UFIT: A Tool for Modeling Faults in UPPAAL Timed Automata⋆

Reza Hajisheykhi[1], Ali Ebnenasir[2], and Sandeep S. Kulkarni[1]

[1] Michigan State University, {hajishey, sandeep}@cse.msu.edu
[2] Michigan Technological University, aebnenas@mtu.edu

Abstract. We present the tool UFIT (*Uppaal Fault Injector for Timed automata*). In UFIT, we model five types of faults, namely, message loss, transient, byzantine, stuck-at, and fail-stop faults. Given the fault-free timed automata model and the selection of a type of fault, UFIT models the faults and generates the fault-affected timed automata model automatically. As a result, the designer can analyze the behavior of the model in the presence of faults. Moreover, there are several tools that extract timed automata models from higher-level programs. Hence, the designer can use UFIT to inject the faults into the extracted models.

1 Introduction

In this paper, we present the tool UFIT for modeling different types of faults in UPPAAL timed automata. Timed automata are important abstractions that are able to both capture real-time behavior and be verified algorithmically (model-checked). Moreover, there are several methods that propose how to extract a timed automata model from higher-level programs such as SystemC programs, hybrid systems, real-time communication protocols, digital circuits, timed asynchronous circuits, etc. [1, 2]. These programs/systems are usually subject to faults and it is necessary to see how they behave in the presence of faults.

There are several techniques for injecting faults into high-level programs such as C, C++, SystemC, etc. [3, 4]. However, such programs are getting more complex and more difficult to get verified. The faults injected also introduce some time overhead and make the verification time even worse. A solution to that would be extracting abstract models from the programs and inject the faults into the extracted models. Nevertheless, most of the algorithms for extracting the models are untimed and do not consider timing constraints. Having a timed model extracted (e.g. timed automata) from higher-level programs, there are several methods/tools for verifying the models in the literature [1, 5, 6]. However, these methods/tools verify the models in the absence of faults. Thus, there is a need for a tool that models and injects faults into timed systems automatically yet does not add too much overhead into the models.

⋆ This work is supported by NSF CCF-1116546, NSF CNS 1329807, and NSF CNS 1318678.

UFIT targets timed automata models and considers five types of faults, namely, message loss, transient, byzantine, stuck-at, and fail-stop faults. It automates the injection of the faults into the model and generates a fault-affected model. Hence, this model can be analyzed with UPPAAL tool-set. In this paper, we illustrate how to use UFIT to model the faults on the well-known Fischer's mutual exclusion problem and analyze the behavior of the fault-affected model. UFIT is written in Python and its source code is available freely and can be downloaded from `https://www.cse.msu.edu/~hajishey/ufit.html`.

2 Modeling and Analysis Using UFIT

In this section, first, we explain UPPAAL timed automata and the input of UFIT. Thereafter, using a runtime example, we introduce our fault modeling approach and inject five types of faults into the example. Finally, we utilize the output of UFIT to analyze the behavior of the model in the presence of faults.

2.1 Input of UFIT

The input of UFIT is a fault-intolerant timed automata model in XML format and a set of parameters. Next, we explain the timed automata and the input XML format. We describe the set of parameters in Section 2.2.

UPPAAL and timed automata. A timed automaton (TA) is a classical finite automaton which can manipulate clocks, evolving continuously and synchronously with the absolute time. Each transition (edge) of such an automaton is labeled by a guard, or a constraint over clock values, which indicates when the transition can be fired, and a set of clocks to be reset when the transition is fired. Each location (vertex) is constrained by an invariant. The invariant restricts the possible values of the clocks for being in the state, which can then enforce a transition to be taken. UPPAAL [7] is an integrated tool environment for modeling, simulation, and verification of real-time systems modeled as networks of timed automata, extended with data types.

XML format. Like the TA model, the XML file has a set of locations and transitions, which are respectively defined by the following tags: “< *location* > *statements* < */location* >” and “< *transition* > *statements* < */transition* >”. The *statements* can be a *name*, an *invariant*, or a *type* (e.g., urgent, committed) for locations, and a *source*, a *target*, or *labels* for transitions. The source and target tags represent the position of the transition. The label tag shows whether the transition has a *synchronization* channel, an *assignment* operation, or a *guard* condition.

We illustrate the set of parameters and our fault modeling approach used in UFIT utilizing a running example from the literature of UPPAAL timed automata, the *Fischer's mutual exclusion protocol* [7] (Figure 2(a)).

The running example. Fischer's protocol is designed to ensure mutual exclusion among several processes (5 processes here) competing for a critical section using timing constraints and a shared variable *id*. In each process P, the process

Fig. 1. The GUI of UFIT

goes to a request location req if it is the turn for no process to enter the critical section (id==0). After x time units in req ($0 \leq x \leq k$), P goes to the wait location and sets id to its process ID. Finally, after at least k time units, P enters the critical section cs if it is its turn. The Fischer's protocol satisfies the following set of requirements/properties in the absence of faults:

```
SPEC1: A[] not deadlock
SPEC2: P(i).req --> P(i).wait
SPEC3: A[] P1.cs + P2.cs + P3.cs + P4.cs + P5.cs <=1
```

where SPEC1 checks wether the system is deadlock-free. The liveness property SPEC2 checks that whenever a process tries to enter the critical section, it will always eventually enter the waiting location. The safety property SPEC3 checks for mutual exclusion of the location cs.

2.2 Internal Functionality

To generate the fault-affected model, in addition to the fault-free model, we need to specify the type of the faults and a set of parameters (see Figure 1). The fault types that UFIT considers are as follows.

- *Message faults,* where a message may be lost while forwarding from one module to another;
- *Fail-stop* faults, where a module fails functionally and the other modules cannot communicate with it;
- *Byzantine* faults, where the faulty component continues to run but produces incorrect results;
- *Stuck-at* faults, where a signal gets stuck-at a fixed value (logical 0, 1, or X) and cannot switch its value, and
- *Transient faults,* where the state of system components is perturbed without causing any permanent damage.

In addition to the fault type, the following discrete variables can be specified:

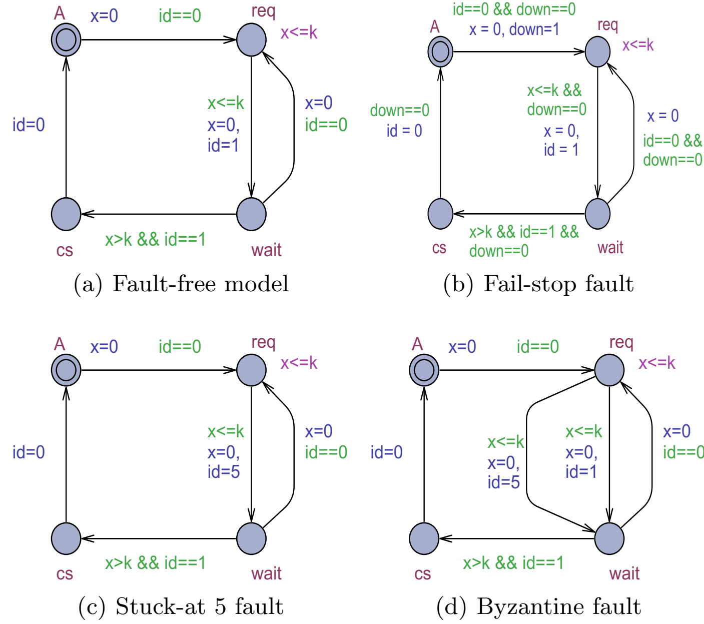
- *Variable subject to faults.* We are not allowed to increase or decrease the value of the clock variable;
- *Module subject to faults.* We assume any module can be subject to faults, and
- *Number of faults.* The number of occurrences of the transient faults that may take place during the computation needs to be defined. The default setting value is 1.

Remark 1. If any of the above variables is not specified, UFIT will set a value for them arbitrarily. For instance, if the module subject to fail-stop faults is not specified, UFIT will fail one of the modules non-deterministically.

Brief discussion about modeling of faults in UFIT. Given the parameters and the fault type, intuitively we model the faults as follows. To model a message fault, we inject a new transition into the module subject to faults in parallel to a transition that has a synchronization channel. The set of assignments/guards of the new transition is similar to that of the original transition except that the synchronization channel is changed. To model a fail-stop fault, we define a variable *down* that shows if a module is failed (down=1). For example, Figure 2(b) illustrates that automaton $P1$ is failed since $P1$ cannot go to location wait and has to stay at location req forever. To model stuck-at faults, UFIT finds the location of the variable subject to faults and changes it to a random value. For example, in Figure 2(c), the value of id is stuck at 5, thereby $P1$ cannot enter the critical section. For modeling byzantine faults, UFIT adds a transition in parallel to that of the original automaton that updates the variable subject to faults and changes its value arbitrarily. Figure 2(d) shows injecting a byzantine faults that changes the value of id, if the faults occur. Modeling of transient faults is similar to that of byzantine faults except that the occurrence of transient faults is limited. UFIT utilizes the number of faults defined in the GUI to limit the number of occurrence of this type of faults.
Extending UFIT. UFIT is easily extensible to cover more types of faults. Specifically, UFIT is written in Python (utilizing PyQt and PySide packages [8]) and uses *XML ElementTree* library to parse the XML file. Thus, to add a new class of faults, only the modeling of that class needs to be added to UFIT.

2.3 Analysis of Results

In this section, we analyze the fault-affected models. Also, in addition to Fischer's protocol, we include the results of the *Viking problem* adapted from [7]. In the Vikings problem, four Vikings want to cross a bridge at night, but they have only one torch and the bridge can only carry two of them. Thus, they can only cross the bridge in pairs and one has to bring the torch back to the other side before the next pair can cross. Each viking has different speed. The question is whether it is possible that all the vikings cross the bridge within a certain time.



(a) Fault-free model (b) Fail-stop fault

(c) Stuck-at 5 fault (d) Byzantine fault

Fig. 2. Fault-free and fault-affected models of Fischer's mutual exclusion protocol. The green texts show either the *guards* or *synchronization*, the blue texts show the *updates*, and the pink texts represent the *names*.

This example is comparable to the question if a packet can reach its receiver in a given time limit in a communication network/Network on Chip (NoC) system. The TA model satisfies the following properties in the absence of faults:

```
SPEC1: A[] not deadlock
SPEC2: E<> Viking1.safe
SPEC3: E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe
```

where SPEC2 illustrates that the first viking eventually gets to the other side of the river and SPEC3 shows that all the vikings are in their safe location.

The results of analyzing the examples are as shown in Table 1. In this table, if requirement x is satisfied, we include s in the table, otherwise v.

3 Conclusions and Future Work

In this paper, we presented the tool UFIT and explained how it models different types of faults in timed automata models. For each type of faults, we utilized a generic approach to transform the UPPAAL model to obtain a fault-affected model. Subsequently, this model was used in UPPAAL to conclude tolerance

Protocol	Cause	Affected Locations	SPEC			Total Time
			1	2	3	(ms)
Fischer's protocol	Fault-free model	–	s	s	s	1250
	Fail-stop	Process P1	v	v	s	143
	Transient	Process P1	v	s	s	79
	Stuck-at	Process P1	v	s	s	81
	Byzantine	Process P1	v	s	s	149
Viking protocol	Fault-free model	–	s	s	s	25
	Fail-stop	Viking 0	v	v	v	23
		Torch	v	v	v	15
	Message loss	Viking to Torch	v	v	v	17
	Byzantine (L=1)	Torch	v	s	v	29
	Stuck-at 0	Torch	v	s	s	15
	Stuck-at 1	Torch	v	s	v	15
	Transient (L=1)	Torch	v	s	v	14

Table 1. Modeling and analyzing the impact of faults.

to faults or to obtain a counterexample. We were either able to verify that the original specification is satisfied or find a counterexample demonstrating the violation of the original specification. Moreover, the time for evaluating the effect of faults was comparable ($< 165\%$) to the verification in the absence of faults.
Future work. Having a fault-affected timed automata mode and a set of properties which are violated, we are working on repairing the model automatically to generate a model that eventually satisfies the set of violated properties while preserving the set of satisfied properties. Moreover, we are working on injecting timing faults utilizing UFIT.

References

1. P. Herber, M. Pockrandt, S. Glesner, Transforming SystemC Transaction Level Models into UPPAAL timed automata, in: S. Singh, B. Jobstmann, M. Kishinevsky, J. Brandt (Eds.), MEMOCODE, IEEE, 2011, pp. 161–170.
2. A. Olivero, J. Sifakis, S. Yovine, Using abstractions for the verification of linear hybrid systems, in: Computer Aided Verification, CAV, 1994, pp. 81–94.
3. P. Lisherness, K. T. Cheng, SCEMIT: a systemc error and mutation injection tool, in: Design Automation Conference, DAC, 2010, pp. 228–233.
4. B. Giovanni, C. Bolchini, A. Miele, Multi-level fault modeling for transaction-level specifications, in: Great Lakes symposium on VLSI, 2009, pp. 87–92.
5. M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: Computer Aided Verification, CAV, 2011, pp. 585–591.
6. J. Springintveld, F. W. Vaandrager, P. R. D'Argenio, Testing timed automata, Theor. Comput. Sci. 254 (1-2) (2001) 225–257.
7. G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, in: SFM, 2004, pp. 200–236.
8. M. Summerfield, Rapid GUI Programming with Python and Qt, California: Prentice Hall, 2008.