Application of Automated Revision for UML models: A Case Study * **

Jingshu Chen and Sandeep Kulkarni

Michigan State University,
3115 Engineering Building, 48824 East Lansing, US
Email: {chenji15,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{chenji15,sandeep}

Abstract. Modern systems often need to address changing environment and/or 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.
Our paper proposes a framework of automated revision of existing program design modeled in UML to add fault-tolerance. Our framework starts with program design modeled in UML state diagram, and then automatically transforms design model to the corresponding underlying computational model. Subsequently, automated revision algorithms are applied to the underlying computational model. Finally the revised program model is converted into an UML model that provides the desired fault-tolerance property. We illustrate the whole work-flow with a case study from automotive systems.

Key words: Fault Modeling, Model-based Design, Model revision, Fault Tolerance.

1 Introduction

The utility of formal methods in the development of high assurance systems has gained widespread use in some segments of industry. With the use of formal methods, there are two main approaches for providing assurance. The first approach, *correct-by verification*, is the most commonly used approach. In this approach, one begins with an existing model (or program) and a set of properties (specification), and verifies that the given program meets the given properties of interest. An embodiment of this approach is *model-checking* [5, 10, 18, 24] and has been widely studied in the literature. However, a pitfall of this approach is that if the manually designed model does not satisfy the requirements then it is often unclear how one can proceed further. Hence, for scenarios where an existing model needs to be revised to deal with the new environment, the new identified faults, or the new requirements, one needs to manually develop the new model if one needs to obtain assurance via modeling checking.

* We would like to thank Shige Wang (General Motors) for providing the UML model for the cruise control system that was used in this case study
** This work is sponsored by USA AFOSR FA9550-10-1-0178 and NSF CNS 0914913 grants.

The second approach, *correct-by-construction*, utilizes the specification of the desired system and constructs a model that is correct. Examples of this approach include [1, 2, 4, 15, 23, 27]. These approaches differ in terms of the expressiveness of the specifications they permit and in terms of their complexity. However, a pitfall of this approach is the loss of reuse (of the original model) and 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*. Model revision deals with the problem where one is given a model/program and a property. And, the goal is to revise the model such that the given property is satisfied. Applications of model revision include scenarios where model checking concludes that the given property is not satisfied. Other applications include scenarios where an existing model needs to be revised due to changes in requirements and/or changes in the environment. For this reason, model revision has been studied in contexts where an existing model needs to be revised to add new fault-tolerance properties, safety properties, liveness properties and timing constraints [6].

Since model revision results in a model that is correct-by-construction, it provides assurance comparable to *correct-by-construction* approaches. Also, since it begins with an existing model, it has the potential to provide reuse during the revision process. Moreover, there are several instances where complexity of model revision is comparable to that of model checking [6].

Based on these observations, our goal in this paper is to illustrate the application of model revision in the context of a case study from automotive systems. One problem in applying formal methods in industrial systems is the high learning curve encountered in formal methods. One approach to reduce the learning curve is to utilize an existing framework utilized by designers and allow formal methods to be hidden under the hood.

One challenge of exploring such an approach of applying the model revision is that the current model-based design is often modeled in a noncomputatonal way, such as UML. UML [29] is a well-known modeling language utilized by 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 with the need of independence from a particular programming language. Since the UML state diagram is able to illustrate the high-level overview of the whole system, it is widely used to model program design. With this motivation, the approach in our paper starts with program design modeled in the UML state diagram.

To overcome the challenge of applying model revision on program design modeled in UML state diagram, that is, model in UML state diagram is not computational one, our approach proposes an automatic transformation mechanism from model in UML state diagram to the underlying computational model. Subsequently, the model revision algorithms are applied on the program modeled in the corresponding underlying computational model. Finally, the revised program modeled in the underlying computational way is converted into program design modeled in UML state diagram with fault-tolerance properties. We illustrate the whole work-flow with a case study from automotive systems- the adaptive cruise control system.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we illustrate the proposed approach by presenting a motivating scenario from design issues in automotive system. In Section 3, we briefly discuss the related concepts of modeling program design and introduce the underlying computational model. Next, in Section 4, we describe the work-flow of the proposed framework step by step. Related work is discussed in Section 5. Finally Section 6 makes concluding remarks.

2 Motivating Scenario

This section presents a motivating scenario in designing automotive system to demonstrate our approach. Figure 1 describes logic design of an Adaptive Cruise Control (ACC) system. The ACC system is comprised of a cruise control system and a sensor system. This system is designed to control the distance between the vehicle and the front vehicle (called leader car) automatically.

As shown in Figure 1, when ACC system is on after initialization, the system can be in one of the three modes: *active*, *ACC_active* or *inactive*. In the *active* mode, the sensor system keeps checking whether there is a leader car appearing within a predefined safe range and the cruise control system keeps checking the sensor result. In the *ACC_active* mode, the sensor system has detected the existence of leader car and the ACC system was notified by the sensor. In the *inactive* mode, the driver is pressing the brakes and the adaptive cruise control relinquishes control to driver for manual control without considering whether the leader car exists. By switching among these three modes, the ACC system targets to control the distance between two cars in the safe distance and keep the current car at a zero relative speed with respect to the leader car. Initially, when there is no leader car detected by the system, the system remains in *active* mode. If a car is detected by the sensor system within the predefined safe distance, the ACC system enters into *ACC_active* mode. ACC system remains staying in *ACC_active* mode until the leader car goes away. If the leader car goes 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 there is no leader car and *ACC_active* mode if a leader car exists.

2.1 Need for Model Revision for Tolerating Sensor Failure

While the adaptive cruise system in Figure 1 operates correctly in the absence of faults, it results in undesired behavior when faults affect the sensor. Specifically, a sensor failure can 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 present 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 an unreliable sensor, one typical fault tolerance policy is to provide redundancy. So 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.

To solve this problem in the above scenario, we propose an automatic approach in this paper. Our approach starts with the design modeled in the UML state diagram. Then the design model in UML state diagram is translated to the corresponding underlying computational model automatically. This translation is annotated to facilitate the given actions of reused UML model in the last step. Based on the underlying computational model, an automatic revision algorithm is applied to get the targeted program in the underlying computational model. Finally, the annotations from the first step are used to translate the revised underlying computation model into an UML model.

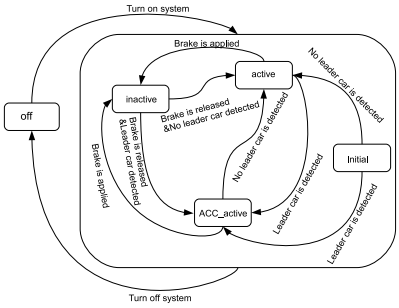

Fig. 1: Logic Design of ACC System.

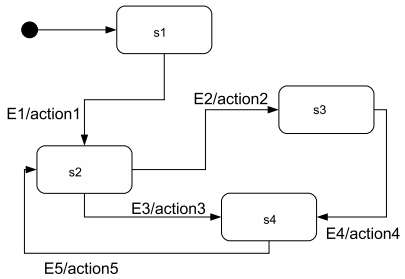

Fig. 2: A case to illustrate modeling program in UML state diagram.

3 Modeling

In this section, we first present basic concept of modeling program design with/without fault tolerance in UML state diagram. Then, we introduce underlying computational model of the program, which is later used in automated revision process. We also discuss the approach of modeling faults.

3.1 Program Design in UML State Diagram

In this section, we describe how one can specify UML state diagrams. There are several advantages of modeling program design in the UML state diagram. The advantages include: 1) UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. 2) UML state diagram enable us to visualize the

program design. 3) UML state diagram enables us to capture any form of fault-tolerance that can be expressed in a state machine-based formalism [14].

UML state diagram is visualized in terms of its *states* and *transitions*, where:

1. **State.** Generally, the state is represented as rounded rectangle. Specially, the initial state(if any) is denoted as filled circle. The final state (if any) is denoted as a hollow circle containing a smaller filled circle.
2. **Transition.** The transition is represented with an arrow. We also denote the trigger event of the transition as an annotation above the arrow in the UML state diagram.

As an illustration, consider the fragment in Figure 2. In this fragment, there are four states: $s1$, $s2$, $s3$ and $s4$. There are five transitions, as follows:

1. $s1 \rightarrow s2$**.** When trigger event $E1$ occurs, $action1$ changes the state from $s1$ to $s2$;
2. $s2 \rightarrow s3$**.** When trigger event $E2$ occurs, $action2$ changes the state from $s2$ to $s3$;
3. $s2 \rightarrow s4$**.** When trigger event $E3$ occurs, $action3$ changes the state from $s2$ to $s4$;
4. $s3 \rightarrow s4$**.** When trigger event $E4$ occurs, $action4$ changes the state from $s3$ to $s4$;
5. $s4 \rightarrow s2$**.** When trigger event $E5$ occurs, $action5$ changes the state from $s4$ to $s2$.

Application in the Case Study: 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. Figure 4 gives the state diagram of the design of the ACC system.

The state diagram in Figure 3 visualizes model design of the ACC system. For better understanding, Figure 4 labels out formal expression of the corresponding annotation. As shown in Figure 3, when the ACC system is turned on, the system will enter into the *active* state after initialization if there is no leader car detected. The system will enter into the *ACC_active* state if the leader car exists according to the information from sensor system. In other words, existence (or nonexistence) of leader car is the trigger condition to change system state between *active* and *ACC_active*. When the brake is applied, irrespective whether it were in *active* or *ACC_active*, the ACC system enters into *inactive* state. When the brake is released, the system will change from *inactive* state into *active* state or *ACC_active* state depending upon the existence of leader car. The whole ACC system will continue to stay in one of these three states until the system is turned off.

3.2 Underlying Computational Model(UCM)

In this section, we describe the underlying computational model (UCM) that is used during the revision process. We convert the model described in Section 3.1 to UCM for

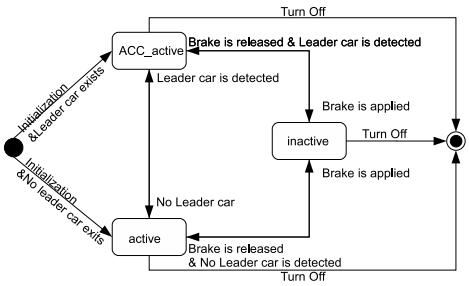

Fig. 3: ACC System Modeled in UML State Diagram.

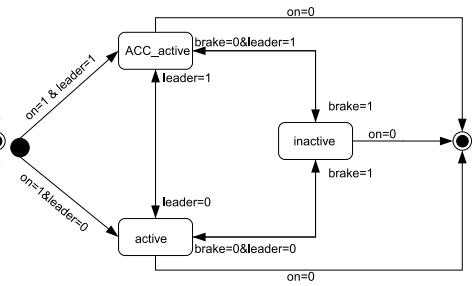

Fig. 4: Annotation in Formal Expression.

the revision process. The UCM is adapted from [7], hence we can utilize the synthesis engine proposed in [7].

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 the 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

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

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* $S_{\mathcal{P}} \times S_{\mathcal{P}}$*.* □

Application in the Case Study (cont'd): The details of modeling the ACC system in UCM are presented in Section 4.

---

4 Framework Description

This section describes the workflow of our framework step by step. First we illustrate how we translate the UML-based system design into the underlying computational model (UCM). Then, we describe how to solve synthesis problem in UCM. Next we introduce how we revise the UML state diagram based on the result from the UCM.

4.1 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 is processed automatically. The details are as follows:

1. For all the states in the UML state diagram, we introduce variable STATE with integer domain $[0, n-1]$ where n is the number of states in the UML. All the states in the UML state diagram are numbered from 0 to n-1. For each state, it is mapped to concrete value assignment of variable STATE. For example, for state 0 in the UML state diagram, it is mapping into STATE==0 .
2. For each trigger condition c mentioned in the UML state diagram, we introduce one variable X_c with domain $\{0, 1\}$. $X_c = 0$ denotes the negative of this trigger condition is satisfied. $X_c = 1$ denotes this trigger condition is satisfied.
3. 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 changes the assignment of STATE according to the target state of the original transition in the UML.

Application in the Case Study (cont'd): The UML model for the ACC system is as shown in Figure 4. Based on the transformation discussed above, the corresponding UCM needs four variables, namely *state*, *on*, *brake* and *leader*. The details of these variables are as follows:

1. *state*.The range of variable *state* is $[0, 4]$. It is used to model 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 range 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. $on = 0$ models the trigger condition that causes the ACC system enters into the stop status.
3. *leader*. The range of variable *leader* is $[0, 1]$. It is used to model whether there is leader car detected by the sensor system.
4. *brake*. The range 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.

Based on the transformation discussed above, the program actions of 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 is represented by the remaining part of the guard condition of the action, that is $on = 1 \& leader = 0$ (which denotes there is no leader car detected when system is turned on). Action 2 models the transition from *initial* state to *ACC_active*($state = 2$) with the triggering event, that is, there is leader car detected when the system is turned on. Action 3 and 4 model the transitions between state *active* and *ACC_active* with the trigger events, that is whether there is leader car detected. Action 5 and 6 models the transition from *active*or*ACC_active* to *inactive* with the triggering event, that is, brakes are applied by driver ($brake = 1$). Action 7 and 8 model the transitions from state *inactive* to *active* (or *ACC_active*) with the triggering event, that is, brakes are released and there is no leader car (or there is leader car). Action $9 - 11$ models the transitions from state *active*, *ACC_active* or *inactive* to state *off*($state = 4$), with the triggering conditions, that is, the system is turned off.

4.2 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 (1) fault actions modeled in UCM, (2) specification, that is, requirements in the presence of faults, and (3) states where program should recover after faults occurs. Next, we specify how we obtain these parts in our framework.

1. **Fault Actions Modeled in UCM.** In our framework, *faults actions* are automatically generated from parameters which are specified by designer from GUI. From GUI, designer needs to specify the following parameters:
   (a) *What type of faults?* Currently, there are three types of faults modeled in our framework: (1) byzantine, (2) transient and (3) crash (failstop). The default setting of our framework is *transient*.
   (b) *What these faults do?* Designer needs to specify the variables the fault affects after specifying types of faults that may occur during the execution:

i. *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). The default for this fault is that the variable can be corrupted to any value in its domain.
ii. *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.
iii. *Crash or Failstop.* For this type of fault, designer needs to specify which variables are prevented from access due to the fault.

(c) *Occurrences of faults?*Designer also need to specify the occurrences of the specified faults. 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.

Application in the Case Study (cont'd): Since three redundant sensors are needed for one byzantine sensor, our framework will declare failure if less than three were available. Hence, we assume availability of three sensors. Since only one can be faulty, the framework generates the fault actions automatically (based on the approach of modeling fault in Section 3) as follows:

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

We define [] as non-deterministic execution of statement. Take action 1 as an example. If the guard condition $leader1 == leader2 == leader3$ is satisfied, the action may change $leader1$ to be 0 or 1. We use an auxialiary 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. The specification of this case study is:

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

The invariant, that is, states where program should recover after fault occurs, is as follows. Note that the above predicate is generated automatically.

$$(((car == 1)\&(on == 1)\&(brake == 0)\&(state == 2)$$
$$(((leader1 == 1)\&(leader2 == 1)\&(leader3 == 0))|;$$
$$((leader1 == 1)\&(leader3 == 1)\&(leader2 == 0))|;$$
$$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 0))|;$$
$$((leader2 == 1)\&(leader3 == 1)\&(leader1 == 1))));$$
$$(((car == 0)\&(on == 1)\&(brake == 0)\&(state == 1)$$

$(((leader1 == 0)\&(leader2 == 0)\&(leader3 == 1))|;$
$((leader1 == 0)\&(leader3 == 0)\&(leader2 == 1))|;$
$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 1))|;$
$((leader2 == 0)\&(leader3 == 0)\&(leader1 == 0))));$

4.3 Step C: Generate Fault Tolerant Program Modeled in UCM- Automatically revise the fault-intolerant program into the fault-tolerant one.

During this step, we apply the symbolic algorithm that revises the fault-intolerant program modeled in UCM automatically into the corresponding fault tolerant version [7]. The algorithm takes an intolerant program, a safety specification, and a set of fault transitions as input and synthesizes a fault-tolerant program.

INPUT:ψ_P:transitions, f:fault transitions, I_P: invariant predicate, $spec$: safety specification
OUPUT:$\psi_{P'}$: transitions of fault tolerant program and $I_{P'}$: 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)
$I_1, fte := I_P - ms, false;$	(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)
$\quad\quad\quad S_2 := S_1;$	(9)
$\quad\quad\quad S_1 := FWReachStates(I_1, \psi_1 \vee f);$	(10)
$\quad\quad\quad S_1 := S_1 - fte;$	(11)
$\quad\quad\quad mt := mt \wedge S_1;$	(12)
$\quad\quad\quad \psi_1 := \psi_1 - Group(\psi_1 \wedge mt);$	(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)
$I_P, \psi_{P'} := I_1, \psi_1;$	(22)
RETURN $I_{P'}, \psi_{P'}$;	(23)

Fig. 5: Add_FT from [7]

We reproduce the algorithm in [7] in Figure 5 and next we provide a short summary of it. This algorithm consists of five steps, as following:

1. **Initialization (Lines 1-3).** In this step, we identify state and transition predicates from where execution of faults alone may violate safety specification.
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 unsafe transition, that is, transitions whose execution may lead to violation of the safety specification.
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.
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 fixpoints in Line 14, 19 and 20 are reached. The algorithm terminates when no progress is possible in all the steps.

Application in the Case Study (cont'd): The revised actions got from Step C are presented as follows. Note that this output is only intended for use in Step D and not meant for designer to analyze it.

1. $state = 0 \;\&\; on = 1 \;\&\; leader1 = 0 \;\&\; leader2 = 0 \;\longrightarrow state := 1;$
2. $state = 0 \;\&\; on = 1 \;\&\; leader1 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
3. $state = 0 \;\&\; on = 1 \;\&\; leader2 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
4. $state = 0 \;\&\; on = 1 \;\&\; leader1 = 1 \;\&\; leader2 = 1 \;\longrightarrow state := 2;$
5. $state = 0 \;\&\; on = 1 \;\&\; leader1 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$
6. $state = 0 \;\&\; on = 1 \;\&\; leader2 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$
7. $state = 1 \;\&\; leader1 = 1 \;\&\; leader2 = 1 \;\longrightarrow state := 2;$
8. $state = 1 \;\&\; leader1 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$
9. $state = 1 \;\&\; leader2 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$
10. $state = 2 \;\&\; leader1 = 0 \;\&\; leader2 = 0 \;\longrightarrow state := 1;$
11. $state = 2 \;\&\; leader1 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
12. $state = 2 \;\&\; leader2 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
13. $state = 3 \;\&\; brake = 0 \;\&\; leader1 = 0 \;\&\; leader2 = 0 \;\longrightarrow state := 1;$
14. $state = 3 \;\&\; brake = 0 \;\&\; leader1 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
15. $state = 3 \;\&\; brake = 0 \;\&\; leader2 = 0 \;\&\; leader3 = 0 \;\longrightarrow state := 1;$
16. $state = 3 \;\&\; brake = 0 \;\&\; leader1 = 1 \;\&\; leader2 = 1 \;\longrightarrow state := 2;$
17. $state = 3 \;\&\; brake = 0 \;\&\; leader1 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$
18. $state = 3 \;\&\; brake = 0 \;\&\; leader2 = 1 \;\&\; leader3 = 1 \;\longrightarrow state := 2;$

The above actions 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.

4.4 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. First, we utilize the *revised* program actions and *recovery* program actions to identify the *changed* transitions in the original UML state diagram.

2. Second, we re-annotate these transitions in the UML state diagram by the guard conditions of these *revised* program actions and *recovery* program actions.

Application in the Case Study (cont'd): The UML state diagram of fault-tolerant ACC system is shown in Figure 6.

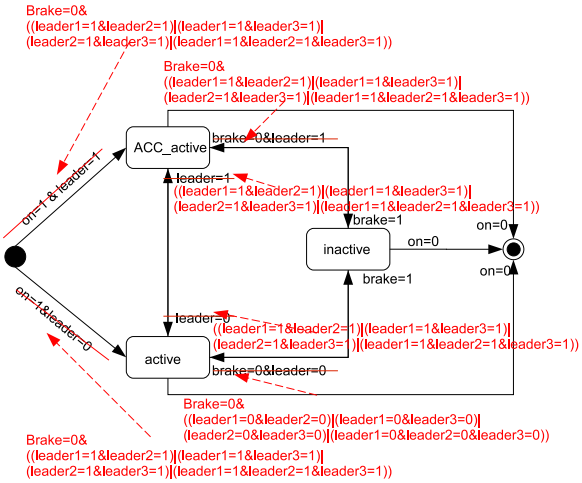

Fig. 6: A Case: The Revised Program in UML state diagram.

5 Related work

Previous work that has focused on formalization of UML state diagram include [21, 22, 25]. Specifically, these approaches first 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 4) 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 the work in [21, 22, 25] 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 work on model revision is closely related to work on controller synthesis [3, 9, 13] and game theory [11, 16, 20]. In this work, 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, in our work, we focus on two types of specifications (1) safety specifications that constrain the transitions that the program is allowed to take, and (2) liveness requirements that require the program to recover 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 considered in [3, 9, 11, 13, 16, 20] (up to 2EXPTIME-complete).

Our work is orthogonal to related work (e.g., [12, 17, 19, 26, 28]) 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.

6 Conclusion

This paper focuses on reducing 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, this work 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.

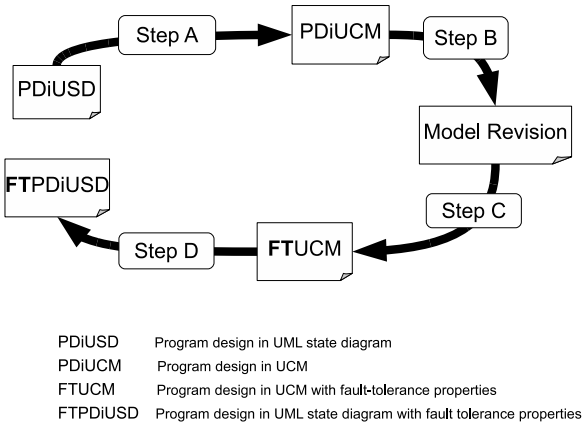

Fig. 7: Work-flow of the Framework.

In our work, we begin with a fault-intolerant version of the UML model. Our framework allows the designer to specify common types of faults (byzantine, crash or transient) as well as an expectation about the occurrences of those faults. Based on these inputs, our framework generates the underlying computational model (that is annotated) and utilizes the model revision tools from [8] to add fault-tolerance. This results in a fault-tolerant version in the underlying computational model. Subsequently, we use the annotations from the first step are used to generate the revised UML model. The workflow of the whole framework is shown in Figure 7. Finally, since our framework is based on BDDs, it has the potential to deal with large state space, e.g., the underlying synthesis engine has been used to revise models with state space exceeding 10^{100}.

We illustrated our framework with a case study from automotive system. In particular, we began with the fault-intolerant version of the UML model for adaptive cruise controller system. We considered the fault that caused false positive and/or false negative sensor readings. Subsequently, we used the framework to obtain the corresponding fault-tolerant version.

One of the future work in this area is to extend this framework to add other properties (e.g., safety properties, liveness properties and timing constraints) to UML models. Another future work is to extend the types of faults that can be considered during the revision process. Also, the synthesis engine in [7] 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. R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
2. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.
3. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control (HSCC)*, pages 19–30, 1999.
4. P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS*, 23(2), March 2001.
5. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.
6. B. Bonakdarpour. *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University, 2008.
7. B. Bonakdarpour and S. 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, pages 3–10, Toronto, Canada, 2007.
8. B. Bonakdarpour and S. S. Kulkarni. Sycraft: A tool for automated synthesis of fault-tolerant distributed programs. In *In International Conference on Concurrency Theory (CONCUR)*, pages 167–171, 2008.
9. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.

10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
11. L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *International Conference on Concurrency Theory (CONCUR)*, 2003.
12. D. de Niz and R. Rajkumar. Glue code generation: Closing the loophole in model-based development. *2nd RTAS Workshop on Model-Driven Embedded Systems*, 2004.
13. D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.
14. A. Ebnenasir and 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, pages 275–282, Washington, DC, USA, 2007. IEEE Computer Society.
15. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synchronize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
16. M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.
17. Z. Gu, S. Wang, and K. G. Shin. Synthesis of real-time implementation from uml-rt models. *2nd RTAS Workshop on Model-Driven Embedded Systems*, 2004.
18. G. Holzmann. The spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
19. P.-A. Hsiung and S.-W. Lin. Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems. *Comput. Lang. Syst. Struct.*, 34(4):153–169, 2008.
20. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238, 2005.
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*, 2008.
22. A. Knapp and 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*, 2002.
23. O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Logic in Computer Science*, 2001.
24. K. Larsen, P.Pattersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
25. J. Lilius and 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*, 1999.
26. S.-W. Lin, S.-W. Lin, C.-H. Tseng, T.-Y. Lee, and J.-M. Fu. Vertaf: An application framework for the design and verification of embedded real-time software. *IEEE Trans. Softw. Eng.*, 30(10):656–674, 2004. Member-Pao-Ann Hsiung and Member-Win-Bin See.
27. O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 274–289, 2006.
28. A. F. Martinez and 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*, pages 288–295, Washington, DC, USA, 2007. IEEE Computer Society.
29. J. Rumbaugh, I. Jacobson, and B. G. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.