MR4UM: A Framework for Adding Fault Tolerance to UML State Diagrams ☆,☆☆

Jingshu Chen, Sandeep Kulkarni

Michigan State University,
3115 Engineering Building, 48824 East Lansing, US

Abstract

Modern systems often need to address the challenges brought on by changing environment and/or newly identified faults. The economic and practical issues dictate that the existing models and/or programs be reused while providing fault-tolerance in the presence of faults.

In this paper, we propose a framework, namely MR4UM, for applying model revision for the existing program design modeled in UML state diagram to add tolerance to newly-identified faults. In particular, MR4UM starts with program design modeled in UML state diagram, and automatically transforms design model in UML state diagram to the corresponding program actions in the underlying computational model (UCM). Then, MR4UM applies the techniques of model revision to the program in UCM and generates a fault-tolerant program in UCM. Finally, MR4UM automatically converts the fault-tolerant program in UCM into the fault-tolerant program design in UML state diagram. We illustrate the stepwise procedure of MR4UM with two case studies: the adaptive cruise control program from automotive system and the altitude switch program from aircraft altitude control system.

Keywords: Fault Modeling, Model-based Design, Model Revision, Fault Tolerance.

☆We would like to thank Shige Wang (General Motors) for providing the UML model for the cruise control system that was used as a case study in this paper.

☆☆This work is sponsored by USA AFOSR FA9550-10-1-0178 and NSF CNS 0914913 grants.

A preliminary version of this paper appeared in 13th International Conference on Distributed Computing and Networking (ICDCN 2012).

Email addresses: chenji15@cse.msu.edu (Jingshu Chen), sandeep@cse.msu.edu (Sandeep Kulkarni)

1. Introduction

The utility of formal methods in the development of high assurance systems has gained widespread use in industry. In general, there are two main approaches for utilizing formal methods in providing assurance: *correct-by-verification* and *correct-by-construction*. *correct-by verification*, as one of the most commonly used approaches, begins with an existing program and a set of properties (specifications), and verifies that the given program meets the given properties of interest. An embodiment of this approach is *model-checking* [1, 2, 3, 4]. *Model checking* has been widely studied in the literature and proved effective in identifying bugs in system design. However, a pitfall of this approach is that if the given program does not satisfy the requirements then designer may need to manually revise the program.

The second approach, *correct-by-construction*, constructs a program from the required specification. Intuitively, this approach constructs the entire program from a constructive proof that a specification is satisfiable. Hence the produced program is guaranteed to satisfy the given specification. Examples of this approach include [5, 6, 7, 8, 9, 10]. They differ in terms of the expressiveness of the specifications they permit and in terms of their complexity. This approach has proven to be effective in constructing a program that meets the requirements of specification. However, a pitfall of this approach is the loss of reuse (of the original program). It has a potential for significant increase in complexity.

To obtain the benefits of these two approaches while minimizing their pitfalls, one can focus on an intermediate approach, *model revision*. The goal of model revision is to revise the given program to meet the required specification. It generates a program through *correct-by-construction*, thus it provides assurance comparable to the approach of *correct-by-construction*. Also, it has the potential to reuse the existing program during the revision process. Applications of model revision include scenarios where a given program is proven to violate the required specification by model checking. Then one may utilize model revision to assist in revising program to meet the requirements. Another typical application for model revision is that an existing program needs to be revised due to changes in environments and/or requirements. Model revision has

been studied in contexts where an existing programl needs to be revised to add new fault-tolerance properties, safety properties, liveness properties and timing constraints [11].

However, the difference between the modeling language used in model revision and the practical modeling language of program design hinders utilization of model revision in the practical design. To permit wider application of model revision in the practical design, it is desirable to lower the learning curve. One ideal approach is that a designer could continue to work with a familiar framework/language while utilizing model revision for program design. This approach is challenging since the current modeling language for program design is usually a non-computational one, such as UML [12]. With this motivation, we propose a framework, namely MR4UM, aiming at assisting designer in utilizing model revision. In particular, since one of the most needs in revision is to add fault-tolerance, our work focuses on utilizing model revision in adding fault-tolerance. And our current implementation addresses the problem of utilizing model revision for adding fault-tolerance to UML. UML is a well-known modeling language used in industry, with focus on system architecture as a means to organize computation, communication and constraints. Of UML diagram sets, state diagram is especially helpful when designers discuss the logic architecture and workflow of the whole system. Our implementation currently focuses on automatically revising UML state diagram to add fault-tolerance. Future work will address the issue of utilizing model revision to revise other UML diagrams and other popular languages used in practical design. We also plan to enrich our framework by utilizing model revision to revise program to meet more properties not only fault-tolerance.

Contributions of the paper. The main contributions of the paper are as follows.

- We propose a framework, namely MR4UM, for applying model revision to UML state diagram for adding fault-tolerance.
- MR4UM proposes an automatic transformation between UML state diagram and the underlying computational language.
- We demonstrate the stepwise procedure of MR4UM with two case studies: the

adaptive cruise control program (ACC) from automotive systems and the altitude switch program (ASW) from aircraft altitude controller.

Organization of the paper. In Section 2, we present motivation of the proposed approach. In Section 3, we briefly discuss the related concepts of modeling program design and introduce the underlying computational model. Section 4 presents the definition of model revision in the context of adding fault-tolerance. Next, in Section 5, we describe the stepwise procedure of MR4UM. Section 6 and 7 present the stepwise application of MR4UM on two case studies: the ACC system and ASW program. Related work is discussed in Section 8. Section 9 discusses several questions that are raised by our work as well as lessons learnt from the case studies. Finally, Section 10 makes concluding remarks and discusses some possible future work.

2. Motivating Scenario

In this section, we present a motivating scenario from automotive industry. An adaptive cruise control program (ACC) in automotive system is designed to control the distance between the vehicle and the front vehicle (called leader car) automatically. Figure 1 describes the logic design of the ACC program.

As shown in Figure 1, when ACC program is on and after initialization, the system enters one of the three modes: *active*, *ACC_active* or *inactive*. In the *active* mode, the sensor keeps checking whether a leader car exists within a predefined safe distance and the ACC program keeps reading the sensor result. In the *ACC_active* mode, the ACC program controls the distance between the leader car and the current car because a leader car is detected within the predefined safe distance. In the *inactive* mode, the driver is pressing the brakes and the adaptive cruise control relinquishes control to driver for manual control without considering the effect of cruise. By switching among these three modes, the ACC system ensures that the current car is at a safety distance away from the leader car and it moves at a steady speed relative to the leader car.

Initially, when no leader car is detected by the sensor, the program stays in *active* mode. If a leader car is detected within the predefined safe distance, the ACC program enters into *ACC_active* mode. ACC program stays in *ACC_active* mode until the leader

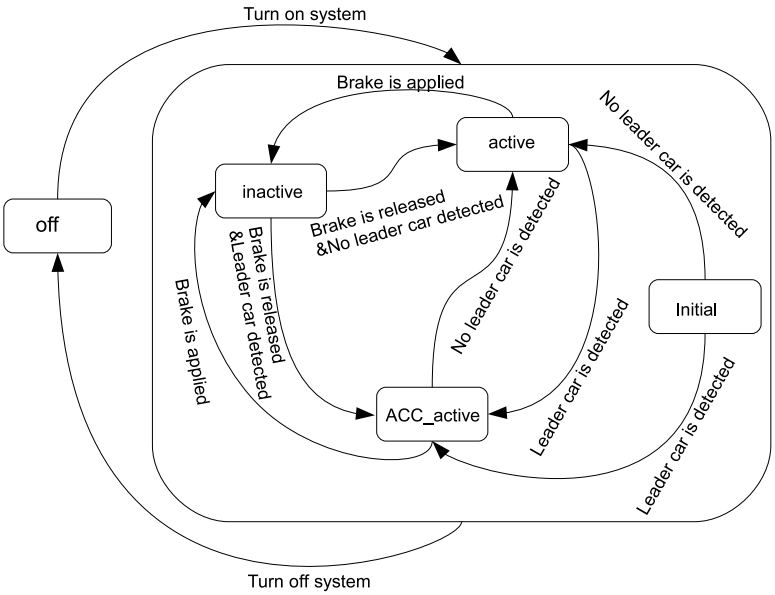

Figure 1: Logic Design of the ACC Program

car moves away or the drive takes the brake. If the leader car moves away from the detectable distance, the ACC system goes back to the *active* mode. Under *ACC_active* mode or *active* mode, the ACC system enters into *inactive* mode when driver taps the brake. When driver stops tapping the brake and presses resume button, the ACC system enters into *active* mode if no leader car is detected and *ACC_active* mode if a leader car is detected. (For simplicity, we do not model the *resume* action. We simply assume that releasing the break is equivalent to resume.)

2.1. Need for Model Revision for Tolerating Sensor Failure

While the ACC program in Figure 1 works correctly in the absence of faults, it may result in undesired behavior if some faults affect the sensor. Specifically, a sensor failure may cause two problems: *false positive* and *false negative*. A *false positive* sensor may cause the sensor to detect a non-existing leader vehicle causing the system to change the state from *active* to *ACC_active*. This would potentially cause the car to slow down unnecessarily to prevent collision with a fictitious car.

A more serious error can result in a *false negative* sensor that fails to detect a leader

vehicle. In this scenario, the car would stay in *active* mode; thereby potentially cause a collision with the leader car.

For the above reasons, the model in Figure 1 needs to be revised to deal with such false alarms (*false positive* & *false negative*). To tolerate the false alarm (*false positive* and *false negative*) caused by the faulty sensor, one typical fault tolerance policy is to provide redundancy. Thus, if the redundancy policy is chosen to tolerate the sensor failure, the problem of revising program design to resist false alarm is to modify the previous system design to utilize the sensor redundancy. After revision, the new system design should get correct information about whether the leader car exists, even in the presence of the false alarm of one sensor.

Motivated by the above scenario, we propose MR4UM, that aims at facilitating program revision to add fault-tolerance with minimal overhead. Specifically, MR4UM requires the original program design and the identified fault(s) as input. The input program should be in the format that the designer is familiar with. The format preservation for input program is a key feature since it will not introduce a new overhead in the scenario where the designer has already had the program design in some specific format. Our current implementation only supports the input program in the UML state diagram. We will enrich the implementation to support other input formats. The designer also needs to identify the behaviors of faults. MR4UM provides a mechanism to assist the designer to specify the actions of faults. Subsequently, MR4UM automatically generates the fault-tolerant program from the input specified by the designer.

3. Modeling

In this section, we present the approach of modeling program in UML diagram as well as in the underlying computation model. These approaches are used for both the original (fault-intolerant) and revised (fault-tolerant) programs.

3.1. Program Design in UML State Diagram

UML is a well-known standardized general-purpose modeling language in the field of object-oriented software engineering. The current version of UML is 2.4.1 and it

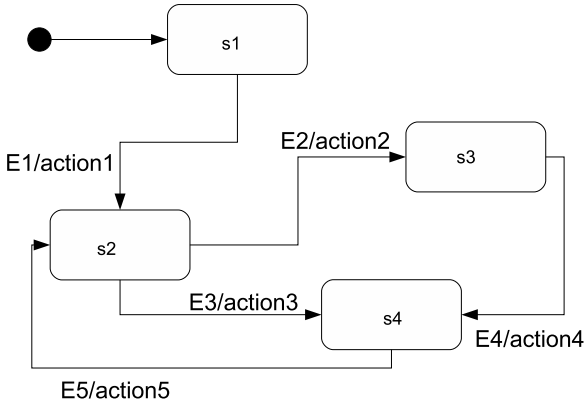

Figure 2: A case to illustrate modeling program in UML state diagram.

has 14 types of diagrams. Our framework currently focuses on adding fault-tolerance to UML state machine diagram. The reason behind choosing UML state machine diagram is that UML state diagram enables us to capture any form of fault-tolerance that can be expressed in a state machine-based formalism [13].

UML state machine diagram describes the states and state transitions of the system. Specifically, the basic elements of a state machine diagram is defined as follows:

1. **State.** A state is represented as a rounded rectangle. The initial state (if any) is denoted by a filled circle. The final state (if any) is denoted as a hollow circle containing a smaller filled circle. The states may be hierarchical in nature and a hierarchical state can be decomposed into several states.
2. **Transition.** A transition switching from one state to another is represented with an arrow. Each transition is associated with a triggering event followed by the list of executed actions. The initial transition originates from the initial state.

The fragment in Figure 2 illustrates one example of UML state machine diagram. This example includes five states: $initial\ state$ (denoted as s_0), s_1, s_2, s_3 and s_4, and six transitions: $s_0 \rightsquigarrow s_1$, $s_1 \rightsquigarrow s_2$, $s_2 \rightsquigarrow s_3$, $s_2 \rightsquigarrow s_4$, $s_3 \rightsquigarrow s_4$ and $s_4 \rightsquigarrow s_2$.

3.2. Underlying Computational Model(UCM)

In this section, we describe the underlying computational model (UCM) that is used during the revision process to add fault-tolerance to the existing UML model. Towards

this end, we convert the model from Section 3.1 to UCM for the revision process. The UCM is adapted from [14] and, hence is suitable for use in the synthesis engine proposed in [14].

Intuitively, the program $\mathcal{P}$ is described in terms of its variables V $\{v_0, v_1, \ldots, v_n\}$ and its transitions p. For such program, a state s of $\mathcal{P}$ is determined by the function $s : V \rightarrow \{true, false\}$, which maps each variable in V to either $true$ or $false$. Thus, a state is represented as the conjunction:

$$s = \bigwedge_{j=0}^{n} l(v_j) \tag{1}$$

where $l(v_j)$ denotes a *literal*, which is either v_j or its negation $\neg v_j$. Note that this notion of state is not restricted to Boolean variables, due to the fact that non-Boolean variables with finite domain D can be represented by $log(|D|)$ Boolean variables. The $state\ space$ is the set of all possible states obtained from the program variables.

Let V' be a set of primed variable: $V' = \{v' | v \in V\}$. Such prime variables are used to denote the target value of variables assigned by a transition. A transition is a pair of states of the form (s, s') specified as a Boolean formula: $s \wedge s'$. The program action is a finite set of transitions $\{t_0, t_1, \ldots, t_n\}$, represented as the disjunction

$$\psi_{\mathcal{P}} = \bigvee_{j=0}^{n} (t_j) \tag{2}$$

Based on Equation 1 and 2, we can also define program in another equivalent fashion in terms of state space $S_{\mathcal{P}}$ and transitions $\psi_{\mathcal{P}}$ as follows:

Definition 1. ***(program)*** *A* program $\mathcal{P}$ *is a tuple* $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$*, where* $S_{\mathcal{P}}$ *is the set of all possible states, and* $\psi_{\mathcal{P}}$ *is a set of transitions, where* $\psi_{\mathcal{P}}$ *is a subset of* $S_{\mathcal{P}} \times S_{\mathcal{P}}$*.* □

Definition 2. ***(computation)*** *A* computation *of* $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ *(or briefly* $\psi_{\mathcal{P}}$*) is a finite or infinite state sequence:* $\overline{s} = \langle s_0, s_1, \ldots \rangle$ *s.t. the following conditions are satisfied: (1)* $\forall j : 0 < j < lengthof(\overline{s}) : (s_{j-1}, s_j) \in \psi_{\mathcal{P}}$*, (2) if* $\overline{s}$ *is finite and terminates in* s_f *then there does not exist any state* s *such that* $(s_f, s) \in \psi_{\mathcal{P}}$*.* □

Definition 3. ***(safety specification)*** *For program $\mathcal{P}$, the* safety specification *ϕ is specified as a set of bad transitions [15] where $\phi \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$.* □

Notation. A transition $\mathcal{T}_t(s_0, s_1)$ **v**iolates the safety specification ϕ iff $\mathcal{T}_t \in \phi$. A sequence $\sigma = \langle s_0,\ s_1,\ \ldots\rangle$ satisfies ϕ iff $\forall j : 0 < j < lengthof(\sigma) : (s_{j-1}, s_j) \notin \phi$.

Definition 4. ***(state predicate)*** *A* state predicate *$\mathcal{S}$ is any subset of $S_{\mathcal{P}}$.* □

Definition 5. ***(closure)*** *A state predicate S is* closed *in program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}}\rangle$ (or briefly $\psi_{\mathcal{P}}$) iff $(\forall (s_0, s_1) \in \psi_{\mathcal{P}} : ((s_0 \in S) \Rightarrow (s_1 \in S)))$.* □

Now we give the formal definition of **satisfies**, i.e., what it means for a program $\mathcal{P}$ to satisfy a specification ϕ.

Definition 6. ***(satisfies)*** *Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}}\rangle$, a state predicate $\mathcal{S}$, and a specification ϕ for $\mathcal{P}$. $\mathcal{P}$* satisfies *ϕ from $\mathcal{S}$ iff (1) $\mathcal{S}$ is closed in $\psi_{\mathcal{P}}$, and (2) every computation of $\mathcal{P}$ that starts from a state in $\mathcal{S}$ satisfies ϕ.* □

In the rest of the paper, we write $\mathcal{P} \models_{\mathcal{S}} \phi$ to denote $\mathcal{P}$ *satisfies* ϕ.

Definition 7. ***(maintains)*** *A program $\mathcal{P}$* maintains *a given specification ϕ from a state predicate $\mathcal{S}$ iff (1) $\mathcal{S}$ is closed in $\psi_{\mathcal{P}}$, and (2) for all computation prefixes α of $\mathcal{P}$, there exists a computation suffix β such that $\alpha\beta \in \phi$. We say that $\mathcal{P}$* violates *ϕ iff it is not the case that $\mathcal{P}$ maintains ϕ.* □

Definition 8. ***(invariant)*** *Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}}\rangle$, a state predicate $\mathcal{I}$, and a safety specification ϕ for $\mathcal{P}$. $\mathcal{I}$ is an* invariant *of $\mathcal{P}$ for ϕ if $\mathcal{P} \models_{\mathcal{I}} \phi$ and $\mathcal{I} \neq \{\}$.* □

We use "$\mathcal{I}$ is an invariant of $\mathcal{P}$" to abbreviate "$\mathcal{I}$ is an invariant of $\mathcal{P}$ for ϕ" in the remaining context of this paper whenever the specification is clear from the context.

Next, we define fault f. The faults f that a program is subject to are systematically represented by transitions. Based on the classification of faults from [16], this representation suffices for physical faults, process faults, message faults and improper initialization. It is not intended for program bugs (e.g. buffer overflow). However, if such bugs exhibit behavior such as component crash, it can be modeled by using this approach. Thus, a fault for $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}}\rangle$ is a subset of $S_{\mathcal{P}} \times S_{\mathcal{P}}$.

Definition 9. ***(fault-span)*** *A state predicate $\mathcal{FS}$ is an f-span (read as* fault-span*) of $\mathcal{P} = \langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ from an invariant $\mathcal{I}$ iff the following conditions are satisfied: (1) $\mathcal{I} \subseteq \mathcal{FS}$, and (2) $\mathcal{FS}$ is closed in $\psi_{\mathcal{P}} \cup f$.* □

Observation. For all computations of $\mathcal{P}$ that start from states in $\mathcal{I}$, $\mathcal{FS}$ is a boundary in the state space of $\mathcal{P}$ up to which (but not beyond which) the states of $\mathcal{P}$ may be perturbed by the occurrence of the transitions in f.

Definition 10. ***(fault-tolerance)*** *A program $\mathcal{P}$ is f-tolerant from an invariant $\mathcal{I}$ for ϕ, iff the following conditions hold:*

1. $\mathcal{P} \models_{\mathcal{I}} \phi$*;*
2. *There exists $\mathcal{FS}$ such that:*
   (a) *$\mathcal{FS}$ is an f-span of $\mathcal{P}$ from $\mathcal{I}$;*
   (b) *$\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \cup f \rangle$ maintains ϕ from $\mathcal{FS}$;*
   (c) *Every computation of $\langle S_{\mathcal{P}}, \psi_{\mathcal{P}} \rangle$ that starts from a state in $\mathcal{FS}$ eventually reaches a state of $\mathcal{I}$;*. □

We denote fault tolerance defined in Definition 10 as masking fault tolerance. Specifically, if program $\mathcal{P}$ is masking f-tolerant from $\mathcal{I}$ for ϕ then $\mathcal{I}$ is closed in $\psi_{\mathcal{P}}$ and every computation of $\mathcal{P}$ that starts from a state in $\mathcal{I}$ satisfies ϕ in the absence of faults. Additionally, in the presence of faults, there is a fault-span predicate $\mathcal{FS}$ ($\mathcal{I} \subseteq \mathcal{FS}$) that is closed in $\psi_{\mathcal{P}} \cup f$.

4. Utilizing Model Revision to Add Fault-tolerance

In this section, we define the model revision problem in the context of adding fault-tolerance.

Problem 4.1. ***The Model Revision Problem*** *for Adding Fault-tolerance*

Given a program $\mathcal{P}$, a safety specification ϕ, an invariant $\mathcal{I}$ of $\mathcal{P}$ from where $\mathcal{P}$ satisfies ϕ and a set of fault actions $\mathcal{F}$: Does there exist a $\mathcal{P}'$ with an invariant $\mathcal{I}'$ such that

- *(C1) $\mathcal{I}' \subseteq \mathcal{I}$,*

- *(*$C2$*)* $(s_0, s_1) \in \mathcal{P}' \wedge s_0 \in \mathcal{I}' \Rightarrow (s_0, s_1) \in \mathcal{P}$*, and*
- *(*$C3$*)* $\mathcal{P}'$ *is fault tolerant to* $\mathcal{F}$ *from* $\mathcal{I}'$ *for* ϕ*.* □

Since the goal of this problem is to add fault-tolerance, the revision for adding fault-tolerance is not permitted to add new behaviors in the absence of faults. To meet this requirement, we include two constraints $C1$ and $C2$. Specifically, constraint $C1$ states that $\mathcal{I}'$ is a subset of $\mathcal{I}$. If $C1$ is not true then it implies that the fault-tolerant program could begin in a state from where the original fault-intolerant program violates its specification. Hence, we cannot conclude correctness of the program behavior (in the absence of faults) if it starts from a state in $\mathcal{I}' - \mathcal{I}$. Thus, constraint $C1$ is required. Likewise, if $\mathcal{P}'$ has new transitions in $\mathcal{I}'$ then it would imply that $\mathcal{P}'$ could have behaviors that are not in $\mathcal{P}$. In other words, the computation of $\mathcal{P}'$ may generate new ways to satisfy $spec$ in the absence of faults. Hence, constraint $C2$ is required.

Based on the above problem statement, we can view transitions of $\mathcal{P}'$ in two parts: (1) transitions of $\mathcal{P}$ that are preserved and (2) new transitions that are added to provide recovery from faults. Furthermore, since a transition in a UML model may correspond to several transitions in the underlying computation model. For this reason, the first part can be subdivided into transitions that are preserved as is and transitions that are preserved in part, i.e., they are restricted to execute under certain conditions. Thus, the transitions of the fault-tolerant program can be partitioned into three types:

1. **O**riginal Transitions $\mathcal{T}_o$. $\mathcal{T}_o$ corresponds to transitions that are preserved as is in the fault-tolerant program.
2. **S**trong Transitions $\mathcal{T}_s$. $\mathcal{T}_s$ corresponds to transitions of the original program that are restricted to execute under certain constraints. In other words, these transitions are strengthened version of the original actions.
3. **R**ecovery Transitions $\mathcal{T}_r$. $\mathcal{T}_r$ does not have the counterpart actions in the input program. $\mathcal{T}_r$ is added by model revision to provide recovery in the presence of faults.

5. Framework Description

In this section, we describe the approach of MR4UM. There are four main steps in MR4UM, including: 1) Step A, converting program design in the UML state diagram into the underlying computational model (UCM); 2) Step B, identifying the effect of faults; 3) Step C, utilizing model revision to add fault tolerance to the input program design; and 4) Step D, converting the revised program in UCM into UML state diagram while utilizing the annotations generated in Step A. Figure 3 describes the work-flow of the whole framework.

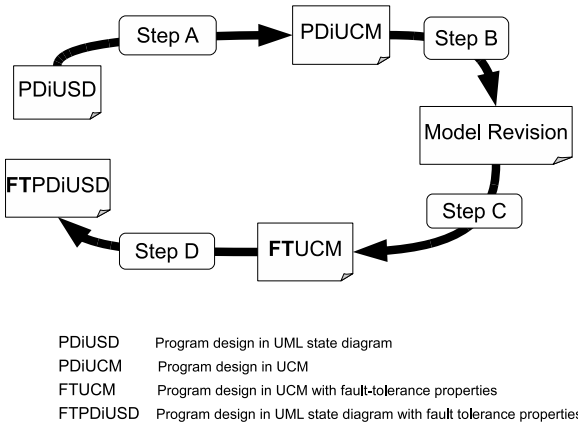

Figure 3: The Stepwise Procedure of MR4UM.

5.1. Step A: Automatically Translate from Program Design Modeled in UML State Diagram to UCM.

In order to utilize the underlying synthesis machine to revise the current design modeled in UML, we translate the UML diagram of the system into the system description based on UCM. This step utilizes syntactic transformation of the UML model. In particular, for every syntactic feature of the UML model, we convert it into the corresponding UCM. This step is automated with the help of following rules:

- **Rule 1: Translation of states in UML state model.** For all the rectangles (representing states) in the UML state diagram, we introduce variable STATE with integer domain $[0, n-1]$ where n is the number of rectangles in the UML.

All the rectangles in the UML state diagram are numbered from 0 to $n-1$. For each rectangle in the UML diagram, we introduce one possible assignment of variable STATE. If UML model consists of hierarchical states then this rule is applied to the states at the lowest level. For example, for a rectangle in the UML state diagram which is numbered with 0, it is mapped into STATE==0.

- **Rule 2: Translation of trigger conditions.** For each trigger condition c mentioned in the UML state diagram, we introduce one variable X_c with domain $\{0,1\}$. $X_c = 1$ denotes this trigger condition is satisfied. $X_c = 0$ denotes this trigger condition is not satisfied. Note that the state space is now defined in terms of the STATE variable from Rule 1 and all the variables from this rule.

- **Rule 3: Translation of transitions.** For each transition in the UML state diagram, we introduce one corresponding program transition P. The guard of P is the conjunction of the corresponding STATE assignment of source state and the corresponding variables of each trigger conditions. The action of P changes the assignment of STATE according to the target state of the original transition in the UML.

Observe that the application of these rules will facilitate the translation of the UML model into the underlying computational model. Since this model is obtained by the above rules, it is straightforward to observe that it has the same behavior as the original UML model.

5.2. Step B: Generate Fault Actions, Specification and Invariants from Parameters specified by Designer

After Step A, we have program actions modeled in UCM. To revise the program design to satisfy the new specification, we need three additional inputs for solving Problem 4.1. They include: (1) fault actions, (2) specification, that is, requirements in the presence of faults, and (3) all the states that program can recover to after faults occur. Moreover, these parameters are closer in spirit to the underlying computational model. For this reason, asking the designer to generate these in UCM is not desired. Unfortunately, they cannot be derived automatically either. For this reason, our framework

facilitates modeling of typical faults that one may encounter in the revision process. Next, we describe how MR4UM obtains these inputs.

1. **Fault Actions Modeled in UCM.** In our framework, *fault actions* are automatically generated from parameters which are specified by designer from GUI. From GUI, designer needs to specify the following parameters:

   - *What type of faults?* Currently, there are three types of faults modeled in our framework: (1) Byzantine, (2) transient and (3) crash (fail-stop). The Byzantine fault allows the fault to change the affected component (variables of the component) in an arbitrary manner. Moreover, this fault can perturb the program several times. A transient fault perturbs the component (variables of the component) once (respectively, finitely many times where the bound is known upfront). And, crash disables certain variables from being updated and, hence, captures the notion that a component (containing those variables) has crashed. The default setting of our framework is *transient*.

   - *Effect of faults on program.* Designer needs to specify the variables the fault affects after specifying types of faults that may occur during the execution: In particular,

     (a) *Byzantine.* For this type of fault, designer needs to specify which variable(s) may be corrupted by the Byzantine component and the possible value(s). For example, in the motivating scenario used in Section 6, the variable representing the leader is affected by faults. As described in Section 6, this fault can perturb the program to an arbitrary state. Hence, the default for this fault is that the variable can be corrupted to any value in its domain. However, if the likely fault is only a false positive then, the designer can state this by saying that the fault can only perturb leader to 1.

     (b) *Transient.* For this type of fault, designer needs to specify which variable is perturbed to the random value. The default for this fault is that the variable can be corrupted to any value in its domain. The difference

between the transient and Byzantine fault is that the former perturbs the program once whereas the latter could perturb the program a finite number of times.

(c) *Crash or Fail-stop.* For this type of fault, designer needs to specify which variables are prevented from update due to the fault. In our framework, we use the *crash* faults to denote the fault that is not detectable. By contrast, *fail-stop* is detectable.

- *Number of occurrences of faults.* Designer also needs to specify the occurrences of the specified faults. In case of Byzantine/crash faults, the number denotes the number of Byzantine/crashed components. In turn, this determines the required level of redundancy. Regarding transient faults, the number denotes the occurrences of transient faults that may occur during the computation. The default setting value is 1.

2. **Specification, that is, Requirements in the Presence of Faults.** In our framework, *specification* is automatically generated from parameters which are specified by designer from GUI. Designer needs to specify each state with variables and corresponding values. The union of the specified states from GUI are used to generate the specification automatically.
3. **States Where Program should Recover after Fault Occurs.** The states where program should recover after faults occurs are generated automatically from initial states (by performing reachability analysis) specified in UML state diagram.

5.3. Step C: Utilizing Model Revision to Add Fault-tolerance

After the first two steps, we have program actions, faults, specification and invariant. In this step, we utilize the technique of model revision to generate the fault-tolerant program. Specifically, we utilize the revision engine in [17]. we review the model revision algorithm in the revision engine next. The details of algorithm are shown in Figure 4. The model revision algorithm requires four inputs: the actions of $\mathcal{P}$(original program), specification ϕ, the invariant $\mathcal{I}$ from where $\mathcal{P}$ satisfies ϕ and fault actions $\mathcal{F}$. It consists of five steps, as follows:

1. **Initialization (Lines 1-3).** In this step, we identify state and transition predicates from where execution of faults alone may violate safety specification. Specifically, if (s_1, s_2) is a fault transition that violates safety then the program should never reach s_1. This is due to the fact that if the program reaches s_1 then execution of fault action can violate safety. Furthermore, if (s_0, s_1) is also a fault transition then s_0 should also not be reached. Otherwise, execution of two fault transitions would violate safety. Continuing this process (with backward reachability in fault transitions), we obtain the set ms such that the fault-tolerant program should not reach a state in ms. For the same reason, the program should not include a transition that reaches ms. Likewise, it should not include transitions that violate safety. Hence, mt denotes the set of these transitions that the program should not include.
2. **Identification of Fault-span (Lines 9-11).** In this step, we identify the fault-span, that is, the reachable states by the program in the presence of faults starting from the program invariant.
3. **Identifying and removing unsafe transition (Line 12-13).** In this step, we identify and remove transitions in mt. It turns out that in several scenarios, the transitions of the fault-tolerant program need to be executed with partial information. For example, consider the example in Section 2. Here, the program cannot read the variable that denotes whether the leader car actually exists. However, such a variable is needed during modeling so we can verify that the transitions of the adaptive cruise control system are consistent with each other. Hence, any time a transition is removed/added, we need to add the corresponding *Group* of transitions; specifically, this group is obtained by changing the values of variables that the program cannot read.

   Unfortunately, this restriction of adding and removing groups causes the complexity of model revision. Hence, we utilize the following heuristics.

   Heuristic H_1: $\mathcal{P}'$ may include (s_0, s_1),where $s_0 \in ms$.

   Reasoning behind H_1. H_1 is based on the premise that the algorithm will ensure that the program never reaches a state in ms. Hence, even if we include such transitions, it will not cause any problems. Moreover, such inclusion is helpful

to increase the success of model revision when (s_0, s_1) is grouped with some other transitions that is desirable in the $\mathcal{P}'$.

Heuristic H_2: $\mathcal{P}'$ may include (s_0, s_1) where $(s_0, s_1) \in mt$ and s_0 is not reachable by transitions in $\mathcal{P}'$ starting from $\mathcal{I}'$ in the presence of faults.

Reasoning behind H_2. H_2 is based on the premise that if the current version of $\mathcal{P}$ does not reach s_0 then it is likely to be true in the final program as well. Hence, including transition (s_0, s_1) is expected to be acceptable. If at some later point, state s_0 is reachable then this transition may be removed. The fault-span computed in this step is used to determine whether state s_0 is reachable in the presence of faults.

4. **Resolving deadlock states (Line 15-18).** To ensure that no new finite computations are introduced to the input fault-intolerant program, we resolve deadlock states in this step by either adding recovery path or eliminating states. First, we attempt to add recovery to the invariant. This recovery could be achieved in a single step or in multiple steps. Only if the recovery is not feasible then we remove transitions so that the deadlock state is not reached. In this step, we use the following heuristic.

   Heuristic H_3: Given a deadlock state $s, s \notin \mathcal{I}'$, $\mathcal{P}'$ either includes a recovery action $(s, s_I), s_I \in \mathcal{I}'$, or makes s unreachable from $\mathcal{I}'$ without eliminating any invariant states.

   Reasoning behind H_3. H_3 is based on the principle that $\mathcal{P}'$ would not eliminate any states and/or transitions unless absolutely required to do so. This is due to the fact that if we remove several states from the invariant then it may be impossible to satisfy specification in the absence of faults. Hence, invariant states are removed only as a last resort.

5. **Re-computing the invariant (Line 20).** In this step, we recomputed the program invariant due to identifying offending states during state elimination.

The algorithm keeps repeating steps until the three fix-points in Line 14, 19 and 20 are reached. The algorithm terminates when no progress is possible in all the steps.

Thus, at the end of this step, we have a fault-tolerant program in UCM that is obtained by the corresponding input from Steps A and B. Moreover, as mentioned

INPUT:$\psi_{\mathcal{P}}$:transitions, f:fault transitions, $I_{\mathcal{P}}$: invariant predicate, *spec*: safety specification
OUPUT:$\psi_{\mathcal{P}'}$: transitions of fault tolerant program and $I_{\mathcal{P}'}$: invariant predicate
$ms := \{s_0 \ : \ \exists s_1, \ s_2, \ \ldots s_n \ :$
$\quad (\forall j \ : \ 0 \leq j < n \ : \ (s_j, \ s_{j+1}) \in f) \wedge \ (s_{n-1}, s_n) \ violates \ spec\};$ (1)
$mt := \{(s_0, s_1) \ : \ ((s_1 \ \in ms) \ \vee \ (s_0, \ s_1) \ violates \ spec)\};$ (2)
$I_1, fte := I_{\mathcal{P}} - ms, \ false;$ (3)

REPEAT (4)
$\quad I_2 := I_1;$ (5)
$\quad$ REPEAT (6)
$\quad\quad S_1, \psi_2 := I_1, \psi_1;$ (7)
$\quad\quad$ REPEAT (8)
$\quad\quad\ S_2 := S_1;$ (9)
$\quad\quad\ S_1 := FWReachStates(I_1, \psi_1 \vee f);$ (10)
$\quad\quad\ S_1 := S_1 - fte;$ (11)
$\quad\quad\ mt := mt \wedge S_1;$ (12)
$\quad\quad\ \psi_1 := \psi_1 - Group(\psi_1 \wedge mt);$ (13)
$\quad\quad$ UNTIL$S_1 = S_2$; (14)
$\quad\quad ds := \{s_0 \mid s_0 \ \in \ S_1 \ \cup \ (\forall s_1 \ : \ s_1 \in S \ : \ (s_0, \ s_1) \notin \psi_1\};$ (15)
$\quad\quad \psi_1 := \psi_1 \vee AddRecovery(ds, I_1, S_1, mt);$ (16)
$\quad\quad ds := \{s_0 \mid s_0 \ \in \ S_1 \ \cup \ (\forall s_1 \ : \ s_1 \in S \ : \ (s_0, \ s_1) \notin \psi_1\};$ (17)
$\quad\quad \psi_1, fte := Eliminate(ds, \psi_1, I_1, S_1, f, false, false);$ (18)
$\quad$ UNTIL $\psi_1 = \psi_2;$ (19)
$\quad \psi_1, I_1 := ConstructInvariant(\psi_1, I_1, fte);$ (20)
UNTIL $I_1 = I_2;$ (21)
$I_{\mathcal{P}}, \psi_{\mathcal{P}'} := I_1, \psi_1;$ (22)
RETURN $I_{\mathcal{P}'}, \psi_{\mathcal{P}'};$ (23)

Figure 4: Add_FT from [14]

earlier, the output in this step consists of: (1) original transitions that are preserved as is, (2) original transitions that are strengthened and (3) recovery transitions.

5.4. Step D: Translate FT Program Modeled in UCM to FT Program Design in UML state diagram.

After the above steps, we obtain revised program modeled in UCM, including (1) *original* program actions, (2) *revised* program actions and (3) *recovery* program actions. We generate the fault-tolerant program design in UML state diagram as follows:

1. We utilize the *revised* program actions to identify the *changed* transitions in the original UML state diagram.
2. We utilize the *recovery* program actions to add new transitions in the UML state diagram.

3. We re-annotate these transitions in the UML state diagram by the guard conditions of these *revised* program actions and *recovery* program actions.

Thus, after MR4UM executes four steps, we obtain a fault-tolerant program design in UML state diagram.

6. Case Study 1: The Adaptive Cruise Control System

In this section, we present the stepwise application of MR4UM on the case of the adaptive cruise control (ACC) system introduced in Section 2. In particular, we begin with the fault-intolerant UML model for this case study and apply our framework to generate a fault-tolerant UML model that satisfies the conditions of Problem 4.1. This case study is organized as follows: First, in Section 6.1, we describe UML state diagram for the fault-intolerant ACC program. In Section 6.2, we describe how MR4UM transforms this UML state diagram into program actions modeled in UCM. In Section 6.3, we describe the user inputs for generating the fault model, specification and invariant. Subsequently, in Section 6.4, we describe how MR4UM utilizes model revision algorithm to add fault tolerance to the input program. Finally, in Section 6.5, we show how MR4UM transforms the fault-tolerant program obtained by model revision into the corresponding fault-tolerant UML state diagram.

6.1. Fault-intolerant UML model for ACC

The model of the ACC system design in the UML state diagram includes five states, namely *active*, *ACC_active*, *inactive*, *initial* and *off*. The *active* state captures the status of ACC system in "active" mode. The *ACC_active* state captures the status of the ACC system in "ACC_active" mode. The *inactive* state captures the status of the ACC system in "inactive" mode. The *initial* state captures the status of the ACC system in the initializing process. The *off* state captures the status that the ACC system is turned off.

The state diagram in Figure 5 visualizes model design of the ACC system. For better understanding, Figure 6 labels formal expression of the corresponding annotation. As shown in Figure 5, when the ACC system is turned on, the system enters into

the *active* state after initialization if no leader car is detected. The system enters into the *ACC_active* state if the leader car is detected. In other words, whether a leader car is detected in a predefined safety distance is the trigger condition to change system state between *active* and *ACC_active*. When the brake is applied, irrespective of whether the system is in *active* or *ACC_active*, the ACC system enters into *inactive* state. When the brake is released, the system switches from *inactive* state into *active* state or *ACC_active* state depending upon whether a leader car is detected. The whole ACC system continues to stay in one of these three states until the system is turned off.

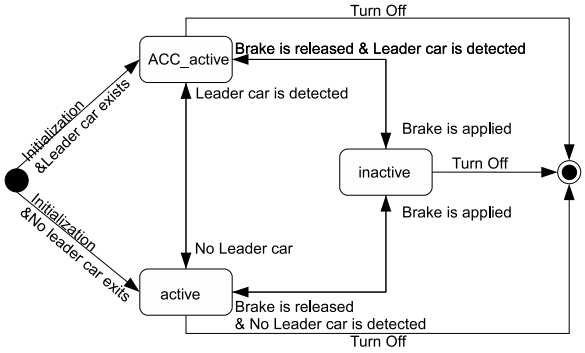

Figure 5: ACC System Modeled in UML State Diagram.

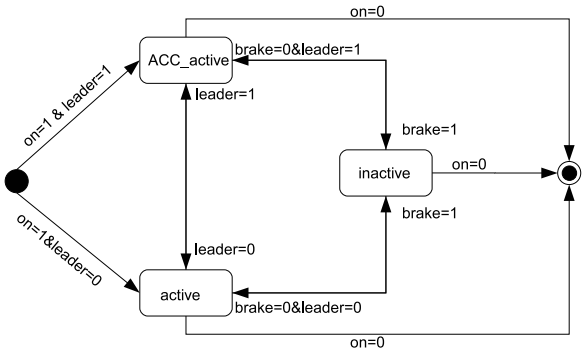

Figure 6: Annotation in Formal Expression.

As one can observe this program works correctly in the absence of faults. The system ensures that the current car maintains a safe distance away from the detected

leader car. And, when the leader car moves away, the system enters into *active* to ensure that the previous speed is resumed.

6.2. Application of Step A: Generating UCM of the ACC System

The ACC system in the UML state diagram is as shown in Figure 6. Based on the transformation discussed in Section 5, generation of program in UCM needs to introduce four variables, namely *state*, *on*, *brake* and *leader*. The details of these variables are as follows:

1. *state*. MR4UM defines the domain of *state* is $\{0, 1, 2, 3, 4\}$. The domain range of *state* captures the five states of the ACC system. $state = 0$ denotes the initial status of the system when it is turned on. $state = 1$ denotes the system is in *active* status. $state = 2$ denotes the system is in *ACC_active* status. $state = 3$ denotes the system is in *inactive* status. And, $state = 4$ denotes the status that the system is turned off.
2. *on*. The domain of variable *on* is $\{0, 1\}$. It is used to denote whether the ACC system is turned on. When the ACC system is turned on, the variable *on* is assigned with 1, otherwise 0.
3. *leader*. The domain of variable *leader* is $\{0, 1\}$. It is used to model whether a leader car is detected by the sensor system. $leader = 1$ denotes that a leader car is detected. $leader = 0$ denotes that no leader car is detected.
4. *brake*. The domain of variable *brake* is $\{0, 1\}$. It is used to denote whether the brake is applied by the driver. $brake = 1$ models the event that the brake is applied during the execution of ACC system. $brake = 0$ models the event that the brake is released during the execution of ACC system.

Furthermore, based on the transformation discussed in Section 5, the program actions of the ACC system in UCM are as follows:

1. $state = 0$ & $on = 1$ & $leader = 0 \longrightarrow state := 1$;
2. $state = 0$ & $on = 1$ & $leader = 1 \longrightarrow state := 2$;
3. $state = 1$ & $leader = 1 \longrightarrow state := 2$;

4. $state = 2$ & $leader = 0 \longrightarrow state := 1$;
5. $state = 1$ & $brake = 1 \longrightarrow state := 3$;
6. $state = 2$ & $brake = 1 \longrightarrow state := 3$;
7. $state = 3$ & $brake = 0$ & $leader = 0 \longrightarrow state := 1$;
8. $state = 3$ & $brake = 0$ & $leader = 1 \longrightarrow state := 2$;
9. $state = 1$ & $on = 0 \longrightarrow state := 4$;
10. $state = 2$ & $on = 0 \longrightarrow state := 4$;
11. $state = 3$ & $on = 0 \longrightarrow state := 4$;

In the above program actions, Action 1 models the transition from *initial* state ($state = 0$) to *active* state ($state = 1$). The triggering condition of this transition includes: 1) no leader car exists within the predefined safety distance and 2) the ACC system is on. This condition is modeled as $on = 1 \& leader = 0$. Action 2 models the transition from *initial* state to *ACC_active*($state = 2$). The triggering condition is that a leader car is detected within the predefined safety distance when the system is on, that is, $on = 1 \& leader = 1$. Action 3 models the transition from *active* ($state = 1$) to *ACC_active* ($state = 2$). The triggering condition for this action is a leader car is detected within the predefine safety distance. Action 4 models a reverse transition of action 3, that is, from *ACC_active* ($state = 2$) to *active* ($state = 1$). The triggering condition for this action is no leader car is detected within the predefined safety distance. Actions 5 and 6 models the transition from *active* (or *ACC_active*) to *inactive*. The triggering condition for both actions is that driver is pressing brakes, that is, $brake = 1$. Actions 7 and 8 model the transitions from state *inactive* to *active* (or *ACC_active*). The triggering condition for these two actions is brakes are released and no leader car is detected within the predefined safety distance. Actions $9 - 11$ model the transitions from state *active*, *ACC_active* or *inactive* to state *off*($state = 4$). The triggering condition for these three actions is that the system is turned off.

6.3. Application of Step B: Generating Remaining Inputs for Model Revision

In this framework, the faults cause the sensor to provide an incorrect value. This can be modeled with Byzantine faults. Moreover, the number of occurrences of this fault is at most 1. And, the fault affects the leader variable from Figure 6.

Since the Byzantine fault affects at most one leader variable, we need a redundancy of three, i.e., we need variables, $leader1$, $leader2$ and $leader3$. Observe that since the redundancy is added for the leader variable, in the absence of faults, all leader values are equal. Hence, $leader1$ can be perturbed from such a state. Moreover, since at most one fault can occur, $leader1$ cannot be perturbed further. If two occurrences of faults were permitted, this would need to be changed so that $leader1$ could be perturbed even if some other (and only 1) leader variable were corrupted. Observe that by performing this analysis, it is possible to generate the guard that identifies when $leader1$ (respectively, $leader2$ and $leader3$) are corrupted. Based on this the fault actions can be modeled as follows:

1. $leader1 == leader2 == leader3 \longrightarrow leader1 := 0 [] leader1 := 1;$
2. $leader1 == leader2 == leader3 \longrightarrow leader2 := 0 [] leader2 := 1;$
3. $leader1 == leader2 == leader3 \longrightarrow leader3 := 0 [] leader3 := 1;$

where [] denotes the non-deterministic execution of statement.

We use an auxiliary variable car to denote whether there is a car in front of the current car. The value of the variable car is only included for modeling purpose. If the value of the sensors are not corrupted by fault, the value will be equal to the variable car. Based on the requirement of the ACC, transitions between *Active* and *ACC_active* must take the status of car into account. Moreover, this has to be done without utilizing the variable car in the revised program. Thus, the set of states the program should not reach are as follows:

$$((car == 1)\&(state'! = state)\&(on == 1)\&(brake == 0)\&(state'! = 2))|$$
$$((car == 0)\&(state'! = state)\&(on == 1)\&(brake == 0)\&(state'! = 1));$$

The invariant, that is, states where program should recover after fault occurs, is as follows. Note that the above predicate is generated automatically from the initial state of the UML model.

$$(((car == 1)\&(on == 1)\&(brake == 0)\&(state == 2)$$

$(((leader1 == 1)\&(leader2 == 1)\&(leader3 == 0))|$;
$((leader1 == 1)\&(leader3 == 1)\&(leader2 == 0))|$;
$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 0))|$;
$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 1))))$;
$(((car == 0)\&(on == 1)\&(brake == 0)\&(state == 1)$
$(((leader1 == 0)\&(leader2 == 0)\&(leader3 == 1))|$;
$((leader1 == 0)\&(leader3 == 0)\&(leader2 == 1))|$;
$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 1))|$;
$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 0))))$;

6.4. Application of Step C: Generation of Fault-Tolerant UCM

In Step C, we utilize the inputs from Sections 6.2 and 6.3. Since this step utilizes the tool SYCRAFT [17], we only provide the output of the synthesized program as follows: (Note that this output is only intended for use in Step D and not meant for designer to analyze it.)

$\mathcal{T}_o$:

$state = 1$ & $on = 0 \longrightarrow state := 4$;
$state = 2$ & $on = 0 \longrightarrow state := 4$;
$state = 3$ & $on = 0 \longrightarrow state := 4$;

$\mathcal{T}_s$:

$state = 0$ & $on = 1$ & $leader1 = 0$ & $leader2 = 0 \longrightarrow state := 1$;
$state = 0$ & $on = 1$ & $leader1 = 0$ & $leader3 = 0 \longrightarrow state := 1$;
$state = 0$ & $on = 1$ & $leader2 = 0$ & $leader3 = 0 \longrightarrow state := 1$;
$state = 0$ & $on = 1$ & $leader1 = 1$ & $leader2 = 1 \longrightarrow state := 2$;
$state = 0$ & $on = 1$ & $leader1 = 1$ & $leader3 = 1 \longrightarrow state := 2$;
$state = 0$ & $on = 1$ & $leader2 = 1$ & $leader3 = 1 \longrightarrow state := 2$;
$state = 1$ & $leader1 = 1$ & $leader2 = 1 \longrightarrow state := 2$;
$state = 1$ & $leader1 = 1$ & $leader3 = 1 \longrightarrow state := 2$;
$state = 1$ & $leader2 = 1$ & $leader3 = 1 \longrightarrow state := 2$;

$$state = 2 \;\&\; leader1 = 0 \;\&\; leader2 = 0 \;\longrightarrow state := 1;$$
$$state = 2 \;\&\; leader1 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$$
$$state = 2 \;\&\; leader2 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader1 = 0 \;\&\; leader2 = 0 \;\longrightarrow state := 1;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader1 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader2 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader1 = 1 \;\&\; leader2 = 1 \;\longrightarrow state := 2;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader1 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$$
$$state = 3 \;\&\; brake = 0 \;\&\; leader2 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$$

The action set $\mathcal{T}_o$ denotes the original actions which are from the original program and unchanged in the fault-tolerant program. The actions in $\mathcal{T}_s$ utilize the redundancy of sensors to tolerate the false alarm (false positive and false negative) caused by an unreliable sensor. Hence the system can get correct information about whether the leader car exists, even in the presence of the false alarm of one sensor.

6.5. Application of Step D: Generating Fault-tolerant UML model for ACC System

In this step, we utilize the fault-tolerant UCM into the corresponding UML state diagram. Observe that some parts of the UML state diagram remain unchanged. For those, we utilize the corresponding part from the UML state diagram. As an example, this results in that three transitions in the UML state diagram remain unchanged in the fault-tolerant UML state diagram, that is, the transition that is from state 1 to state 4, the transition that is from state 2 to state 4 and the transition that is from state 3 to state 4.

Moreover, for revised actions, we strengthen the conditions under which the actions can be executed. For example, the triggering condition of transition from state 0 to state 1 is revised from $on = 1 \& leader = 0$ to $on = 1 \& ((leader1 = 0 \& leader2 = 0) | (leader1 = 3 \& leader2 = 0) | (leader1 = 0 \& leader3 = 0))$. Thus, the UML state diagram of fault-tolerant ACC system is shown in Figure 7.

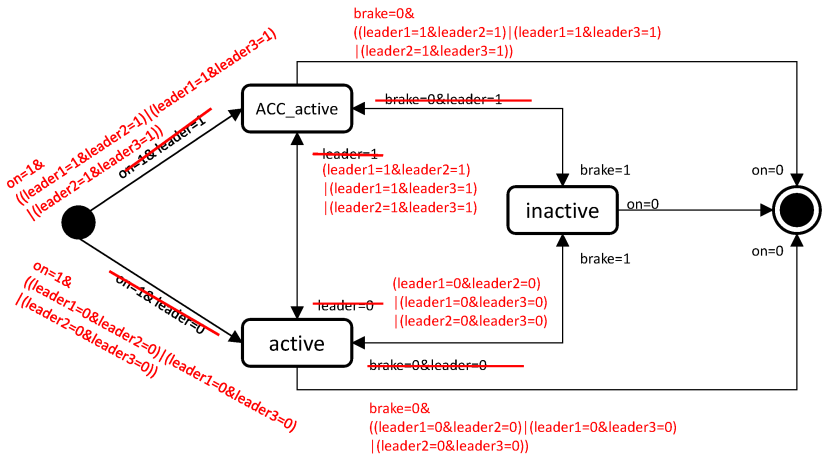

Figure 7: The Revised ACC program in UML State Diagram.

7. Case Study 2: The Altitude Switch Controller

In this section, we illustrate the work-flow of MR4UM with a simplified version of an altitude switch (ASW) program from the aircraft altitude controller system. This program is adapted from [18]. The ASW program monitors a set of input variables coming from two analog altitude sensors and a digital altitude sensor. And, it generates an output and activates an actuator when the altitude is less than a pre-determined threshold. In our case study, MR4UM begins with the fault-intolerant ASW program in UML state diagram. And it generates the fault-tolerant program that satisfies Problem 4.1.

This case study is organized as follow. Section 7.1 describes the fault-intolerant ASW program in UML state diagram. In Section 7.2, we describe how MR4UM transforms the ASW program in UML state diagram into program actions modeled in UCM. Section 7.3 describes how MR4UM generates fault actions, specification and invariant from the user's inputs. Next, Section 7.4 describes how MR4UM utilizes model revision algorithm to add fault tolerance to the input program. Finally, Section 7.5 shows how MR4UM transforms the fault-tolerant program in UCM into the corresponding fault-tolerant UML state diagram.

7.1. Fault-intolerant ASW Program in UML State Diagram

The UML state diagram of fault-intolerant *ASW* program includes three state, namely, *initialization*, *awaitActuator* and *standby*. The *initialization* state captures the status when the *ASW* program is initializing. The *awaitActuator* state captures the status when the *ASW* program is waiting for the actuator to power on. The *standby* state captures the status when the *ASW* program is in "standby" mode.

The state diagram in Figure 8 visualizes model design of *ASW* program. Figure 9 provides formal expression of the corresponding annotation in Figure 8. As shown in Figure 8, *ASW* program enters into the *standby* state after initialization from *initialization* state, and, the flag variable *init* is set to be 1 to denote the initialization process is done. *ASW* program resets into the *initialization* state from *standby* state if the reset process is triggered, and, the flag variable *reset* is assigned 1 to denote the reset process is done. When the actuator is powered off and actuator power-on is allowed, *ASW* program switches state from *standby* to *awaitActuator*, and, the flag variable *altBelow* is assigned 1 to denote that the altitude is below a specific threshold. *ASW* program switches from *awaitActuator* state to *standby* state if the actuator is powered on, and, the flag variable *actuatorStatus* is assigned 1 to denote the actuator is powered on. *ASW* program switches from *awaitActuator* state to *initialization* state if the program is reset, and, the flag variable *reset* is assigned 1 to denote that the program is reset. *ASW* program works correctly in the absence of faults.

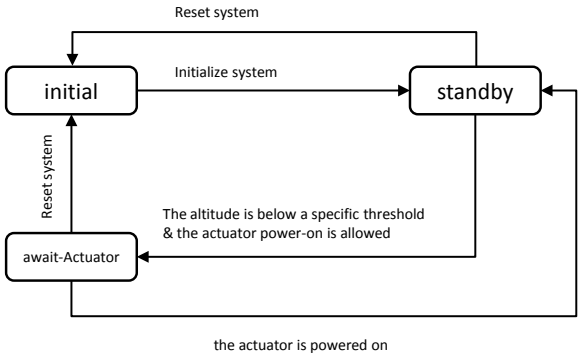

Figure 8: UML State Diagram of ASW Program.

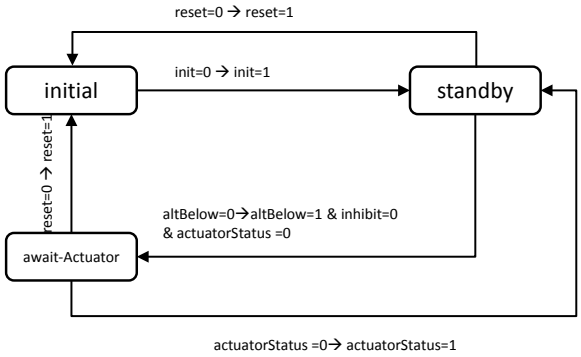

Figure 9: UML State Diagram of ASW Program with Annotation in Formal Expression.

7.2. Application of Step A: Generating UCM of the ASW Program

The UML state diagram of the ASW program is as shown in Figure 9. Based on the transformation discussed in Section 5, MR4UM introduces seven variables, namely, $state$, *init,reset, inhibit, altBelow*, *actuatorStatus* and *altFail* when generating the program actions in UCM. These variables are defined as follows:

1. *state*. The domain of variable *state* is $\{0, 1, 2, 3\}$. It is used to denote the four states of the ASW program. $state = 0$ denotes the initial status of the system when it is turned on. $state = 1$ denotes the system is in *Await-Actuator* status. $state = 2$ denotes the system is in *standby* status. Plus, $state = 3$ denotes the system is in *faulty* status.
2. *init*. The domain of variable *init* is $\{0, 1\}$. It is used to denote whether the altitude controller program is initialized. When the program is initialized, the variable *init* is assigned 1, otherwise 0.
3. *reset*. The domain of variable *reset* is $\{0, 1\}$. It is used to model whether the program is being reset. When the program is reset, the variable *reset* is assigned 1, otherwise 0.
4. *inhibit*. The domain of variable *inhibit* is $\{0, 1\}$. It is used to model whether the actuator power-on is inhibited. $inhibit = 1$ denotes that the actuator power-on is inhibited. And, $inhibit = 0$ denotes the actuator power-on is allowed.
5. *altBelow*. The domain of variable *brake* is $\{0, 1\}$. It is used to denote whether the altitude is less than a pre-determined threshold. $altBelow = 1$ models the

event that the altitude is below a specific threshold. $altBelow = 0$ models the event that the altitude is not below the threshold.

6. *actuatorStatus*. The domain of variable *actuatorStatus* is $\{0, 1\}$. It is used to model whether the actuator is powered on. $actuatorStatus = 1$ denotes that the actuator is powered on. And, $inhibit = 0$ denotes the actuator is not powered on.
7. *altFail*. The domain of variable *brake* is $\{0, 1\}$. *altFail* is equal to 1 when analog and digital altitude meters fail, otherwise 0.

Hence, the program actions of ASW program in UCM are as follows:

1. $state = 0$ & $init = 1 \longrightarrow state := 2; init := 0$;
2. $state = 2$ & $reset = 0 \longrightarrow state := 0; reset := 1$;
3. $state = 2$ & $altBelow = 0$ & $inhibit = 0$ & $actuatorStatus = 0 \longrightarrow state := 1; altBelow := 1$;
4. $state = 1$ & $actuatorStatus = 0 \longrightarrow state := 2; actuatorStatus := 1$;
5. $state = 1$ & $reset = 0 \longrightarrow state := 0; reset := 1$;

In the above program actions, Action 1 models the transition that causes program from *initialization* mode ($state = 0$) to *standby* mode ($state = 2$). This transition triggers the reassignment of the variable $init$, that is $init = 0 \rightarrow init = 1$ (which denotes the program finishes initialization process). Action 2 models the transition from *standby* mode to *initialization* mode ($state = 0$). This transition triggers the reassignment of the variable $reset$, that is, $reset = 0 \rightarrow reset = 1$ (which denotes the program is done with the resetting process). Action 3 denotes the transition from *standby* mode ($state = 2$) to *await-Actuator* mode ($state = 1$). The triggering condition of this action is the actuator power-on is not inhibited and the actuator is not powered on. This action triggers the reassignment of the variable $altBelow$, that is $altBelow = 0 \rightarrow altBelow = 1$ (which denotes that the altitude is below a specific threshold). Action 4 denotes that transition from *await-Actuator* ($state = 1$) mode to *standby* mode ($state = 2$). This action triggers the reassignment of the variable $actuatorStatus$, that is $actuatorStatus = 0 \rightarrow actuatorStatus = 1$ (which

denotes that the actuator is powered on). Action 5 denotes the transition from *await-Actuator* ($state = 1$) to *initialization* mode ($state = 0$). This action triggers the reassignment of the variable $reset$, that is $reset = 0 \rightarrow reset = 1$ (which denotes the system is reset).

7.3. Application of Step B: Generating Remaining Inputs for Model Revision

The targeted fault-tolerant program is required to tolerate such a faulty status: the altitude sensors incur malfunction. This type of fault is recognized as *transient* fault. To model this fault action in UCM, the designer needs to specify the effect of faults, that is which variable is perturbed and the possible value of the corrupted variable. In this case, the designer specifies the fault may corrupt variable $state$ into 3, that is, $state = 3$ denotes the faulty status of the program. Besides, the designer also needs to specify the triggering condition of the faults. In this case, the designer specifies three triggering conditions for three different transitions that corrupt the program into faulty status.

- *initFailed = 1.* This condition denotes the situation where the program stays in the initialization mode for more than 0.6 second.
- *altFailOver = 1.* This condition denotes the situation where the condition $altFail = 1$ remains true more than 2 seconds.
- *awaitOver=1.* This condition represents the situation where the program stays in the *await-Actuator* mode for more than 2 seconds.

Hence, MR4UM generates fault actions as following:

1. $initFailed == 1 \longrightarrow initFailed := 0, state := 3;$
2. $altFailOver == 1 \longrightarrow altFailOver := 0, state := 3;$
3. $awaitOver == 1 \longrightarrow awaitOver := 0, state := 3;$

In this case, the designer requires the safety specification, as follows:

- If the altitude sensor fails, the program should not transfer from *standby* mode to *await-Actuator* mode;

- The program can only recover to the initialization mode from the faulty mode;
- The program can recover from the faulty mode if the program is not reset.

Hence, the designer specifies the specification from GUI of the framework as follows:

$((altFails == 1)\&(state == 2)\&(state' == 0))|$
$((state == 3)\&(state == 2)\&(state' == 1))|$
$((state == 3)\&(reset == 1));$

The invariant of the program consists of the states where the program is not in the faulty states, i.e., $state! = 3$.

7.4. Application of Step C: Generation of Fault-Tolerant UCM

After the first two steps, MR4UM obtains the inputs from Section 7.2 and 7.3. In this step, MR4UM utilizes model revision to generate the fault-tolerant program in UCM. The result from Step C is invisible to MR4UM users since this output is only intended for Step D and not meant for designer to analyze it. In particular, the fault-tolerant program for this case study obtained in Step C is as follows:

$\mathcal{T}_o$:

1). $state = 0$ & $init = 1 \longrightarrow state := 2; init := 0;$

2). $state = 2$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

3). $state = 1$ & $actuatorStatus = 0 \longrightarrow state := 2; actuatorStatus := 1;$

4). $state = 1$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

$\mathcal{T}_s$:

5). $state = 2$ & $altBelow = 0$ & $inhibit = 0$ & $actuatorStatus = 0$ & $altFail = 0$
$\longrightarrow state := 1; altBelow := 1;$

$\mathcal{T}_r$:

6). $state = 3$ & $reset = 0 \longrightarrow state := 0; reset := 1;$

We observe that a new constraint, $altFail == 0$, is added to Action 5. This change denotes that the revised program allows to change its state to the *await-Actuator*

mode only when the sensors are not corrupted. Action 6 is newly added to the revised program which denotes that the program recovers from faulty mode ($state = 3$) to the initialization mode ($state = 0$). The other actions remains the same as the actions of the input program.

7.5. Application of Step D: Generating Fault-tolerant UML model for ASW Program

In this step, MR4UM converts the fault-tolerant program in UCM into the corresponding model in UML state diagram. The UML state diagram of fault-tolerant ASW program is as shown in Figure 10.

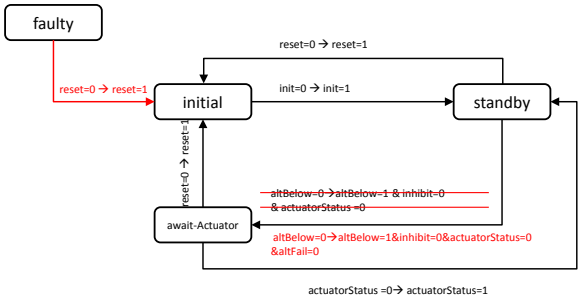

Figure 10: The Revised ASW Program in UML State Diagram.

8. Related work

Previous work in [19, 20, 21] addresses the problem of formalization of UML state diagram. Specifically, these approaches define operational semantics of the UML state diagram and then utilize it for simulation, verification and/or code generation. The first step in our framework (Section 5) is inspired by these approaches. However, unlike the previous work, in our work the translations of UML model needs to be annotated so that we can subsequently obtain a revised UML model after adding fault-tolerance. Another important difference between our work and these works is that our work focuses on the problem of model revision whereas they focus on the problem of model checking. Thus, our work is complementary to previous work in that our framework can be applied in scenarios where the given UML model fails to satisfy the given property.

The problem of model revision is closely related to the work on controller synthesis [22, 23, 24] and game theory [25, 26, 27]. In these works, supervisory control of real-time systems has been studied under the assumption that the existing program (called a *plant*) and/or the given specification is *deterministic*. Moreover, in both game theory and controller synthesis, since highly expressive specifications are often considered, the complexity of the proposed synthesis methods is very high. By contrast, our work focuses on two types of specifications (1) safety specifications that constrain the transitions that the program is allowed to take, and (2) liveness specifications that require the program recovers to its original behavior. For this reason, the complexity of the problem considered in our framework (P to NP-complete depending upon the specific problem formation) is considerably lower than that is considered in [22, 23, 24, 25, 26, 27] (up to 2EXPTIME-complete).

Our work is orthogonal to related work (e.g., [28, 29, 30, 31, 32]) that focuses on transforming an abstract UML model into a concrete (such as C++) program while ensuring that the location of concrete program in memory, its data flow etc. meet the constraints of the underlying system. In particular, our work focuses on revising the given model into another UML model that satisfies the fault-tolerance property. Thus, our work will advance the applicability of this existing work by allowing designers to add properties of interest in the abstract model and then using existing work to generate concrete program.

Approaches in [13, 33] develop corrector pattern for specifying nonmasking fault-tolerance and failsafe fault-tolerance respectively. These proposed analysis methods are validated in terms of UML diagrams. While these works simplify and modularize fault-tolerance concerns and facilitate to analyze the functional and fault-tolerance concerns and their mutual impact, application of a synthesis tool in automatically adding fault-tolerance is an on-going direction of these works. Our work utilizes the synthesis tool [17] to automate the revision process for adding fault-tolerance.

The work in [34] proposes an approach of automating and formalizing the translation from high level design models, specifically, Software Cost Reduction (SCR)[18], to a format that can be used by the automated revision/synthesis tools of example. SCR is a set of formal methods for constructing and verifying requirements specifica-

tion document. By contrast, our work focuses on issues of automatic revision of UML state diagram for adding fault-tolerance.

9. Discussion and Lessons Learnt

In this section, we discuss several questions that are raised by our work as well as lessons learnt from these case studies.

One question is the scalability of the approach of automatic revising the existing program design proposed in our paper. Our framework benefits from the underlying synthesis engine. This tool utilizes BDDs to mitigate the state explosion problem and a heuristic based algorithm [14] to mitigate the complexity (NP-complete) of model revision. Specifically, this approach has been used to permit model revision of programs with state space exceeding 10^{100}. Hence, we expect the framework to be able to handle moderate sized problems.

Another question is about the choice of UML as the front end for our framework. We chose UML because it is one of the commonly used platforms to specify requirements. And, although there is an existing work on formalization of UML models, the problem of model revision has not been addressed in this context. Our approach is also feasible for revising program design modeling in other approaches (e.g. AADL [35]) by modifying the mapping mechanism between the model of program design and underlying computational model. It has also been demonstrated in revising SCR specifications [36].

One of the difficulties in developing this framework lies in the fact that the revised fault-tolerant model in UCM is BDD based. Although converting the UCM model into UML state diagram involves some challenges, they can be overcome by understanding (1) the part of the UML model that will remain intact in the final model, (2) the part of the UML model where the structure of the original model will remain intact in the final model although some of the details (e.g., conditions on the arrows) will change, and (3) the part of the UML model that is completely new and added for dealing with recovery from faults. Since BDD based approaches permit us to check conditions (1) and (2) effectively with negligible cost, obtaining the revised UML state diagram is

feasible. One of the future works in this area is to optimize the third part that identifies the actions that provide recovery.

10. Conclusion

This paper focuses on lowering the learning curve required for application of formal methods, specifically model revision, by keeping the formal methods under-the-hood to a large extent. Specifically, we propose a framework, namely MR4UM, which allows designers to apply model revision to existing UML models that need to be revised to provide fault-tolerance. We chose to apply model revision for UML models since it is one of the commonly used modeling techniques.

One of the future work is to enrich our framework to utilize model revision to UML diagrams for adding other properties (e.g., safety properties, liveness properties and timing constraints). Another future work is to extend the types of faults that can be considered during the revision process. Also, the synthesis engine in [14] requires description of faults in UCM, the set of states to which the program should recover and the requirements that should be satisfied during recovery. Of these, the first two are automatically generated. While the third parameter, requirements in the presence of faults, cannot be automated, we intend to provide default features that identify the commonly used requirements.

References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (1992) 142–170.

[2] G. Holzmann, The spin model checker, IEEE Transactions on Software Engineering 23 (1997) 279–295.

[3] T. Ball, B. Cook, V. Levin, S. K. Rajamani, Slam and static driver verifier: Technology transfer of formal methods inside microsoft, in: In: IFM. (2004, Springer, 2004, pp. 1–20.

[4] K. Larsen, P.Pattersson, W. Yi, UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer 1 (1997) 134–152.

[5] E. A. Emerson, E. M. Clarke, Using branching time temporal logic to synchronize synchronization skeletons, Science of Computer Programming 2 (1982) 241–266.

[6] O. Maler, D. Nickovic, A. Pnueli, From MITL to timed automata., in: Formal Modeling and Analysis of Timed Systems (FORMATS), pp. 274–289.

[7] R. Alur, T. Feder, T. Henzinger, The benefits of relaxing punctuality., Journal of the ACM 43 (1996) 116–146.

[8] A. Arora, P. C. Attie, E. A. Emerson, Synthesis of fault-tolerant concurrent programs, Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC) (1998).

[9] P. Attie, A. Emerson, Synthesis of concurrent programs for an atomic read/write model of computation, ACM TOPLAS 23 (2001).

[10] O. Kupferman, M. Y. Vardi, Synthesizing distributed systems, in: Logic in Computer Science, pp. 389 – 398.

[11] B. Bonakdarpour, Automated Revision of Distributed and Real-Time Programs, Ph.D. thesis, Michigan State University, 2008.

[12] J. Rumbaugh, I. Jacobson, B. G., The Unified Modeling Language Reference Manual, Pearson Higher Education, 2004.

[13] A. Ebnenasir, B. H. C. Cheng, Pattern-based modeling and analysis of failsafe fault-tolerance in uml, in: Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium, HASE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 275–282.

[14] B. Bonakdarpour, S. Kulkarni, Exploiting symbolic techniques in automated synthesis of distributed programs, in: Proceedings of In IEEE International Conference on Distributed Computing Systems(ICDCS), ICDCS '07, Toronto, Canada, pp. 3–10.

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

[16] J.-C. Laprie, Dependable computing and fault tolerance: Concepts and terminology, Proceedings of the 15th International Symposium on Fault-Tolerant Computing (1985) 2–11.

[17] B. Bonakdarpour, S. Kulkarni, Sycraft: A tool for automated synthesis of fault-tolerant distributed programs, in: Proceedings of International Conference on Concurrency Theory (CONCUR), Toronto, Canada, pp. 167–171.

[18] R. Bharadwaj, C. Heitmeyer, Developing high assurance avionics systems with the scr requirements method, in: Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, PA, pp. 1D1/1 – 1D1/8.

[19] J. Lilius, I. P. Paltor, Formalising uml state machines for model checking, in: UML'99 Proceedings of the 2nd international conference on The unified modeling language: beyond the standard, pp. 430–444.

[20] A. Knapp, S. Merz, Model checking and code generation for uml state machines and collaborations, in: In Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, Proc. 5th Wsh. Tools for System Design and Verification, pp. 59–64.

[21] T. A. J. Jori Dubrovin, Symbolic model checking of hierarchical uml state machines, in: ACSD: 8th International Conference on Application of Concurrency to System Design, pp. 108 – 117.

[22] P. Bouyer, D. D'Souza, P. Madhusudan, A. Petit, Timed control with partial observability., in: Computer Aided Verification (CAV), pp. 180–192.

[23] D. D'Souza, P. Madhusudan, Timed control synthesis for external specifications., in: Symposium on Theoretical Aspects of Computer Science (STACS), pp. 571–582.

[24] E. Asarin, O. Maler, As soon as possible: Time optimal control for timed automata, in: Hybrid Systems: Computation and Control (HSCC), pp. 19–30.

[25] B. Jobstmann, A. Griesmayer, R. Bloem, Program repair as a game, in: Computer Aided Verification (CAV), pp. 226–238.

[26] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, M. Stoelinga, The element of surprise in timed games, in: International Conference on Concurrency Theory (CONCUR), pp. 144–158.

[27] M. Faella, S. LaTorre, A. Murano, Dense real-time games., in: Logic in Computer Science (LICS), pp. 167–176.

[28] D. de Niz, R. Rajkumar, Glue code generation: Closing the loophole in model-based development, 2nd RTAS Workshop on Model-Driven Embedded Systems (2004).

[29] A. F. Martinez, K. Kuchcinski, Graph matching constraints for synthesis with complex components, in: DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, IEEE Computer Society, Washington, DC, USA, 2007, pp. 288–295.

[30] Z. Gu, S. Wang, K. G. Shin, Synthesis of real-time implementation from uml-rt models, 2nd RTAS Workshop on Model-Driven Embedded Systems (2004).

[31] P.-A. Hsiung, S.-W. Lin, Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems, Comput. Lang. Syst. Struct. 34 (2008) 153–169.

[32] S.-W. Lin, S.-W. Lin, C.-H. Tseng, T.-Y. Lee, J.-M. Fu, Vertaf: An application framework for the design and verification of embedded real-time software, IEEE Trans. Softw. Eng. 30 (2004) 656–674. Member-Pao-Ann Hsiung and Member-Win-Bin See.

[33] A. Ebnenasir, B. H. C. Cheng, A pattern-based approach for modeling and analyzing error recovery, in: Workshops on Software Achitectures for Dependable systems (WADS), pp. 115–141.

[34] F. Abujarad, S. S. Kulkarni, Automated addition of fault-tolerance to scr toolset: A case study, in: The Seventh International Workshop on Assurance in Distributed Systems and Networks (ADSN), in ICDCSW '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems Workshops, pp. 539–544.

[35] P. Feiler, B. Lewis, S. Vestal, The sae architecture analysis and design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering, in: In Proceedings of the RTAS Workshop on Model-driven Embedded Systems, pp. 1–10.

[36] C. Heitmeyer, M. Archer, R. Bharadwaj, R. Jeffords, Tools for constructing requirements specifications: The scr toolset at the age of ten, in: International Journal of Computer Systems Science and Engineering, pp. 19–35.