

Distinguished Element. An element c_{index} $(1 \leq index \leq n)$ that establishes the correction of X based on the correction of $X_1, \cdots, X_n$ is called the *distinguished element*. That is, the distinguished element finalizes the correction of a global predicate based on the correction of its participants.

Structure. We present two basic structures for the corrector pattern: *sequential* and *parallel*. The correction of X can be done either (i) sequentially, where participants c_i, for $1 \leq i \leq n$, correct their correction predicates X_i one after another, or (ii) in parallel, where all elements c_i, $1 \leq i \leq n$, correct their correction predicates concurrently. For example, if the inter-object associations in a UML object model form a linear (respectively, hierarchical) structure then a sequential (respectively, parallel) corrector is more appropriate. We illustrate the structure of the sequential corrector pattern in Figure 8. The shadowed objects represent the elements of the corrector pattern encapsulated in a dashed box that denotes an instance of the corrector pattern. The distinguished element of the corrector pattern is depicted by the dark shading.

In Figure 8, each corrector participant c_i is associated with an object $object_i$ in which the predicate X_i $(1 \leq i \leq n)$ should be corrected. (Note that $object_i$ may be associated with more than one corrector element, each belonging to a different instance of the corrector pattern.) The distinguished element is associated with the participant c_n, which establishes the correction of X_n and X. Figure 9

[4] In the design and implementation phases, the corrector 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 corrector elements would not be worse than the use of conventional redundancy mechanisms.

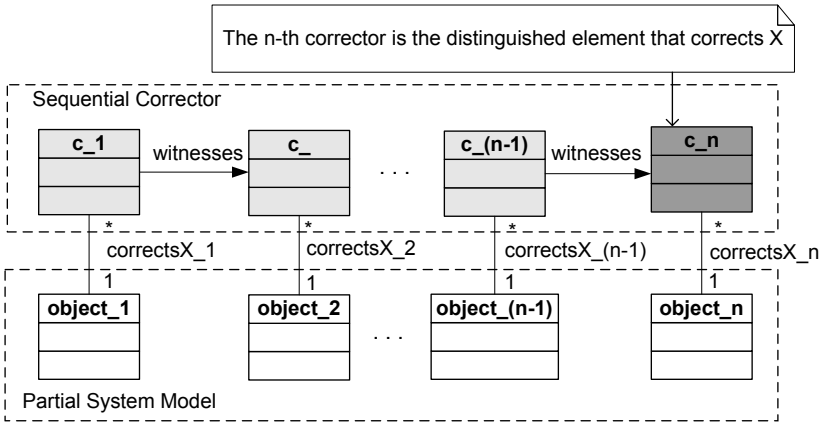

Fig. 8. The structure of the sequential corrector.

illustrates the application of an instance of the sequential corrector to the UML model of the ACC system. The inter-connection of the functional objects in the functional model of the FIS system plays a factor as to which corrector pattern should be applied (sequential, parallel or a combination of both). For example, if the associations between the functional objects constitute a linear structure, then a sequential corrector is more appropriate, whereas for the case of hierarchical associations between functional objects (e.g., tree-like associations), a parallel corrector would be instantiated. Moreover, a combination of sequential and parallel correctors may also be used for the correction of a predicate. Due to space constraints, we omit the presentation of such combinations and the parallel corrector (see [23] for details).

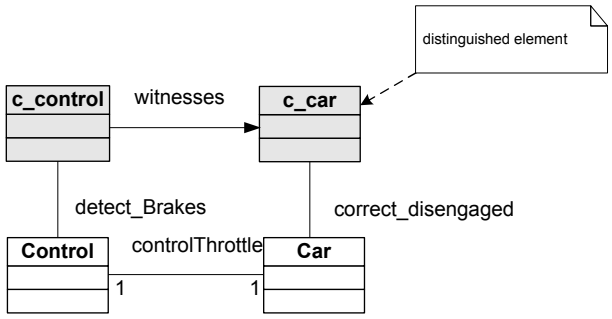

Fig. 9. Composition of a sequential corrector pattern with the ACC system.

Witness Predicate. Since we decompose the global correction predicate X into a set of local correction 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 of the corrector pattern. The truth value of the witness predicate is an indication that X has been corrected. We also consider a witness predicate Z_i

for each element c_i to represent that c_i corrects X_i. We say c_i *witnesses* iff Z_i is *true*. In the case of the sequential corrector, the distinguished element c_{index} (i.e., c_n) sets the value of Z_{index} (i.e., Z_n) to *true* if $c_1, \cdots, c_{n-1}$ witness their correction predicates and X_n holds.

Invariant. The invariant of the correction pattern is a state predicate $\mathcal{I}_\mathcal{C}$ such that $\mathcal{I}_\mathcal{C} = \{s : Z_i(s) \Rightarrow (\forall j : 1 \leq j < i : Z_j(s))\}$. Intuitively, it means that, in an invariant state s, if a corrector element c_i witnesses, then all its predecessors should also witness.

Correction Requirements. In order to ensure the recovery of the composition of an instance of the corrector pattern with the UML model of an FIS, the corrector pattern and its participants should meet the following requirements (adapted from [42]): (1) *Safeness.* It is never the case that the witness predicate Z is *true* when the correction predicate X is *false*; i.e., the corrector pattern never lies. (2) *Progress.* It is always the case that if X becomes 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 the predicate X is *true* (i.e., Z remains *stable*). (4) *Convergence.* The correction predicate X will eventually hold and will continuously remain true. Each participant c_i should also meet the above requirements for Z_i and X_i. The first three requirements (safeness, stability, and progress) specify the requirements for the detection of the predicate X while the convergence states that X will eventually hold. A pattern that only meets the safeness, progress and stability requirements guarantees to detect the correction predicate X if it ever holds, but does not guarantee to establish X if it is falsified. In special instances of the corrector pattern, we may have some participants c_i that perform only as a detector.

The correction requirements can be specified in Linear Temporal Logic (LTL) [43] using (i) the universal operator $\Box$, where $\Box Y$ means that the state predicate Y always holds; (ii) the next state operator $\bigcirc$, where $\bigcirc Y$ means that in the next state Y holds, and (iii) the eventuality operator $\Diamond$, where $\Diamond Y$ means that the state predicate Y eventually holds. We respectively specify *safeness* and *stability* as $\Box(Z \Rightarrow X)$ and $\Box(Z \Rightarrow (\bigcirc(Z \vee \neg X)))$. We specify *progress* as the following LTL expression: $\Box(X \Rightarrow \Diamond Z)$ and the LTL formula $\Diamond(\Box X)$ specifies the convergence requirement. Note that the correction requirements can also be specified using Dwyer *et al.* [44] specification patterns. For example, the safeness and stability can be represented in terms of the Universality specification pattern defined by Dwyer *et al.* [44].

ACC Example: The instance of the corrector pattern applied to the ACC system comprises two elements $c_{control}$ and c_{car} modeled as two new objects in the UML model of the ACC system (see Figure 9). The element $c_{control}$ behaves as a detector that only monitors the state of the Control.Brakes *signal (i.e., detects $X_{control}$) and the element c_{car} should correct X_{car} when it is false; i.e., when* Car.disengaged *is false, it should be set to true. The element $c_{control}$ sets its witness predicate $Z_{control}$ to true when $X_{control}$ holds; i.e., when brakes are applied. The element c_{car} sets its witness predicate Z_{car} to true when $X_{control}$ and X_{car} hold. The invariant of the corrector pattern is equal to $Z_{car} \Rightarrow Z_{control}$. More specifi-*

cally, c_{car} continuously checks with $c_{control}$ to see whether brakes are applied or not. If $c_{control}$ witnesses, then c_{car} corrects its correction predicate (i.e., $X_{car} \equiv$ Car.disengaged*) if necessary. Such a correction is established by a local corrective action that sets the state variable* Car.disengaged *and the witness predicate Z_{car} to true.*

Behavior. Figure 10 depicts a strategy for correcting a predicate $X \equiv (X_1 \wedge X_2 \wedge \cdots \wedge X_n)$ in a sequential fashion. The distinguished element can witness if all its predecessors $c_1, \cdots, c_{n-1}$ have already witnessed their correction predicates. In other words, if Z holds then $Z_1 \wedge \cdots \wedge Z_n$ must hold as well. Note that, for simplicity, there are no timing requirements in Figure 10, and all participants execute asynchronously. Notice that, for nonmasking fault-tolerance, we do not explicitly impose any order on the recovery of corrector elements as long as progress and convergence requirements are satisfied and recovery to the invariant is guaranteed. Nonetheless, depending on the problem at hand, satisfying the above requirements may require us to impose a specific recovery order in that which element should recover first. For example, in a token ring protocol, the direction of token circulation should be consistent with order of recovering elements.

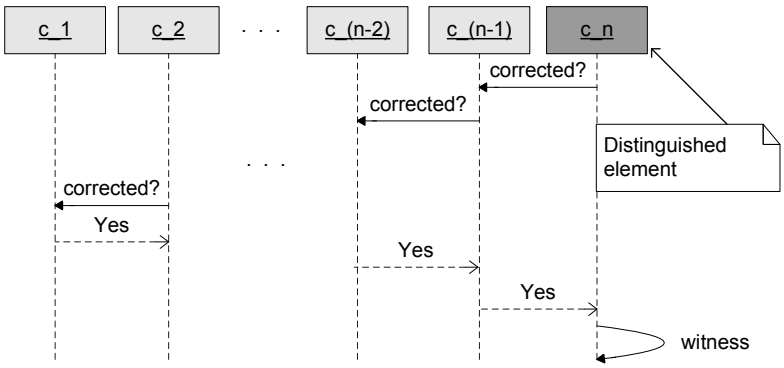

Fig. 10. The behavior of a sequential corrector.

ACC Example: In Figure 11, the element $c_{control}$ continuously checks the state of the Control *object to determine whether or not the brakes have been applied. The c_{car} corrector element also monitors the state of the* Car *object to check whether or not the cruise control system is disengaged. Before the distinguished element c_{car} witnesses (i.e., sets Z_{car} to true), it checks the truth value of $Z_{control}$. If $Z_{control}$ is true and X_{car} is false (i.e.,* Car.disengaged *= false), then c_{car} establishes its correction predicate (i.e., sets* Car.disengaged *to true) and witnesses.*

Consequences (Interference). We say the *corrector pattern interferes with the UML model of the FIS* iff in the absence of faults the candidate model violates the safety requirements of the FIS or causes the FIS to deadlock (i.e.,

violates the functional requirements of the FIS). We also say the *UML model interferes with the corrector pattern* iff in the presence of faults the candidate model violates safeness, stability, progress or the convergence of the corrector pattern. Intuitively, since we use the corrector pattern to correct the invariant of a system, an instance of the corrector pattern will execute only when the invariant of the system does not hold. Thus, as long as the system is functioning in its invariant the corrector pattern will not execute any actions, thereby preventing any interference. However, once the state of the system is perturbed outside its invariant, we may have a concurrent execution of the system actions and the actions of the corrector pattern. Since we are only dealing with deadlocks and non-progress cycles, we have two cases to investigate. If the system reaches a deadlock state outside its invariant, then only the corrector elements will be active, thereby no interference will occur (because the FIS is deadlocked) and the corrector pattern guarantees to reestablish the system invariant. In the case of non-progress cycles, an interference-free composition of the functional model and the corrector pattern guarantees the progress and convergence of the corrector pattern, which in turn leads to breaking the cycle by a gradual reestablishment of the invariant predicate.

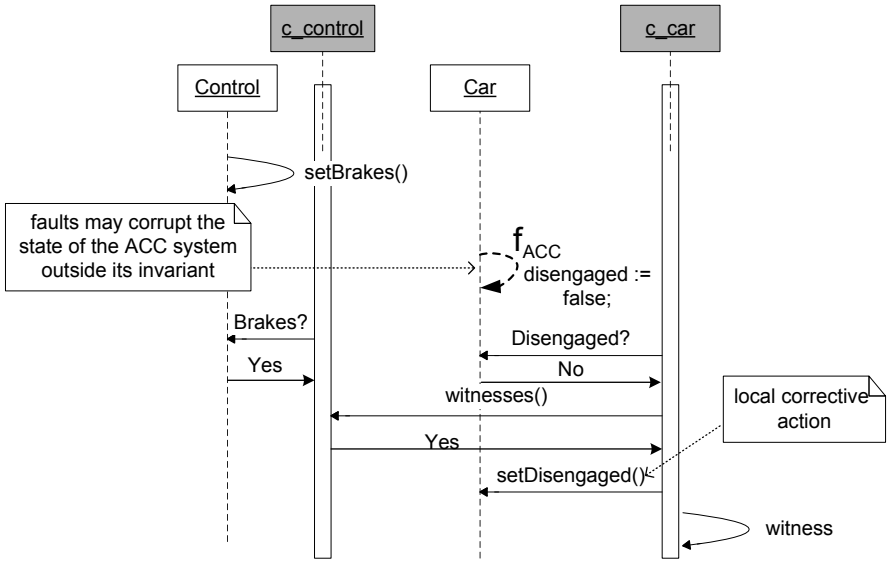

Fig. 11. The behavior of the sequential corrector applied to the ACC system.

Nonmasking fault-tolerance of the corrector pattern. Since the instances of the corrector pattern are also subject to faults, we must ensure that the corrector pattern is itself nonmasking fault-tolerant to *the effect of faults*. To guarantee the nonmasking fault-tolerance of the corrector pattern, the progress and convergence of the corrector pattern should be met. This is because when faults occur the only requirement for the corrector pattern is to eventually recover to its invariant $\mathcal{I}_{\mathcal{C}}$. For example, if faults occur after some

corrector element c_i $(i > 1)$ witnesses, then the witness predicate of some c_j, for $1 \leq j < i$, may be falsified due to the effect of faults. As a result, the invariant $Z_i \Rightarrow (\forall j : 1 \leq j < i : Z_j)$ will no longer hold. However, since X_j holds (notice that Z_j was set to true because X_j had become true at some point), after faults stop occurring, the progress property of the element c_j guarantees that Z_j will again become true, thereby resulting in the recovery of the entire corrector pattern to its invariant $\mathcal{I}_{\mathcal{C}}$. In another scenario, the effect of faults may cause Z_i to become true while none of its predecessors has witnessed, thereby violating the invariant predicate $\mathcal{I}_{\mathcal{C}}$. (In this case, faults directly violate the safety of the corrector pattern, which is not a concern since recovery to the important is the only requirement.) Since the convergence requirement guarantees that all predecessors of c_i will eventually witness, the invariant $\mathcal{I}_{\mathcal{C}}$ will eventually be established. Therefore, a design of the corrector pattern that meets the progress and convergence requirements is itself nonmasking fault-tolerant to the effect of faults.

ACC Example: The effect of f_{ACC} faults on the corrector pattern applied to the ACC system is that faults may corrupt the value of the witness predicates $Z_{control}$ and Z_{car} to false. The progress of the corrector element $c_{control}$ guarantees that the corrector pattern will recover to its invariant $Z_{car} \Rightarrow Z_{control}$ if Z_{car} holds and $Z_{control}$ has been falsified by faults.

Remark. In this section, we only considered the application of an instance of the corrector pattern for the ACC system. While the ACC example is small, the number of the elements of an instance of the corrector pattern cannot go beyond the number of system components. Moreover, for nonmasking fault-tolerance, it is often the case that only one instance of the corrector pattern should be instantiated to model the correction of the violations of system invariant. As a result, at most one corrector element will be composed with each system component, which does not hinder the scalability of our approach. Moreover, even though composing a corrector pattern with the functional model of an FIS system may add a layer of complexity, the modularity provided by the corrector pattern facilitates the management of such complexity. (Besides, other pattern-driven methods may also suffer from this additional layer of complexity introduced by pattern instantiation.)

6 Generating Promela Code and Automated Analysis

In order to enable rigorous analysis of modeling artifacts in model-driven development of fault-tolerant systems, we generate formal specifications of UML models and use model checkers for detecting the inconsistencies between fault-tolerance and functional requirements. Towards this end, we extend the Hydra UML formalization framework [20] to generate the formal specification of the UML models of FTSs in the Promela modeling language [21]. Hydra [20] is a generic framework for generating formal specifications from UML diagrams. Promela is a language for modeling concurrent and distributed programs in the model checker SPIN [21]. The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (con-

current) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications.

Hydra uses a set of mapping rules (see Figure 12) to translate the entities in a UML metamodel to the entities in a Promela metamodel. For example, Hydra translates each UML object to a *proctype* in Promela. Thus, each element c_i of the corrector pattern will be formalized as a separate process that is concurrently executed with the processes that represent UML functional objects. The transitions of the state diagram of each object are formalized as the atomic actions of the corresponding process in Promela. The inter-object associations at the UML level are formalized as message exchange channels in Promela.

Extending Hydra for fault formalization. In UML state diagrams, we distinguish fault transitions from regular transitions by defining a Fault stereotype [25]. The extended Hydra treats the transitions of a fault-type f differently than other transitions in that it integrates the transitions of f (modeled in different state diagrams) in a separate process Fault_f in Promela that is concurrently executed 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.

UML Metamodel Entity		*Promela Metamodel Entity*
Object	→	proctype
Instance variable	→	Variable
Association	→	Channel
Generalization	→	Duplicated proctype
State	→	State block
Composite State	→	proctype
Concurrent Composite State	→	Concurrent proctypes
Transition	→	Transition

Fig. 12. An excerpted set of formalization rules in Hydra [20].

Analysis. We use the SPIN model checker to simulate and verify the Promela specifications generated by Hydra. Moreover, we visualize the results of checking Promela models in UML state/sequence diagrams. For example, while verifying the UML model of an FTS against interference-freedom constraints, we may find counterexamples that represent the inconsistencies of the corrector pattern and the functional objects. To analyze such inconsistencies, we use SPIN to simulate the counterexamples and use Theseus [22] to visualize each step of the SPIN simulation in UML state/sequence diagrams. Such a visualization of counterexamples facilitates the analysis and refinement of UML models.

ACC Example: In the formalization of the UML model of the ACC system, five proctypes are generated corresponding to the Control, Car *and* Radar *objects and the corrector elements* $c_{control}$ *and* c_{car}*. Fault formalization results in the generation of a* Fault_ACC *proctype that, when executed, may nondeterministically set the values of* Car.disengaged, $Z_{control}$ *and* Z_{car} *to false. To verify the nonmasking fault-tolerance of the candidate model (i.e., the composition of the UML model and the corrector pattern), we first verify the invariant*

$\mathcal{I}_{ACC}$ as an assertion, without including the Fault_ACC *proctype in the generated Promela model (i.e., model in the absence of faults). The corresponding LTL property is specified as $\Box(\mathcal{I}_{ACC})$, which was verified in the absence of faults. We also verify that, in the absence of faults, the candidate model does not deadlock. This ensures that the corrector pattern does not interference with the functional model in the absence of faults. Afterwards, we verified the progress (denoted $\Box(X_{ACC} \Rightarrow \Diamond Z_{car})$) and the convergence (denoted $\Diamond(\Box X_{ACC})$) of the corrector pattern while including the* Fault_ACC *proctype in the Promela model in order to ensure that the corrector pattern is itself nonmasking fault-tolerant. The reachability of the invariant $\mathcal{I}_{ACC}$ is also ensured by the convergence of the corrector pattern; i.e., the candidate model eventually recovers to its invariant. In the verification of interference-freedom constraints, we encountered a counterexample in which the safety of $c_{control}$ was violated. Since $c_{control}$ is instantiated as a detector, we had to modify the model so that this inconsistency is resolved. The Theseus [22] visualization tool highlighted a set of safety-violating transitions in the system state diagram that would reach to a state where* Control.Brakes *was false, but $Z_{control}$ had remained true. Hence, to resolve this inconsistency, we modified such safety-violating transitions by adding some falsification actions that would atomically set $Z_{control}$ to false if the brakes were no longer applied. Notice that, in this case, resolving the inconsistencies of the corrector pattern and the functional model required a change in the behavior of functional model. Such modifications illustrate how the addition of fault-tolerance concerns may require some changes in functional requirements.*

7 Related Work

In this section, we discuss related work for modeling and analysis of error recovery. Several approaches [45, 46] exist for modeling and analyzing dependability aspects most of which focus on system availability and reliability without providing a reusable artifact for specifying error recovery. For example, Lopez-Benitez [45] presents a technique based on stochastic Petri nets for modeling and analysis of local and global system availability in the presence of node and communication link failures. Huszerl and Majzik [47] generate stochastic Petri nets from UML state charts in order to provide quantitative measures for comparing different redundancy management strategies against crash failures. Bondavalli *et al.* [46] present an approach for dependability analysis in both structural and behavioral UML models based on an intermediate Petri net model generated from UML diagrams. While they also model faults as timed transitions in Petri net models and generate a tool-independent intermediate Petri net model from UML diagrams, their approach for modeling fault-tolerance is based on exception handling and replication, whereas the corrector pattern provides an abstract reusable modeling artifact, which can be refined to a fault-tolerance design mechanism (e.g., exception handling).

In error recovery based on exception handling [48–51], the focus is on the design of systems that tolerate exceptional conditions by systematic exception resolution. For example, Xu *et al.* [48] formally model exception handling in distributed systems and use coordinated atomic actions [52] to provide a distributed

mechanism for exception resolution. Garcia and Beder and Rubira [50, 51] separate the concern of exception handling from functional concerns by introducing a meta-level architecture that captures the logic of exception handling in concurrent and distributed systems. While their approach provides a set of patterns for designing different tasks involved in exception handling, no measures are provided for ensuring the fault-tolerance of exception handlers and for verifying the interaction between error recovery and functional concerns in concurrent systems.

In addition to the above approaches, several formal models for error recovery exist in the literature [34, 38, 42, 53, 54] that provide a foundation for automated analysis of error recovery. Arora and Gouda [34] introduce the notion of convergence that presents a generic point of view of recovery in the presence of different types of faults. Based on Arora and Gouda's [34] work, Arora and Kulkarni [42] show that a wide range of legacy fault-tolerance mechanisms can be captured by two basic fault-tolerance components, namely detectors and correctors, based on which we have presented two fault-tolerance analysis patterns [23]. Belli and Grosspietsch [53] provide a hybrid formal framework for modeling and specifying fault-tolerance against erroneous inputs and design flaws, where they use Petri nets for hierarchical specification of concurrent systems and regular expressions for specifying low-level system actions. Magee and Maibaum [54] use modal action logic to specify and verify fault-tolerance in component-based systems, where they adopt a state-based model in partitioning the system state space to the set of normal and abnormal states. Aforementioned approaches provide formal frameworks for specifying and analyzing fault-tolerance concerns, whereas the corrector pattern provides a semi-formal means for capturing and specifying fault-tolerance concerns in earlier stages of the system development lifecycle.

In summary, the corrector pattern provides a design-independent abstraction for capturing the requirements of error recovery before any design decision is made. Such an abstraction simplifies the task of modeling as the focus is on identifying constraints (i.e., correction predicates) that should be satisfied by a fault-tolerant system independent of what design mechanism is used for providing recovery. Moreover, the use of the corrector pattern enables modular specification and analysis of recovery requirements, which in turn simplifies the traceability of recovery from requirements analysis to design and implementation phases. In addition to providing a means for early modeling of recovery, we are investigating the application of techniques for the addition of fault-tolerance [38] in automatic specification and instantiation of the corrector pattern in UML state diagrams.

8 Conclusions and Future Work

In this paper, we introduced an object analysis pattern, called the *corrector* pattern, for modeling and analyzing nonmasking fault-tolerance, where a nonmasking fault-tolerant program guarantees to recover from error conditions to a set of legitimate states (called invariant). Instances of the corrector pattern are added to the UML model of a system to create the UML model of its fault-tolerant version. The corrector pattern also provides a set of constraints

for verifying the consistency of functional and fault-tolerance requirements and the fault-tolerance of the corrector pattern itself. We extended McUmber and Cheng's UML formalization framework [20] 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 nonmasking fault-tolerance, we employed the Theseus visualization tool [22] that animates counterexample traces and generates corresponding sequence diagrams at the UML level. Even though in this paper we presented only the corrector pattern for specifying nonmasking fault-tolerance, we have also developed a companion *detector* pattern [23] for modeling failsafe fault-tolerance, where a failsafe fault-tolerant system guarantees safety even when faults occur. The use of the detector and corrector patterns simplifies and modularizes fault-tolerance concerns and helps to separate the analysis of functional and fault-tolerance concerns, while providing a means to analyze their mutual impact. As an extension of this work, we are investigating the application of a synthesis tool that we have previously developed (called Fault-Tolerance Synthesizer [39]) in automating the identification of the fault-span, scenarios with faults, and the instantiation of the corrector pattern.

References

1. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8), August 1986.
2. B. Randall. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, pages 220–232, 1975.
3. F. Cristian. Exception handling and software fault-tolerance. *IEEE Transactions on Computers*, C-31(6), June 1982.
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. 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.
6. T. Saridakis. A system of patterns for fault-tolerance. *The 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 535–582, 2002.
7. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms. www.omg.org/docs/ptc/04-06-01.pdf, 2002.
8. 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.
9. 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.
10. M. Tkatchenko and G. Kiczales. Uniform support for modeling crosscutting structure. *Appeared in AOM Workshop held in conjunction with AOSD*, 2005.
11. A. Arora. *A foundation of fault-tolerant computing*. PhD thesis, The University of Texas at Austin, 1992.

12. D. Ilic and E. Troubitsyna. Modeling fault tolerance of transient faults. *Proceedings of Rigorous Engineering of Fault-Tolerant Systems*, pages 84 – 92, 2005.
13. L. Laibinis and E. Troubitsyna. Fault tolerance in use case modeling. *the Workshop on Requirements for High Assurance Systems*, 2005.
14. C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Software Practice and Experience*, 35:195–236, 2005.
15. 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.
16. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
18. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.
19. S. Konrad, Betty H. C. Cheng, and L. A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970 – 992, 2004.
20. W.E. McUmber and Betty 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.
21. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 1997.
22. Heather Goldsby, Sascha Konrad, Betty H.C. Cheng, and Stephane 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.
23. Ali Ebnenasir and Betty 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.
24. Ali Ebnenasir and Sandeep S. Kulkarni. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. *In the extended abstracts of the ACM workshop on the Specification and Verification of Component-Based Systems (SAVCBS), Newport Beach, California*, 2004.
25. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
26. B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.
27. 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.
28. 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.
29. S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 337–344, 2002.
30. Sandeep 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.

31. 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):201–215, July-September 2005.
32. M. Demirbas and A. Arora. Convergence refinement. *International Conference on Distributed Computing Systems*, pages 589 – 597, 2002.
33. R. de Lemos and A. Romanovsky. Coordinated atomic actions in modeling objects cooperation. *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 152–161, 1998.
34. 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.
35. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
36. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, apr 2001. Springer-Verlag.
37. G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.
38. Ali Ebnenasir. *Automatic Synthesis of Fault-Tolerance*. PhD thesis, Michigan State University, 2005.
39. Ali Ebnenasir and Sandeep S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cs.mtu.edu/~aebnenas/research/tools/ftsyn.htm`.
40. 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.
41. V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.
42. 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.
43. E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.
44. M.B. Dwyer, G.S. Avrunin, and J.C. 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.
45. N. Lopez-Benitez. Dependability modeling and analysis of distributed programs. *IEEE Transactions on Software Engineering*, 20(5):345–352, 1994.
46. A. Bondavalli and *et al.* Dependability analysis in the early phases of UML-based system design. *International Journal of Computer Systems Science and Engineering*, 5:265–275, 2001.
47. G. Huszerl and I. Majzik. Modeling and analysis of redundancy management in distributed object-oriented systems by using uml statecharts. In *27th Euromicro Conference*, pages 200–207, 2001.
48. J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pages 12–21, May 1998.

49. 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.
50. A. F. Garcia, D. M. Beder, and C. M. F. Rubira. A unified meta-level software architecture for sequential and concurrent exception handling. *The Computer Journal, British Computer Society*, 44(6):569–587, 2001.
51. D. Beder and C. Rubira. A meta-level software architecture based on patterns for developing dependable collaboration-based designs. *In Proceedings of the second Brazilian workshop on fault-tolerance*, 2000.
52. Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS*, pages 499–508, 1995.
53. F. Belli and K. E. Grosspietsch. Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study. *IEEE Transactions on Software Engineering*, 17(6):513–526, 1991.
54. Jeff Magee and Tom Maibaum. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 30–36, 2006.