



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# Design and Verification of a Fault-Tolerant System

---

Miaomiao Zhang, Zhiming Liu and Anders P. Ravn

November 2007

## UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on **formal methods** for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research **[R]**, Technical **[T]**, Compendia **[C]** or Administrative **[A]**. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

P.O. Box 3058  
Macao

---

# Design and Verification of a Fault-Tolerant System

---

Miaomiao Zhang, Zhiming Liu and Anders P. Ravn

## Abstract

We present a design of a triple modular fault-tolerant system that is a real case we received from our collaborators in the aerospace field. The system is used to compute the action that a subsystem should take and output the result to another subsystem. We model the system as timed automata, where a fault is modelled as an unobservable transition from a “good state” to an “error state”. Based on the faults that we were given by the application engineers, we design a system to tolerate the faults and use UPPAAL to check relevant properties.

**Keywords:** Fault-tolerance, real-time embedded systems, abstraction techniques, model checking.

**Miaomiao Zhang** is former fellow of UNU-IIST and now an associate professor at Tongji University, Shanghai, China. [miaomiao@mail.tongji.edu.cn](mailto:miaomiao@mail.tongji.edu.cn).

**Zhiming Liu** is a Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasising sound methods and tools for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for OO development. His teaching interests are Communication, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. His email address is [Z.Liu@iist.unu.edu](mailto:Z.Liu@iist.unu.edu)

**Anders P. Ravn** is professor at Aalborg University, Denmark. He has been a close collaborator of Zhiming Liu for many years, and visited UNU-IIST twice in the past two years. Email: [apr@cs.aau.dk](mailto:apr@cs.aau.dk).

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>System with a Non-Faulty CU and System with a Faulty CU</b>	<b>3</b>
2.1	Correct CU . . . . .	3
2.2	A Faulty CU . . . . .	4
<b>3</b>	<b>Design of the Triple Modular Fault-Tolerant Computer System</b>	<b>4</b>
3.1	Impulse generator . . . . .	5
3.2	CU and Watchdog . . . . .	6
3.3	Voters . . . . .	7
3.4	Arbiter . . . . .	8
3.5	Design of the Parametric Constraints . . . . .	8
<b>4</b>	<b>Model of the Fault-Tolerant System</b>	<b>9</b>
4.1	Fault Assumptions . . . . .	9
4.2	Impulse . . . . .	10
4.3	CU . . . . .	10
4.4	Voter . . . . .	12
4.5	Arbiter . . . . .	13
4.6	Parameters for the Complete Model . . . . .	14
<b>5</b>	<b>Verification</b>	<b>14</b>
<b>6</b>	<b>Conclusion and Future Work</b>	<b>15</b>



# 1 Introduction

Fault-tolerant computing is important and effective to improve the dependability of real-time systems [1, 5, 6]. This paper presents a design for a fault-tolerant system that is applied in the aerospace field; it will for instance receive readings from gyroscopes and star trackers, and then it routes signals to the thrusters to control a trajectory.

If we have a computation unit (CU) that does not suffer from hardware faults and correctly implements the computation, everything would be fine. However, in reality, computers or programs may fail, either because of hardware failures, or faults in the design or in the implementation of an algorithm. Obviously, a system with a single faulty CU may fail to give a correct output when a fault occurs during the execution. Consequently, if no fault-tolerant action is taken, such a faulty output may cause a system failure that violates the overall system requirements. In a system for an aerospace application, using fault-tolerant techniques to avoid failures for computers is important since maintenance is almost impossible once it is sent to the space.

In this paper, we adopt a classical fault-tolerant mechanism that uses multiple versions of the CU, which preferably are designed and implemented independently. The necessary redundancy is thus in space, preserving timing properties of the combined system. Redundancy in the time domain, for instance recovery block mechanisms, do not preserve timing properties as easily.

Although the principles of the mechanism are well known. It is far from easy to model the faults, the assumptions one makes for the concrete design. Abstraction techniques have to be applied so that the model of the design can be verified with the model checking tool.

The faults that we are asked to consider are transient, and therefore we can design a restart (or recovery) mechanism for a CU when it fails. This involves detecting the occurrence of a fault in a CU, and when it fails, there must be a component to trigger the restart. We are told by the domain engineer that they are concerned with three kinds of faults that may occur in a CU. The first kind of faults cause the CU to enter a deadlock state in which the CU does not take any action. For this, we equip each CU with a watchdog that is activated periodically (kicked) during normal operation. When a fault occurs, the CU deadlocks and stops kicking its watchdog. As a consequence, the watchdog timer grows until it exceeds a predefined value, (we say that the watchdog overflows), and that shall trigger a restart of the CU.

An occurrence of a fault of the second kind causes the CU to output incorrect data. In this case, the CU still keeps kicking its watchdog and thus the watchdog will not overflow. To detect an error caused by this kind of fault, we introduce a component called a *voter*. With the assumption that at anytime only a minority of the CUs can fail (in the concrete case with three CUs at most one of them can be in an error state), the voter can detect which CU has failed and thus can trigger a restart. Furthermore, the voter can also *mask* the incorrect output from the failed CU.

In the third case, a CU fails by entering a livelock state. In this state, the CU fails to output any result but it keeps kicking its watchdog periodically. Consequently, the watchdog timer does not

overflow and such a CU failure cannot be detected by the watchdog. However, the error detection in this case can be done by the voter by reading empty values of the failed CU. Thus we convert an omission fault into a value fault.

To avoid that the voter becomes a single point of failure, which would reduce the overall fault tolerance of the system, we can use more voters. We therefore need to design a component, called the *arbiter*, to detect an error in a voter and select the output from the voter that has not failed. With this design, the arbiter should also take the responsibility from the voter to trigger the restart of the failed CUs. To illustrate the idea of this design, we use two voters and show how the arbiter can trigger the restart of a failed CU, without considering the error detection in a voter.

Admittedly, in this paper, we only consider CU faults and do not design the concrete switching mechanism for voters when one of them crashes, so in our analysis we assume all the voters are not faulty. Also, the arbiter is a single point of failure, but it is an extremely simple piece of hardware, thus we accept this risk.

The main contribution is that we provide a model and a verification to show the system designed in the above way meets the requirements that domain engineers demand. We use a *fault-affected computational model* as in the papers [5, 6]. However, we use timed automata to model the design and CTL to specify the properties, instead of TLA that is used for both the specification of a design and the specification of properties in [5, 6].

When developing the model, we ask that it should be: (a) as close as possible to the informal system description and easily translated to system hardware implementation, (b) incorporate the fault hypotheses and, (c) allow effective application of the model checking tool for verification of the required properties.

We use UPPAAL [8] that is available at [www.uppaal.com](http://www.uppaal.com). It is an integrated tool environment for formal specification, validation and verification of real time systems modeled as networks of timed automata [7]. The language for the new version UPPAAL 4.0 includes a subset of the c programming language, and the syntax of timed automata for specifying timing constraints. The c code embedded in the automaton can be easily read and translated by the engineers to for instance the Verilog language for hardware implementation. Due to these extensions, the UPPAAL syntax appears to be sufficiently expressive for the description of critical parts of system specifications. The verification is convincing evidence that model checking is an effective technique for validating the behavior of a fault-tolerant embedded system [2, 3, 4].

The remaining sections are organized as follows: Section 2 describes the system with a single CU, that also defines the desired properties of the system. We then introduce the assumed faults into the automaton of the CU. Section 3 presents the design of the fault-tolerant system. In Section 4, we use a network of timed automata in UPPAAL to model the fault-affected behavior of the system. We verify the correctness properties in Section 5 in order to demonstrate that the system tolerates the assumed faults. Finally, Section 6 concludes the paper and points out future work.

## 2 System with a Non-Faulty CU and System with a Faulty CU

We use a fault-free CU to describe the correct behavior of the system. Then, we discuss the ways that the CU may fail and show how a fault-affected CU behaves. This motivates the design of the fault-tolerant systems in Section 3.

### 2.1 Correct CU

A CU receives the sampling data from a component as its inputs and computes an output to be used as input for another component. Let `cu_input` and `cu_output` denote the input and output of the U. In general, the CU computes a function of the input and possible some internal state, which we shall ignore without loss of generality, that is,  $\text{cu\_output} = f(\text{cu\_input})$ , for some function  $f$ . When a fault occurs, the CU may compute an incorrect value, that is  $\text{cu\_output} \neq f(\text{cu\_input})$ , or may not compute any output anymore. Within the system, an impulse generator is used to first clear the data and then issue an edge impulse *synclk\_xms* in every period of T time units to force the CU to read its inputs, compute and output the result to `cu_output`.

To simplify the model, we assume that the value of `cu_output` ranges from -1 to 1, and 1 is the correct value, -1 is the incorrect value, 0 is the cleared value. Initially, the value is 1.

The time spent on the computation is small and can be ignored. Fig.1 displays the CU system composed from one non-faulty CU and one impulse modelled in UPPAAL. The automaton Impulse clears the buffer `cu_output` and sends a *synclk\_xms!* signal each T time units that synchronizes with the automaton CU. In automaton Impulse, a clock *x* is used to record the time elapsed.

This system satisfies the following two properties: 1) every period of T time units, the CU computes a new value correct value, that is put in `cu_output`, 2). The value 1 is kept for T time units before it is cleared. This can be specified in UPPAAL as the following four properties  $P_1$ - $P_4$ :

- $P_1 : A[](\text{impulse.x} \geq 0 \text{ and } \text{impulse.x} \leq T)$
- $P_2 : A[](\text{impulse.x} > 0 \text{ imply } \text{cu\_output} == 1)$
- $P_3 : A[](\text{cu\_output} == 0 \rightarrow \text{cu\_output} == 1)$
- $P_4 : A[](\text{cu\_output} == 1 \rightarrow \text{cu\_output} == 0)$

In UPPAAL, the syntax  $P \rightarrow Q$  denotes a “leads-to” property meaning that whenever P holds Q will eventually hold. It is equivalent to  $A[] (P \text{ imply } A \langle \rangle Q)$ . The verification results show that all the above properties are valid for the model.



Figure 1: Non-faulty CU system: (a) CU automaton (b) Impulse automaton

## 2.2 A Faulty CU

In this case, the domain engineers are concerned with the following faults:

- **FAULT0:** The CU enters a deadlock state and it stops doing anything at all.
- **FAULT1:** The CU enters an error state in which it computes incorrect results.
- **FAULT2:** The CU enters a livelock state and executes only internal actions without outputting any result.

Following the technique in [5, 6], we model the behavior of the fault-affected CU with the automaton in Fig.2. In the fault-free location **Good**, the CU automaton nondeterministically selects which fault may occur. If **FAULT0** occurs, it moves to the error location **Error0** and the automaton deadlocks. Since the *synclk\_xms* is a broadcast channel, when the CU stays in location **Error0**, the Impulse automaton can still execute the *synclk\_xms!*. The location **Error1** is reached from location **Good** when **FAULT1** occurs. When the automaton stays in this state, it outputs incorrect data when the signal *synclk\_xms* is issued. Similarly, if **FAULT2** occurs, the CU goes to location **Error2**, in which it fails to output a data when the signal *synclk\_xms* is issued. There are also faulty transitions from location **Error1** to **Error0** and **Error2**. One may wonder why the CU needs a synchronization in **Error1** and **Error2** with the Impulse automaton. This is in fact to avoid *Zeno behavior* in the composed automaton. We can add self-transitions on these two states without synchronization with the Impulse, but additional clocks are then needed to rule out the *Zeno* behavior. This would complicate the model unnecessarily.

Using UPPAAL, the checks show that none of the properties  $P_1$ - $P_4$  for the fault-free system is satisfied by the fault-affected system.

## 3 Design of the Triple Modular Fault-Tolerant Computer System

To tolerate the assumed faults, we design a triple modular system shown in Fig.3. It consists of the components of three CUs each is equipped with a watchdog, two voters, one arbiter and one

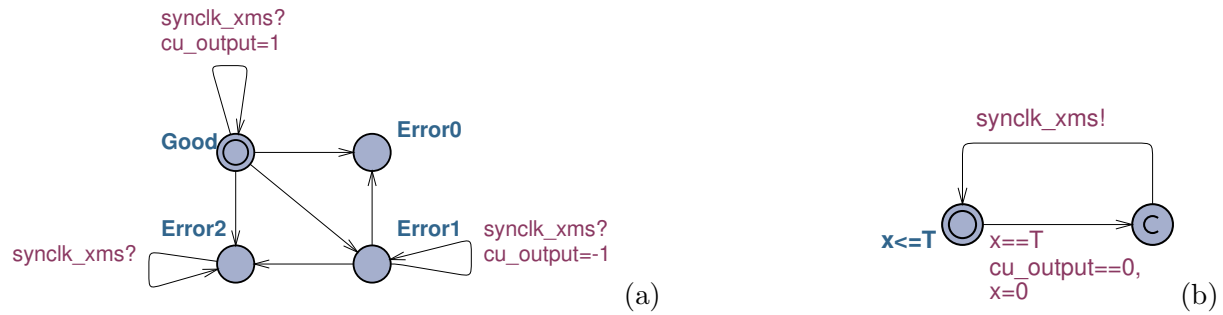


Figure 2: Faulty CU system: (a) CU automaton b) Impulse automaton

impulse generator. The watchdogs are modeled together with their CUs.

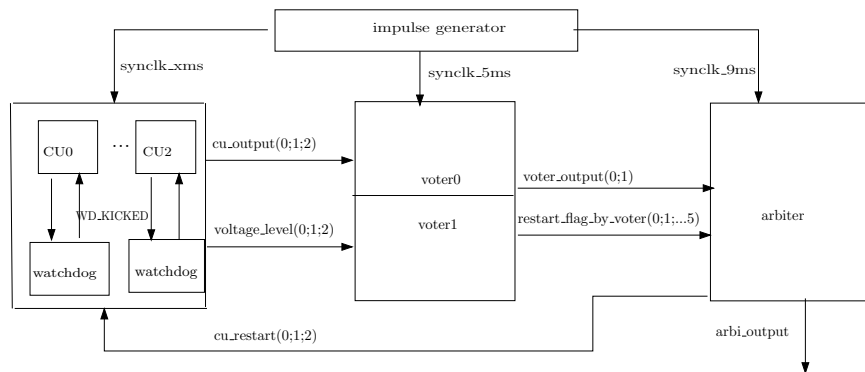


Figure 3: The triple modular redundancy system

### 3.1 Impulse generator

The impulse generator issues edge impulses to force the components to process their inputs. In a cycle, a synchronization impulse *synclk\_xms* is generated first to trigger the three CUs to process their inputs simultaneously. After a period *CU\_PERIOD* of time, a *synclk\_5xms* impulse is generated to trigger the two voters to process their inputs from the CUs simultaneously. Impulse *synclk\_9xms* is produced with a delay of *VOTER\_PERIOD* to activate the arbiter to process its inputs from the voters. A *synclk\_xms* impulse is produced again after a period of *ARBI\_PERIOD* time to trigger the CUs in the next cycle. So, all the three types of impulses are generated in every period of T, where T is equal to *CU\_PERIOD + VOTER\_PERIOD + ARBI\_PERIOD*.

### 3.2 CU and Watchdog

To make a CU recover from an error state, we introduce a restart mechanism for CU to determine when a CU needs a restart and which component is responsible for triggering a restart from an error state.

Introducing the restart mechanism, we illustrate the procedure of the restart for CU. When a CU restarts it enters a reset phase and stays in this phase for a period of `RESET_PERIOD` time before it enters a startup phase, in which it takes a period of `START_PERIOD` time. In the hardware design, the voltage change of a special pin of the CU signals this procedure. The voltage value stays low (0) for a period `RESET_PERIOD` before it is changed to high (1), see Fig. 4. We use a Boolean array `voltage_level` to denote the level of the pin voltage of the CUs. The fact that `voltage_level[i]` is 0 implies that CU *i* is in the reset phase, and it is in the startup phase or working status otherwise. The change of the value of `voltage_level[i]` from 1 to 0 indicates a restart of CU *i*.

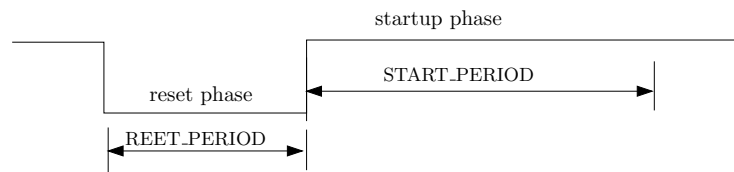


Figure 4: CU startup procedure

To allow a CU to recover from the error state `Error0`, we introduce a watchdog for each CU. The timer of the watchdog of the CU starts to count from the time when the CU enters a startup phase. In all the states, except for the state `Error0`, the CU kicks the watchdog in every period of `T` time units to set the timer value to `WD_KICKED`.

**Faulty behavior of a CU with a watchdog** The model of a fault-affected CU in Fig. 2 should now be augmented with the behavior of the watchdog:

- **FAULT0:** In this case, the CU stays in the error state `Error0` and stops. It also stops kicking the watchdog.
- **FAULT1:** In this case, the CU stays in the error state `Error1`. When `synclk_xms` is issued, the CU kicks its watchdog, but outputs an incorrect result.
- **FAULT2:** In this case, the CU stays in the error state `Error2`. When `synclk_xms` is issued, the CU kicks its watchdog, but fails to output a result.

Therefore, when the CU stays in location `Error0`, the watchdog timer grows until it overflows, that is, the timer value is equal to the constant `WD_PERIOD`. We decide that when timer of the

watchdog overflows, it triggers its CU to restart, and thus recover from an error state due to the occurrence of `FAULT0`. However if `FAULT1` or `FAULT2` occurs, the watchdog is still kicked normally. So the watchdogs cannot be used to detect occurrences of these faults.

### 3.3 Voters

To detect errors caused by `FAULT1` or `FAULT2`, we design a voter. The voter receives inputs from the three CUs. Assume that at any time at most one CU is in an error state, the voter votes for the value that is agreed by at least two CUs, and identifies whether the failed CU need a restart.

As we said before, a voter may fail. We use two voters, `VOTER0` and `VOTER1`. The purpose to have two of them is to avoid the voter as a single point of failure that reduces the overall dependability of the system.

When `VOTER0` and `VOTER1` receive a *synclk\_5xms* impulse, they simultaneously start to process the input data to determine which CU works correctly and select the correct result. The time spent on data processing is much smaller than the constant `VOTER_PERIOD`. This enables the voter to complete the computation before the occurrence of the *synclk\_9xms* impulse. We define voting functions as follows in which buffer `voter_output` is used to store the output of the voter.

$$\text{voter\_output} = \begin{cases} \text{cu\_output}[0] & \text{(if cu\_output}[0]=\text{cu\_output}[1] \text{ or } \text{cu\_output}[0]=\text{cu\_output}[2]) \\ \text{cu\_output}[1] & \text{(if cu\_output}[0] \neq \text{cu\_output}[1] \text{ and } \text{cu\_output}[0] \neq \text{cu\_output}[2]) \end{cases}$$

`VOTER0` is considered as the primary one in the sense that if both voters work well then the arbiter will select the result of `VOTER0` as its output. Only when `VOTER0` goes awry, is the result of `VOTER1` selected by the arbiter as the final output.

Furthermore, the voter has to detect whether a CU is in the reset phase or startup phase. For this, the value of the pin voltage `voltage_level[i]` is relayed to the voter. Therefore, in addition to the voting function, each voter is enhanced with the following functionalities.

**Working mode of a CU** Let `CU_normal_by_voter[3 × j + i]` be a Boolean variable, where  $i = 0, 1, 2, j = 0, 1$ . When `VOTER j` reads a correct output from CU  $i$ , it assigns 1 to the variable `cu_normal_by_voter[3 × j + i]` to indicate that CU  $i$  enters the *normal mode*. The CU remains in this mode until `VOTER j` finds a value change of `voltage_level[i]` from 1 to 0, and at this time `VOTER j` assigns 0 to `cu_normal_by_voter[3 × j + i]` and CU  $i$ 's mode becomes *abnormal*. The value of `cu_normal_by_voter[3 × j + i]` remains as 0 in the later cycles until `VOTER j` reads a correct value from CU  $i$ .

**Trigger a CU to restart** Let `restart_flag_by_voter[3 × j + i]` be a Boolean variable. It is used by VOTER *j* to trigger CU *i* to restart. There is no need for the voter to force a CU to restart every time when it reads an incorrect value from the CU. In particular, to make the watchdog, that takes some time for its timer to overflow, to be effective, we allow a CU to output a number of incorrect values before it is restarted. We introduce two parameters `n1` and `n2`, which are positive integers. When CU *i* is in the normal mode, VOTER *j* triggers it to restart by setting `restart_flag_by_voter[3 × j + i]` to 1, after having contiguously having received `n1` incorrect values from CU *i*. Similarly, when CU *i* is in the abnormal mode, voter *j* triggers the CU to restart after continuously having received `n2` incorrect values.

In summary, if FAULT1 or FAULT2 occurs, the voter can detect this fault by its incorrect inputs. The values of `restart_flag_by_voter` and `voter_output` will be read by the arbiter when it receives a `synclk_9xms` impulse, to trigger a CU to restart and to output the majority value.

### 3.4 Arbiter

Upon receiving a `synclk_9xms` impulse, the arbiter acquires the outputs of each voter. Let *j* be the index of the voter that the arbiter trusts, and `arbi_output` be the output of the arbiter. In each cycle, the arbiter assigns `voter_output[j]` to `arbi_output`, and sends a restart signal `cu_restart[i]` to CU *i* if `restart_flag_by_voter[3 × j + i]` equals 1. To insure that the arbiter completes the computation before the arrival of the next `synclk_xms` impulse, the computation time the arbiter spends must be less than ARBI\_PERIOD.

To summarize, the subsystem has three output variables: `cu_output` of a CU to the voters, `voter_output` of a voter to the arbiter and `arbi_output` of the arbiter to other components of the overall system. Only the last one is external to the subsystem.

### 3.5 Design of the Parametric Constraints

Taking the real hardware implementation into consideration, for example, a watchdog timer must not overflow when the CU is in the restart procedure, we have the following constraints.

$$(1) \quad \text{WD\_PERIOD} > 2T + \text{START\_PERIOD}$$

$$(2) \quad \text{WD\_PERIOD} > T + \text{WD\_KICKED}$$

A CU can restart by either the overflow of its watchdog timer or by a restart signal issued from the arbiter. Therefore, it could be that even when the CU is in the startup phase, it can enter the

reset phase again when it receives a restart signal from the arbiter. This delays the CU returning to normal working mode. To avoid repeated restart of the CU, we add the following constraint.

$$(3) \quad n2 > \lceil \frac{\text{RESET\_PERIOD} + \text{START\_PERIOD}}{T} \rceil + 2$$

Finally, according to the system mechanism, a voter can also detect incorrect output of a CU when FAULT0 occurs. If we set the value of the constant WD\_PERIOD too big, but the value of the constant n1 too small, the restart of the CU is triggered by the signal sent from the arbiter instead of the overflow of the watchdog. This would make the watchdog ineffective in detecting the occurrences of FAULT0. In reality, the engineers expect to distinguish the type of faults and correspondingly make change to improve the software. To do so, one of the guidelines can come from the behavior that shows which component will trigger the restart. Thus, the above scenario is not desirable, but can be avoided by the following constraint:

$$(4) \quad \lfloor \frac{\text{WD\_PERIOD}}{T} \rfloor > n1 + 1$$

## 4 Model of the Fault-Tolerant System

Before introducing the formal faulty model, we first give the fault assumptions. Based on that, we give a formal model with four automata: *Impulse*, *CU*, *Voter*, *Arbiter*.

### 4.1 Fault Assumptions

To achieve fault-tolerance of the triple modular system, we need the fault assumptions as well as the timing assumptions about the occurrence of faults [5, 6]. Thus at any time:

- at most one CU encounters a fault, and
- the minimum time between the occurrences of faults should be long enough to allow the successful recovery of a CU from an error state, and
- no faults occur in the voters and the arbiter.

These assumptions will be reflected in the model and some will be checked with the model checking tool.

### 4.2 Impulse

Figure 5 displays the automaton *Impulse*, which specifies how the impulse generator periodically produces edge impulse. The clock *x* is used to record the time passing between sending two edge impulses. Every *ARBI\_PERIOD* units of time, *clear\_fifo()* is executed to set *cu\_output* to 0. Immediately, i.e. in zero time, after this, *Impulse* broadcasts a CU synchronization signal *synclk\_xms!* to trigger the three CUs to process their inputs. A *synclk\_5xms!* is broadcast *CU\_PERIOD* time units after sending *synclk\_xms!*. A *synclk\_9xms!* is sent *VOTER\_PERIOD* time units after sending *synclk\_5xms!*.

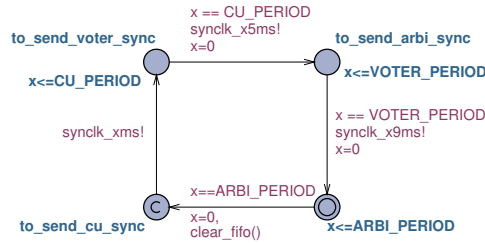


Figure 5: Impulse automaton

### 4.3 CU

Fig.6 displays the automaton *CU[i]*, where *i = 0, 1, 2* is the index of a CU. It models the following behaviors of *CU[i]*:

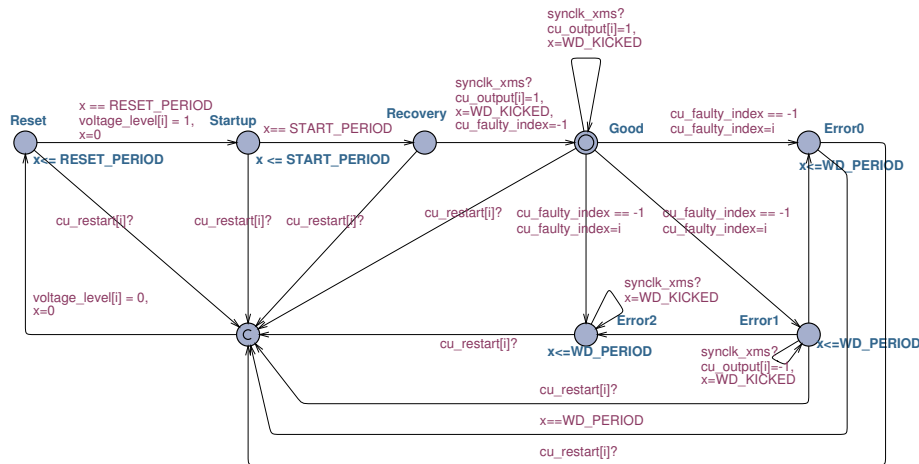


Figure 6: CU automaton

- The restart process.

- Decision on what to do after receiving a signal.
- Faults and faulty behavior.

In the automaton a clock  $x$  is used to measure the waiting time of the restart process and as the timer of watchdog. Initially  $\text{CU}[i]$  works well and stays in location **Good**. To denote the index of the faulty CU, an auxiliary global variable `cu_faulty_index` that takes values in the range of -1 to 2 is introduced, such that

- it is initially  $-1$ , representing that all the CUs are in location **Good**,
- when a fault occurs in  $\text{CU}[i]$ , the value of `cu_faulty_index` becomes  $i$ , and
- it remains  $i$  until the restarting process completes when the automaton enters location **Good** and `cu_faulty_index` becomes  $-1$ .

The self-loop in location **Good** models the scenario when a synchronization impulse *synclk\_xms* occurs, the CU outputs a correct result and kicks its watchdog.

When no fault has occurred in any CU, i.e. all the CUs are in location **Good**, a fault can non-deterministically occur in a CU and change the CU to an error location, **Error0**, **Error1** or **Error2**. We use `cu_faulty_index = i` to denote that a fault has occurred in  $\text{CU}[i]$ . The guard of the transitions ensure that the fault occurs if none of the three CUs has encountered a fault. After the CU enters in location **Error0** because of an occurrence of **FAULT0**, it stops working in this location and is triggered for a restart either by the overflow of the watchdog or a restart signal sent from the arbiter.

An occurrence of **FAULT1** moves the automaton to location **Error1**. In this location, the CU kicks its watchdog but outputs incorrect data when *synclk\_xms!* is issued. In this location when the automaton receives a `cu_restart[i]?` signal from the arbiter, the CU switches to the committed location immediately for a new restart.

An occurrence of **FAULT2** moves the automaton to location **Error2**. When *synclk\_xms!* is issued, CU kicks its watchdog, but fails to output a value. Similar to the case of **FAULT0** and **FAULT1**, in this location the automaton may receive a `cu_restart[i]?` signal from the arbiter for a restart.

Because of the parametric constraints in Subsection 3.5, and because of the signal *synclk\_xms!* which is generated in a  $T$  cycle, delay transitions of none of the locations of **Good**, **Error1** and **Error2** will lead to watchdog overflow. Therefore, there are not watchdog overflow transitions in these locations.

The CU reset and startup phases are described by the locations **Reset** and **Startup** and the transitions between them. As shown in Fig. 4, the automaton resides in location **Reset** for **RESET\_PERIOD** time units and then jumps to location **Startup** with `voltage_level` set to 1. It

stays in location `Startup` for a `START_PERIOD` period before moving to location `Recovery`. In location `Recovery`, when the synchronization `synclk_xms` occurs, `CU[i]` starts to output a correct data, meanwhile it kicks its watchdog and sets `cu_faulty_index` to `-1`. This means that `CU[i]` recovers from a fault and enters location `Good`.

Due to the fact that the watchdog timer starts to record the time elapsed since the `CU` entered a startup phase, and due to the constraints on the hardware parameters, watchdog timer overflows in location `Reset` or `Start` or `Recovery` do not occur.

**An over-approximation** By weakening guards, we might add behavior to an automaton. If an invariant is verified for the “weakened” automaton it will also hold for the original automaton. In the `CU` automaton, we have weakened the guards of the transitions caused by the occurrences of the faults. For these transitions, there are no channels or clocks used for synchronisation, but only the guard `cu_faulty_index == -1` is used to ensure the global assumption on faulty behavior that at anytime no more than one `CU` can fail. Therefore, the automaton `CU[i]` does not specify the exact time difference between two consecutive faults. Instead, it only models the fact that the minimum time when the next fault can occur should be long enough to let the `CU` output a correct result. This allows more timing behaviors regarding to occurrences of faults in the model.

#### 4.4 Voter

The automaton `Voter[j]` is shown in Fig. 7(a), where  $j = 0, 1$ , is the index of a voter. We introduce a local variable `CU_error_time[i]` to record the number of incorrect values that the voter read from `CU[i]`. Initially the automaton stays in location `Idle`. When it receives a `synclk_5ms` signal, it calls two functions: `fault_check()` and `vote()`. For each `CU`, function `fault_check()` defined in Fig. 8 computes the following steps.

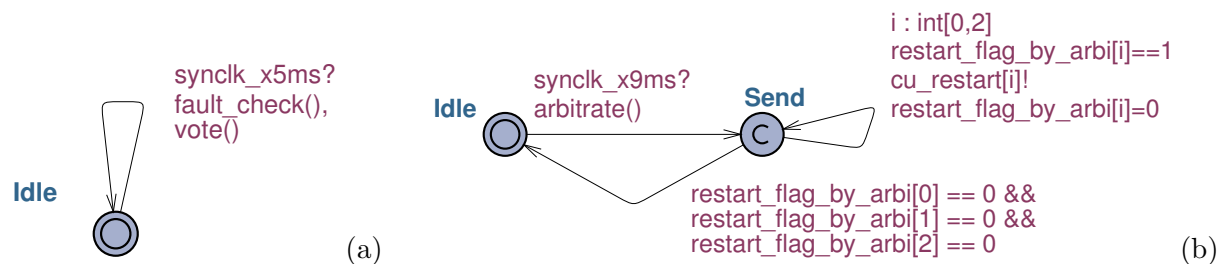


Figure 7: (a) Voter automaton (b) Arbiter automaton

- It checks if there is a restart of `CU[i]`. A local variable `lvoltage_level[i]` is used to store the value of `voltage_level[i]` from the last cycle.

- It checks the data read from `cu_output[i]` and decides if `CU[i]` needs a restart.

Function `vote()` completes the voting algorithm. It compares results from the three CUs, and outputs the majority value to the buffer `vote_output`, see Fig. 9.

```
void fault_check()
{ int i;
  for(i = 0; i < 3; i++)
    restart_flag_by_voter[i + id * 3] = 0;
  for(i = 0; i < 3; i++)
  {
    if (!voltage_level[i]==1 && voltage_level[i]==0) //edge jumps
    { cu_error_time[i] = 0;
      cu_normal_by_voter[i]=0; //judge CU i is in abnormal mode
    }
    voltage_level[i] = voltage_level[i]; //voltage_level is updated
    if(cu_output[i] !=1) //data from CU i is incorrect
    { cu_error_time[i]++;
      if(cu_normal_by_voter[i] == 1) //if CU i is in normal mode
      {if(cu_error_time[i] >= n1)
        {cu_error_time[i] = n1;
          restart_flag_by_voter[i + j * 3] = 1; //CU i needs a restart
        }
      }
      else // CU i is in abnormal mode
      { if(cu_error_time[i] >= n2)
        { cu_error_time[i] = n2;
          restart_flag_by_voter[i + j * 3] = 1; //CU i needs a restart
        }
      }
    }
  }

  else // data from CU i is correct
  { cu_error_time[i] = 0;
    cu_normal_by_voter[i]=1; //judge CU i is in normal mode
  }
}
}
```

Figure 8: Function `fault_check()`

```
void vote(){
  if (cu_output[0]==cu_output[1])
    voter_output[j]=cu_output[0];
  else if (cu_output[0]==cu_output[2])
    voter_output[j]=cu_output[0];
  else voter_output[j]=cu_output[1];
}
```

Figure 9: Function `vote()`

## 4.5 Arbiter

We use a local Boolean variable `restart_flag_by_arbi[i]` to express if `CU[i]` needs to restart. The automaton resides in `Idle` initially, whenever `synclk_x9mx` is issued it executes the function `arbitrate()` and moves to location `Send`, see Fig. 7(b).

```

void arbitrate() {
  int i;
  for (i=0;i<=2;i++)
    restart_flag_by_arbi[i]=restart_flag_by_voter[i];
  arbi_output=voter_output[0];
}

```

Figure 10: Function `arbitrate()`

The transition from location `Send` models that if `Arbiter` decides CU `i` should restart, it sends a restart signal to notify `CU[i]` to enter a reset procedure via action `CU_restart[i]!` that synchronizes with the `CU[i]` automaton.

## 4.6 Parameters for the Complete Model

Considering the concrete hardware implementation, we decide on the following values for the different timing constants; these values satisfy the parametric constraints in Subsection 3.5. These definitions are copied almost verbatim in the UPPAAL declaration section of our model.

```

CU_PERIOD:    0.5 ms (time difference between synclk_xms and synclk_5xms)
VOTER_PERIOD: 0.4 ms (time difference between synclk_5xms and synclk_9xms)
ARBI_PERIOD:  0.1 ms (time difference between synclk_9xms and synclk_xms)
RESET_PERIOD: 100 ms (the period that CU stays in reset phase)
START_PERIOD: 300 ms (the period that CU stays in startup phase)
WD_KICKED:    450 ms (the value watchdog timer kicked)
WD_PERIOD     500 ms (the maximum time watchdog timer can record)
n1: 30 (the number of incorrect data voter allows if voter judges CU normal)
n2: 550 (the number of incorrect data voter allows if voter judges CU abnormal)

```

## 5 Verification

First of all, we check the fault-tolerant related properties. The properties  $Q_{10}$ - $Q_{42}$  of the fault-tolerant system ensures the correctness properties  $P_1$ - $P_4$  of the fault-free system in Subsection 2.1. Here  $Q_{20}$  means that at any time, at most one CU encounters a fault. This correctness is based on the notion of *refinement mapping* in [10, 5, 6].

- $Q_{10} : A[](\text{impulse.x} \geq 0 \text{ and } \text{impulse.x} \leq T)$
- $Q_{20} : A[]((\text{cpus}(0).\text{Good} \text{ and } \text{cpus}(1).\text{Good} \text{ and } \text{cpus}(2).\text{Good})$   
 $\text{or } (\text{cpus}(0).\text{Good} \text{ and } \text{cpus}(1).\text{Good})$   
 $\text{or } (\text{cpus}(0).\text{Good} \text{ and } \text{cpus}(2).\text{Good})$   
 $\text{or } (\text{cpus}(1).\text{Good} \text{ and } \text{cpus}(2).\text{Good}))$
- $Q_{21} : A[](\text{voter\_output}[0] == 1 \text{ and } \text{arbi\_output} == 1)$

- $Q_{30} : A[](\text{cu\_output}[0] == 0 \rightarrow \text{cu\_output}[0] == 1)$
- $Q_{31} : A[](\text{cu\_output}[1] == 0 \rightarrow \text{cu\_output}[1] == 1)$
- $Q_{32} : A[](\text{cu\_output}[2] == 0 \rightarrow \text{cu\_output}[2] == 1)$
- $Q_{40} : A[](\text{cu\_output}[0] == 1 \rightarrow \text{cu\_output}[0] == 0)$
- $Q_{41} : A[](\text{cu\_output}[1] == 1 \rightarrow \text{cu\_output}[1] == 0)$
- $Q_{42} : A[](\text{cu\_output}[2] == 1 \rightarrow \text{cu\_output}[2] == 0)$

The next correctness property is that the CU should not restart when it is in the startup phase. We only need to check if `Arbiter` can send a restart signal whenever the CU resides in location `Startup`.

- $A[] \text{ not forall } (i : CType) (\text{CUs}(i).\text{Startup} \text{ and } \text{arbiter}.\text{Restart\_flag\_by\_arbi}[0] == 1).$

Finally, the system has no deadlock.

- $A[] \text{ not deadlock.}$

The checking of each of these properties only takes a few seconds. The verification results in UPPAAL reveal that all the above properties are satisfied by the model of the design.

## 6 Conclusion and Future Work

We have presented our design of a triple modular fault-tolerant system, and applied formal techniques in modeling of a fairly complex concurrent system, and shown verification with the model checking tool UPPAAL. We were given this problem by domain engineers in the aerospace field. Our experience shows that formal modeling and verification are applicable to verification problems in practical fault-tolerant systems. Such designs are in general hard to test for software and system engineers, and their solution require delicate techniques in modelling from experts in the area of formal verification.

The results also show that UPPAAL is able to model the system faithfully. Especially, the `c` programming language in the tool, which in our model includes the functions `fault_check()`, `clear_fifo()`, `vote()` and `arbitrate()`, can be easily translated to Verilog hardware language during the implementation.

In our paper, we ignored the computation time for each component. One reason is that the computation time is relatively small compared with the time different between two consecutive

impulses, so this does not affect the correctness properties. The other reason is that when we use a model with multiple clocks to describe the computation time for each component, we encounter the state space explosion problem. To further reduce state space, as demonstrated by [9], which has been successfully applied in an Zero Configuration protocol, we may use dead variable reduction here to abstract our model. Dead variable reduction is a well known static analysis technique, that has for instance been studied in the PhD thesis of Yorav [11]. A variable  $v$  is said to be dead at a location  $l$  if on every execution path from  $l$ ,  $v$  is defined before it is used, or is never used at all. Clearly, systems that only differ in the values of dead variables are equivalent in a very strong sense (bisimilar). In our model, array `cu_output` is dead in location `Idle` in automaton `Voter`, and can be reset to zero after completing functions `fault_check()` and `vote()` upon entering this location. Another example is that the arrays `voter_output` and `restart_flag_by_voter` are dead in location `Idle` in automaton `Arbiter`, and can be reset to a default value upon occurrence of the transition to this location.

For future work, there are several directions that we pursue. We have not considered the voter switching strategy, neither of a faulty voter. In the current system the arbiter trusts the output of `VOTERO` by default. So clearly, defining the switching mechanism is a direction in which the effort on modeling and analyzing the fault-tolerant properties can also be extended. Up to now, we have verified the functionality requirements of the system. In our future work, we will consider how the dependability (performance) of the system may be checked using a model checker for real-time probabilistic systems.

## References

- [1] B.W. Johnson. *Design and analysis of fault-tolerant digital systems*. Addison-Wesley Publishing, 1989.
- [2] Francis Schneider, S.M. Easterbrook, J.R. Callahan and G.J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *Proceedings of the 3rd International Conference on Requirements Engineering*, pages:4-13, IEEE Computer Society, 1998.
- [3] S. Gnesi, G. Lenzini and F. Martinelli. Logical specification and analysis of fault tolerant systems through partial model checking. In: *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, pages. 57-70. Electronic Notes in Theoretical Computer Science 118. 2003.
- [4] C. Bernardeschi, A. Fantechi and S. Gnesi. Model checking fault tolerant systems. *Journal of Software Testing, Verification and Reliability (STVR)*, 12(4):251-275, Publisher: John Wiley & Sons, 2002.
- [5] Z. Liu and M. Joseph. Specification and verification of fault-tolerance timing, and scheduling. *ACM Transactions on Programming Languages and Systems*. 21(1): 46-89 (1999).
- [6] Z. Liu, M. Joseph. Verification of fault-tolerance and real time. *FTCS 1996*: 220-229, IEEE Computer Society, 1996.

- 
- [7] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, pages 183–235, 1994.
- [8] G. Behrmann, A. David and K.G. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, LNCS 3185, pages 200-236. Springer Verlag, 2004.
- [9] B. Gebremichael, F.W. Vaandrager, M. Zhang. Analysis of the Zeroconf Protocol Using UPPAAL. In *Proceedings of the 6th Annual ACM & IEEE Conference on Embedded Software (EMSOFT 2006)*, ACM Press, pages 242-251. 2006.
- [10] M. Abadi and L. Lamport. The existence of refinement mapping. *Theoretical Computer Science* 83, 2, 253–284.
- [11] K. Yorav. *Exploiting syntactic structure for automatic verification*. PhD thesis, The Technion, Israel Insitute of Technology, 2000.