Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 121–136
www.elsevier.com/locate/entcs

Parametric Analysis of an Improved Fault Tolerant System

Miaomiao Zhang and Wenzhong Qin[1,2]

School of Software Engineering
Tongji University
Shanghai, China

Abstract

We report our preliminary study on an improved triple modular computer system from the aerospace field, which is designed to compute the course of action that other subsystems take and output the result to other subsystems. Based on the formal model of the system, we derive constraints on the values of the parameters that occur in our model, and manually prove that " When a CPU is in the startup phase, it should not restart again due to its watchdog timer overflow or a restart signal sent from the arbitrator ".

Keywords: fault-tolerance, real-time embedded systems, parametric constraints.

1 Introduction

In safety-critical applications, ranging from national defense to interests of commercial companies and private citizens, techniques of fault tolerant computing are important and effective to improve the reliability and dependability of real-time systems [1,7,8].

In this paper, we present our study on a triple modular computer system from the aerospace field, which is designed to compute the course of action that other subsystems take and output the result to other subsystems. This work is based on our early experiment and improvement on a design originally given to us by the practical engineer. In the experiment, we analyzed the drawbacks of the design, and suggested a design that provides more effective fault-tolerance. In the improvement [6], we endow each component of the system with more functions. Then we give a formal model of the improved system using a network of timed automaton [9] in UPPAAL [10]. The detailed C code specification can be easily read and translated

[1] Supported by NSFC (No.60603037 and No.60673114), 863 Program (No.2006AA01Z165).
[2] Email: miaomiao,wenzhongq@mail.tongji.edu.cn

doi:10.1016/j.entcs.2008.03.089

to Verilog hardware language by the engineers. We further prove that with the timing parameters the hardware engineers provide, the improved system satisfies a list of important functionality requirements. Among these, the most important one is $\mathcal{T}$, which is " When a `CPU` is in the startup phase, it can not restart again due to its watchdog timer overflow or a restart signal sent from the arbitrator ".

However, UPPAAL does not support parameters so we need to instantiate the parameters for some values in different hardware environments to check whether or not $\mathcal{T}$ is fulfilled. In our current work, we improve on the above results in two ways: (1) we find constraints on the parameters that ensure satisfaction of property $\mathcal{T}$; (2) we prove manually with the parametric constraints, the system satisfies the correctness property. Therefore, rather than verifying the correctness of the system for a single, specific choice of parameter values like in UPPAAL, we derive (by hand) general constraints on the parameters which ensure correctness and give the correctness proof. Such technique has been applied in other case study, such as lego car system [12].

As shown by our earlier work, the system falls into the case of the big state space problem [13,14].To make the testing and verification with fixed parameter values practically feasible, several abstraction techniques have been applied in the modeling previously. Still the same for the manual proof in our current paper, there is no interconnection between any two of the `CPUs`, and each voter can independently judge whether its corresponding `CPU` needs a restart. Thus, the properties $\mathcal{T}$ of the system can be described and manually proved by working on a simplified version of the system with only one `CPU`, one voter and one arbitrator.

The remainder of the paper is organized as follows: Section 2 presents the original triple modular redundancy system and an improved version of the system, focusing on the augmented functions of the voter and the arbitrator. With a network of timed automata, Section 3 gives a formal model of the improved system in UPPAAL. Section 4 presents the correctness properties that we want to prove, the constraints needed for their validity. Section 5 roughly gives the correctness proofs. Finally, Section 6 concludes the paper.

2 Improved Triple Modular Fault Tolerant Computer System

2.1 The Original One

The original system mainly consists of the components: three `CPUs`, two voters, one arbitrator and one impulse generator.

2.1.1 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 let the three `CPUs` process their inputs simultaneously. After a certain period `CPU_PERIOD` of time, a *synclk_5xms* impulse is generated to trigger the two voters

to process their inputs simultaneously. Impulse *synclk_9xms* is produced a period VOTER_PERIOD of time afterwards since the sending of *synclk_5xms* to activate the arbitrator to process its inputs. A *synclk_xms* impulse is produced again after a period of ARBI_PERIOD time since the sending of *synclk_9xms* impulse. So, all the three types of impulses are generated in a T-cycle, where T is equal to CPU_PERIOD + VOTER_PERIOD + ARBI_PERIOD.

2.1.2 CPU

As a CPU starts, it first enters a reset phase and stays there for a period C before it enters a startup phase, in which it may stay for maximally a period of RUB time. In other words, from the time when CPU jumps to startup phase till the time when it is in working status, the period lies in the interval [0, RUB]. In the hardware design, the voltage change of a special pin from the CPU illustrates this procedure: the voltage value stays in the low level (0) for a period C before it is changed to the high level (1). We use a Boolean array flag to denote the value of pin voltage of the CPUs. Thus, the fact that flag[i] is 0 implies that CPU i is in the reset phase, and it is in the startup phase or working status otherwise.

A CPU receives various sampling data as its inputs and computes results as outputs. Let cpu_voter_fifo[i] denote the buffer that stores the output of CPU i, that can be read by its voters. After receiving a *synclk_xms* impulse at some time t, all the three CPUs start to execute the following actions simultaneously.

(i) Clear the data in buffer cpu_voter_fifo which are the results from the last cycle.

(ii) Process and compute the inputs.

(iii) Output the processing result to cpu_voter_fifo.

Among these actions, the second action to process and compute the inputs takes more time than the other two actions do. When a computation times out, that is, when processing time is equal to or bigger than CPU_PERIOD, the program installed on the CPU may enter into a deadlock or an infinite loop. So the inputs a voter later reads are incorrect.

2.1.3 Watchdog

Each CPU is connected with an exterior watchdog. The watchdog timer starts to record the time elapsed since the connected CPU enters into a startup phase. In the normal mode, the CPU kicks the watchdog periodically with timer being set to the value RWD. If the CPU is hanging or is unable to work correctly, it then fails to periodically reset the watchdog timer. As a consequence, the timer expires a constant WD_UB. In this case, the CPU is considered to have a permanent fault and is switched off. A signal demotion from watchdog is issued to trigger that.

2.1.4 Voters

The system includes two voters, voter0 and voter1, which respectively judge the result of which CPU is perfect. Triggered by a impulse *synclk_5xms*, they start to

process their inputs simultaneously. The purpose to have two of them is to avoid the voter as a single point of failure that reduces the overall fault tolerance behavior of the system. We use a boolean variable `voter_status` to describe if a voter is faulty or not, where 1 indicates the voter is faulty. In our later model, to reduce system behaviors, we assume all the voters are not faulty. This assumption will not alter the checking results of the interested properties.

Whenever voter0 and voter1 receive a *synclk_5xms* impulse, they simultaneously start to process the input data to determine which `CPU` works correctly and select the correct result. The time spent on data processing for each voter may be different but it is bounded by a constant `VOTER_PERIOD`.

2.1.5 Arbitrator

Upon detecting a *synclk_9xms* impulse, the arbitrator starts to work. Depending on the statuses of the voters, i.e, if voter is faulty or not, and whether a voter is primary or not, the arbitrator decides to output either the result of voter0 or voter1. We use `arbitrator_data` to denote the arbitrator output. To insure the arbitrator completes the computation before the arrival of *synclk_xms* impulse that triggers the next `CPU` cycle, the computation time the arbitrator requires must be less than `ARBI_PERIOD`.

2.2 System Improvement

The purpose of this system is to successfully use the three-modular mode mechanism as often as possible. However, the system designed up to now has several disadvantages (1) as soon as watchdog timer overflows, the connected `CPU` is switched off even if the `CPU` fault is resumable, (2) even though an infinite computation loop will generate incorrect results to a voter all the time, the watchdog is kicked normally during that time.

In both cases,triple modular redundancy is changed to two modular redundancy, which weakens fault-tolerance effect. A `CPU` restart mechanism can solve the above problems. Therefore, to solve both problems, we define a mechanism that addresses the following aspects.

(i) When does a `CPU` need a restart? When and which component will trigger this restart?

(ii) When is `CPU` considered to have a permanent fault and to be switched off.

To answer these question, we need to add new functionalities to each component. These require us to introduce new information and coordinate these information flowed in between these components.

2.2.1 CPU Restart and Permanent Fault

Compared with the previous system, each `CPU` is now augmented with a restart function. A watchdog timer overflow will cause the restart of the `CPU` instead of considering it as a permanent fault to be switched off. Besides, not only can the

watchdog trigger a restart of the CPU, but the arbitrator will also send a restart signal to the CPU if needed. The details on how a voter together with the arbitrator judges whether the CPU needs to restart will be described in the following subsection.

In the improved system, we also require that a CPU should be switched off whenever it has a permanent fault. However the judgement when a CPU is considered to have a permanent fault needs to be refined. As said previously, a CPU is regarded to be permanently faulty when the connected watchdog timer expires. While for the new system the modification is: when a CPU continuously has restarted six times, each voter together with the arbitrator decide that the CPU has a permanent fault. For a voter to detect if a CPU, for instance CPU i, has restarted, the information regarding CPU i is either in the reset phase or the startup phase, that is flag[i], is relayed to voter. The value change of flag[i] from 1 to 0 indicates a restart of CPU i.

2.2.2 Improvement of Voter

In addition to the old functionalities, each voter is enhanced with the following functionalities in the new system.

Decide if CPU i works normally

If the data that voter j reads from the output buffer of CPU i, that is cpu_voter_fifo[i] is correct, CPU i is considered to be in normal status. We denote this by setting a local boolean variable cpu_normal_by_voter[3 × j + i] to 1. If voter j finds a value change of flag[i] from 1 to 0, it then judges that CPU i is in non-normal status by assigning 0 to cpu_normal_by_voter[3 × j + i]. The value of cpu_normal_by_voter[3 × j + i] being 0 is not altered in the later cycles until voter j reads correct data from CPU i. So the normal status and the non-normal status of CPU are decided by voter.

Judge if CPU i needs a restart or not

Let restart_flag_by_voter[3 × j + i] be boolean variable. It is used to indicate voter j has judged whether or not CPU i needs a restart. To make the watchdog, that takes some time for its timer to overflow, to be effective, we allow a CPU to output a number of incorrect values before it it is restarted. We introduce two parameters n1 and n2, which are positive integers. When CPU 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 CPU i. Similarly, when CPU i is in the abnormal mode, voter j triggers the CPU to restart after continuously having received n2 incorrect data.

Count the number of restart times of CPU i

To judge if CPU i has a permanent fault and thus needs to be switched off, voter j needs to count how many times CPU i has continuously restarted. A variable restart_count_by_voter[3 × j + i] is used to record the number of times of restart of CPU i judged by voter j. If voter j finds that flag[i] varies from 1 to 0,

`restart_count_by_voter`[3 × `j` + `i`] is increased by 1. It is set to 0 if voter `j` reads a correct data from `cpu_voter_fifo`[`i`]. The values of `restart_flag_by_voter` and `restart_count_by_voter` are later read by the arbitrator.

2.2.3 Improvement of Arbitrator

Upon receiving a *synclk_9xms* impulse, the arbitrator acquires the outputs of each voter, which include four kinds of information: the voting result of each voter, the voter's status, whether each `CPU` needs a restart or not, number of times of restarts of each `CPU`. Depending on the voter's status, for instance, if voter0 is good, the arbitrator decides to use the result of vote0 and sends a restart signal `dsp_restart`[`i`] to `CPU i` if `restart_flag_by_voter`[`i`] is 1.

Moreover, arbitrator is equipped with a function to send a modular degradation signal by setting the boolean variable `demotion`[`i`] to 1 in case when it decides there is a permanent fault in `CPU i`. Based on the status of the voters, the arbitrator determines if it should believe in the outputs of voter0 or that of voter1. Secondly if the arbitrator deems voter `j` is good, then it checks if `restart_flag_by_voter`[3 × `j` + `i`] is equal to 6. If it is, the arbitrator determines that `CPU i` has a permanent fault and triggers it to be switched off. The current system and information flow between components is shown in Fig. 1. The thick arrows in the figure show the new information exchange, compared with the previous system.

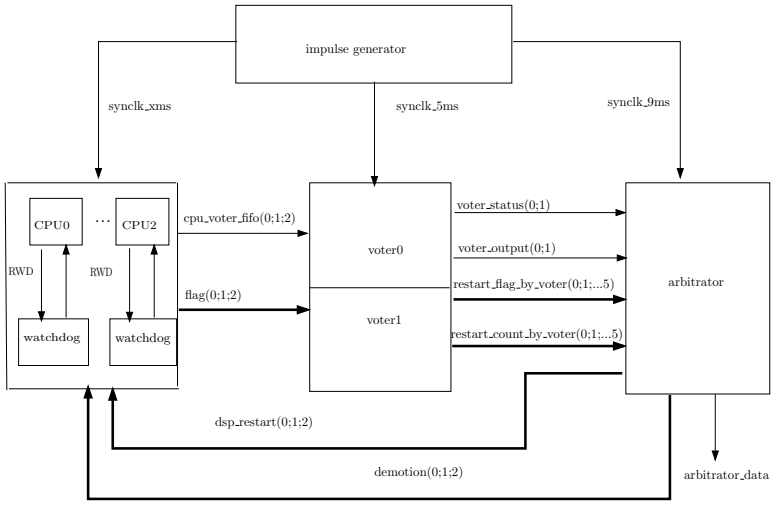

Fig. 1. Triple modular redundancy system after improvement

3 System Model

Based on the modified system, we give a formal model of the system with five automata: `Impulse`, `CPU`, `Voter`, `Arbitrator` and `Demoter`. We refer to [6] for more detailed explanation of the model.

3.1 Impulse

Figure 2(a) displays the automaton `Impulse`, which specifies how the impulse generator produces edge impulse in turn. Clock `x` is used to record the time passing in between sending two edge impulses.

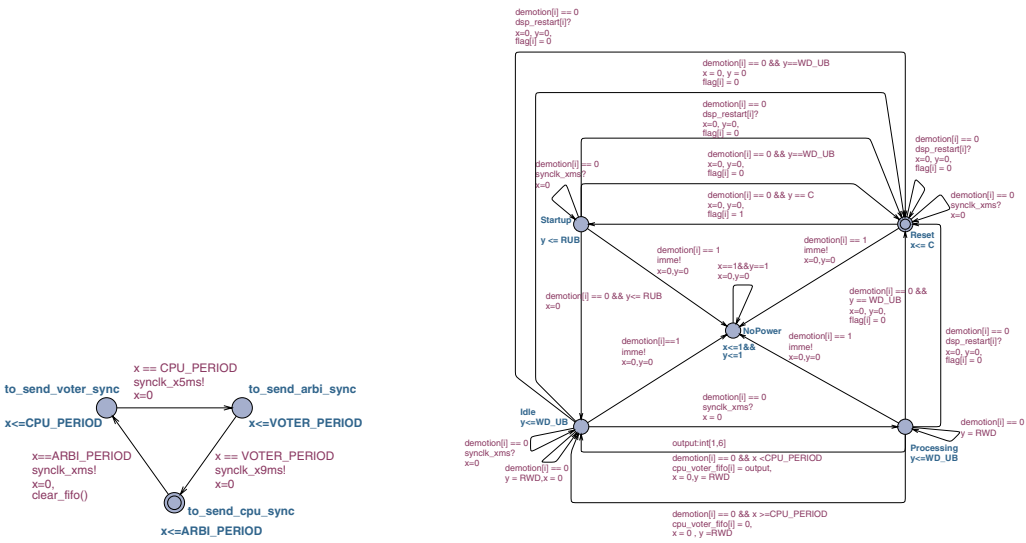

Fig. 2. a) Impulse automaton b) `CPU` automaton

3.2 CPU

Fig.2(b) displays the automaton `CPU[i]`, where `i` is the index of a `CPU`.

Fig. 3. `Demoter` automaton

There are two clocks `x`,`y` in the automaton, where `x` is used to measure the data processing time, `y` is used to measure the waiting time of the watchdog timer. The `CPU` reset and startup phases are embodied by means of the locations `Reset` and `Startup` and the transitions in between them.

After the startup phase, `CPU[i]` stays in `Idle` to wait for a `CPU` synchronization impulse. In reality, the `CPU` has its own real state: the non-faulty state and the faulty state. In the case of the non-faulty state, the `CPU` kicks its watchdog with the timer set to `RWD`, where we use a transition surrounding `Idle` to indicate that. For the faulty case, it ceases doing this. As a result, the watchdog timer settles in `Idle` and will reach the limit `WD_UB` with the time being elapsed, which forces the automaton to restart. Each outgoing transition from location `Idle` to `Reset` denotes a restart phenomenon. Once a synchronization impulse *synclk_xms* occurs, that is, when a *synclk_xmx*? transition occurs, The `CPU` may either transit to location `Processing`

with clock x reset to 0 if CPU is non-faulty, or to location Idle again if the CPU is faulty.

In location Processing, the CPU processes the sampling data and outputs the result to buffer cpu_voter_fifo[i]. The waiting period is modeled by resetting the clock x upon entering location Processing and by bounding the time of the CPU may staying in this location with the invariant $y \leq$ WD_UB. In this location, the CPU may kick the watchdog, or stop doing so which leads to the timer overflow and a consequent move to location Reset. Two outgoing transitions from Processing to Idle model the computation timeout and non-timeout cases.

Whenever the watchdog timer overflows or receives a dsp_restart[i]? signal from the arbitrator, CPU[i] will switch to Reset without time delay, and set flag[i] to 0. This occurs in any of the active locations: Reset, Startup , Idle and Processing. The fact that dsp_restart[0]? being an urgent channel ensures the transition is taken immediately as it is enabled. The location Nopower in the automaton designates that CPU[i] has a permanent fault and is turned off. From any of the locations, except for NoPower, CPU[i] transits to NoPower immediately after it detects that the arbitrator has decided this CPU has a permanent fault, i.e. demotion[i] == 1.

3.3 Voter

The automaton Voter[j] is shown in Fig. 4(a), where j is the index of a voter. A clock x is used to record the voter processing time. We introduce a local variable cpu_error_time[i] to record the number of incorrect data that the voter obtained from CPU[i]. Initially the automaton stays in its Idle location. As soon as it receives a *synclk_5ms* signal, it immediately jumps to location Processing with x reset to zero. The processing time is non deterministic, but bounded in the interval [0, VOTER_PERIOD). As to the transition from Processing to Idle, Voter[j] calls two functions: fault_check() and vote(). The function fault_check() is defined as follows.

```
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(demotion[i] == 0)//CPU i is not in NoPower location
 { if (local_flag[i]==1 && flag[i]==0) //edge jumps
     {cpu_error_time[i] = 0;  //set cpu error time to 0
      cpu_normal_by_voter[i]=0; //judge CPU i is in non-normal state
      if(restart_count_by_voter[i + id * 3] >= 6) //CPU i restart times exceeds 6
              restart_count_by_voter[i + id * 3] = 6;
      else // CPU i restart times does not exceed 6
              restart_count_by_voter[i + id * 3]++;
     }

     local_flag[i] = flag[i]; //local_flag is updated
    if(cpu_voter_fifo[i] == 0) //data from CPU i is not correct
     {cpu_error_time[i]++;
      if(cpu_normal_by_voter[i] == 1) //CPU i is in normal state
        {if(cpu_error_time[i] >= n1)
           {cpu_error_time[i] = n1;
            restart_flag_by_voter[i + id * 3] = 1; //CPU i needs a restart
           }
        }
      else // CPU i is non-normal state
        {if(cpu_error_time[i] >= n2)
           {cpu_error_time[i] = n2;
            restart_flag_by_voter[i + id * 3] = 1; //CPU i needs a restart
           }
```

```
      }
     }
    else  // data from CPU i is correct
      { cpu_error_time[i] = 0;
        restart_count_by_voter[i + id * 3]=0;
        cpu_normal_by_voter[i]=1; //CPU i is in normal state
      }
   }
 }
}
```

a) b)

Fig. 4. a)Voter automaton b) Arbitrator automaton

3.4 Arbitrator

The automaton `Arbitrator` resides in `Idle` initially, whenever *synclk_x9mx* is issued; that is when a *synclk_x9mx*? transition occurs, it jumps to location `Processing` and stays there non-deterministically in the interval [`0`, `ARBI_PERIOD`). It then switches to `Idle` by completing the function `arbitrate`(). We use a local boolean variable `restart_flag_by_arbi`[`i`] to express if `CPU`[`i`] needs to restart judged by automaton `Arbitrator`.

```
void arbitrate() {
  int i;
  if(voter_status[0] == 0) // voter 0 is good
    arbitrator_data = voter_output[0];
  else  // voter 1 is good
    arbitrator_data = voter_output[1];
  for(i = 0; i < 3; i++)
  {restart_flag_by_arbi[i] = 0;
   if
   { if(voter_status[0] == 0)// voter0 is good
      {restart_flag_by_arbi[i] = restart_flag_by_voter[i];
       if(retart_count_by_voter[i] == 6) // CPU i restart times exceeds 6
           {demotion[i] = 1; //switch off CPU i
            restart_flag_by_arbi[i] = 0; //CPU i does not need a restart
           }
      }
     else // voter1 is good
      {restart_flag_by_arbi[i] = restart_flag_by_voter[i + 3];
       if(restart_count_by_voter[i + 3] == 6)
           {demotion[i] = 1;
            restart_flag_by_arbi[i] = 0;
           }
      }
   }
 }
}
```

4 Parametric Constraints

4.1 The Desired Property

Verification of the above model in UPPAAL reveals that both watchdog and arbitrator can trigger a CPU to restart. This can solve the problems proposed in section 3. However, we also found unpleasant scenario regarding a CPU restart. Since there is no information intersection in between any two CPUs, and each voter independently judges that whether or not there exists a CPU restart, in the following section, we use a product automaton of these four automata: CPU[0], Voter[0], Arbitrator and Impulse to describe the phenomena, the parametric constraints and the proof.

With respect to CPU[0] recovery to normal state as soon as possible when it restarts, we expect that by setting the parameters, the current system satisfies the requirement–" When a CPU is in the startup phase, it should not restart again due to its watchdog timer overflow or a restart signal sent from the arbitrator ". More specifically, we prove that:

- $\mathcal{T}1$: CPU[0].Startup $\rightarrow$ Arbitrator.restart_flag_by_arbi[0] = 0
- $\mathcal{T}2$: CPU[0].Startup $\rightarrow$ CPU[0].y $<$ WD_UB

Here $\rightarrow$ means "imply". By testing a lot in UPPAAL with different sets of parameter values, we found that the above properties can be either satisfied or not. This gives us the intuition that the properties hold under certain parametric constraints. So, the interesting work now is to find the constraints and prove that for any parameters values that meets the constraints, the properties hold.

4.2 Parametric Constraints

As described before, there are a number of key timing parameters in the system: the time period CPU_PERIOD between *synclk_xms* and *synclk_5xms* impulse, the time period VOTER_PERIOD between *synclk_5xms* and *synclk_9xms* impulse, the time period ARBI_PERIOD between *synclk_9xms* and *synclk_xms* impulse, and the impulse cycle T. In addition to these, there is also the parameters n1 and n2, the reset time C, the upper bound RUB in CPU startup phase. These parameters have been set to different values and have been shown to influence the above property greatly.

The following lemma is needed to support our explanation to parametric constraints.

Lemma 4.1 *Function fault_check() of Voter[0] will judge that CPU[0] needs a restart or not, CPU[0] is normal or not, ect., and is executed in the transition of Voter[0] from location Processing to location Idle. The transition is labeled as S. For any cycle $\mathcal{M}$, let the time point that S happens is a, and the next cycle $\mathcal{M}+1$, the time point that S happens is b, then the time difference between a and b is (ARBI_PERIOD+CPU_PERIOD, T+VOTER_PERIOD).*

Proof. This lemma can be easily proved since the processing time of Voter[0] lies in the interval [0, ARBI_PERIOD). □

Below we give the constraints on the parameters, and show by counterexamples that they are necessary to prove that properties $\mathcal{T}1$ and $\mathcal{T}2$ do hold by the transitions of the network of automata. In the next section we will establish that the proposed constraints are also sufficient for correctness.

$$\mathtt{T} + \mathtt{VOTER\_PERIOD} \leq \mathtt{C} \tag{1}$$

$$\mathtt{WD\_UB} > \mathtt{RUB} \tag{2}$$

$$\mathtt{n1} > \lceil \frac{\mathtt{RUB} + \mathtt{C}}{\mathtt{T}} \rceil + 2 \tag{3}$$

$\mathtt{T} + \mathtt{VOTER\_PERIOD} < \mathtt{C}$ in inequality (1) is used to ensure that `fault_check`() of `Voter`[0] can find the update of variable `flag`[0], so as to update the related variables correctly. To illustrate this, suppose at time `t`, `Voter`[0] finishes the execution of functions `fault_check`() and `vote`(). The computation is done with `local_flag`[0] and `cpu_normal_by_voter`[0] respectively set to 1. At the same time but follows the execution, `CPU`[0] enters into location `Reset` with `flag`[0] set to 0. We denote this reset as $\mathcal{R}$ (This scenario can be easily found in UPPAAL). By lemma 4.1, it might be that at time point $\mathtt{t}_1$, $\mathtt{t}_1 = \mathtt{t} + \mathtt{T} + \mathtt{VOTER\_PERIOD} - \epsilon$, where ϵ is infinite small, `fault_check`() is executed for the first time since the time point `t`. However, if $\mathtt{T} + \mathtt{VOTER\_PERIOD} > \mathtt{C}$, then at time $\mathtt{t}_2$, $\mathtt{t}_2 = \mathtt{C} < \mathtt{t} + \mathtt{T} + \mathtt{VOTER\_PERIOD}$, `CPU`[0] transits to location `Startup` from location `Reset`, meanwhile sets `flag`[0] to 1. As a result, at time $\mathtt{t}_1$, since both values of `flag`[0] and `local_flag`[0] are 1, function `fault_check`() is not able to detect the reset $\mathcal{R}$. According to the `CPU` reset mechanism, when `cpu_normal_by_voter`[0] is equal to 1, `Voter`[0] allows for `n2` incorrect data before the judgement of `CPU`[0] restart. Since `CPU`[0] can stay in location `Startup` for `RUB` units and $\mathtt{n}_2 \ll \mathtt{n}_1$, with time elapsed the value of `restart_flag_by_arbi`[0] could be 1. So, there exits the scenario that the `CPU` can transit from `Startup` to `Reset`, which violates property $\mathcal{T}1$.

Inequality (2) is also needed to avoid a restart of `CPU`[0] in location `Startup`. Suppose it is not satisfied, because `CPU`[0] enters into location `Startup` with `y` set to 0, then after `WD_UB` time units, watchdog timer may expire the constant `WD_UB`. This causes `CPU`[0] to move to location `Reset` again and hence violates property $\mathcal{T}2$.

Inequality (3) is used to give the maximum number of incorrect data of `CPU`[0] that `Voter`[0] can tolerate since it judges that this `CPU` is non-normal. Consider the case that `CPU`[0] stays in location `Reset` and `y` is equal to 0,`cpu_normal_by_voter`[0] is equal to 0. Since *synclk_5xms* is generated in a `T` cycle, and the value of `cpu_normal_by_voter`[0] does not change in location `Reset` and `Startup`, `Voter`[0] therefore counts the number of incorrect data until the number equals $\frac{\mathtt{RUB}+\mathtt{C}}{\mathtt{T}} + 1$. It further judges that this CPU needs to restart. Violation of $\mathtt{n1} > \lceil \frac{\mathtt{RUB}+\mathtt{C}}{\mathtt{T}} \rceil + 1$ will lead to the phenomenon of a `CPU`[0] restart in location `Startup`. In the inequality "+2 " is used to give more allowance.

5 Correctness Proof

Since UPPAAL can not deal with the verification of the parametric constraints but only of the fixed parameters values, in the following, we prove $\mathcal{T}1$ and $\mathcal{T}2$ manually

using a product automaton $\mathcal{A}$ of these automata: CPU[0], Impulse, Voter[0] and Arbitrator.

To prove $\mathcal{T}1$, we need a list of additional invariants:

(i) I1 : CPU[0].Reset $\wedge$ CPU[0].y $>$ T $\rightarrow$ cpu_voter_fifo[0] $=$ 0

(ii) I2 : CPU[0].Startup $\wedge$ CPU[0].y $>$ 2T $\rightarrow$ Voter[0].local_flag[0] $=$ 1

(iii) I3 : CPU[0].Idle $\rightarrow$ Voter[0].local_flag[0] $=$ 1

(iv) I4 : CPU[0].Processing $\rightarrow$ Voter[0].local_flag[0] $=$ 1

(v) Let α be the value of cpu_normal_by_voter[0] as CPU[0] transits from location Reset to location Startup, then
I5 : CPU[0].Startup $\rightarrow$ Voter[0].cpu_normal_by_voter[0] $= \alpha$

(vi) I6 : CPU[0].Reset $\wedge$ CPU[0].y $>$ 2T $\rightarrow$ Voter[0].cpu_normal_by_voter[0] $=$ 0

(vii) I7 : CPU[0].Reset $\wedge$ CPU[0].y $>$ 2T $\rightarrow$ restart_flag_by_voter[0] $=$ 0

We now prove these lemmas.

Lemma 5.1 I1 : CPU[0].Reset $\wedge$ CPU[0].y $>$ T $\rightarrow$ cpu_voter_fifo[0] $=$ 0

Proof.

To prove this lemma, first we need to prove an additional invariant I10. An auxiliary boolean variable taken1 initialized to 0 is required to record if the transition that synchronize with the action *synclk_xms* is taken or not. The assignment of the transition has an extra part: taken1 := 1. For all the discrete transitions from location Reset to the other locations, taken1 is set to 0. We first prove the below invariant:

I10 : CPU[0].Reset $\wedge$ taken1 $=$ 1 $\rightarrow$ cpu_voter_fifo[0] $=$ 0

In order to prove that I10 is an invariant, it suffices to prove that it holds initially and is preserved by all discrete transitions as well as by all time delay steps of $\mathcal{A}$.

We consider all the transitions that arrive at location Reset, and all the transitions that update the value of taken1 or cpu_voter_fifo[0] when CPU[0] stays in location Reset. For all the other transitions from state s, $s \models$ I10, since the left side of the formula is equivalent to false after the transition, it is trivial to observe that I10$'$ holds.

Let φ_{1d} be the time delay transition in location Reset, where d is a real number to let all the clocks increment with this value. Let φ_2 be the transition that synchronize with the action *synclk_xms*, φ_3 be the transition that receives a signal dsp_restart. The left and the right transitions from location Idle to location Reset are φ_4 and φ_5. The left and the right transitions from location Processing to location Reset are φ_6 and φ_7. The top and the bottom transitions from location Startup to Reset are φ_8 and φ_9.

Initially, since taken1 is 0, I10 is true. Suppose state $s \models$ I10, we now prove

for any of the above transition φ, $s \xrightarrow{\varphi} s'$, $s' \models \texttt{I10}'$. Namely, $\texttt{I10} \wedge \varphi \rightarrow \texttt{I10}'$.

Ad1 Assume $\texttt{I10} \wedge \varphi_{1d}$, since transition φ_{1d} does not change any value of the variables appearing in the implication, clearly $\texttt{I10}'$ holds.

Ad2 Assume $\texttt{I10} \wedge \varphi_2$.

If taken1 is equal to 0, then $\texttt{taken1}' = 1$ and $\texttt{cpu\_voter\_fifo}[0]' = 0$.

If taken1 is equal to 1, then $\texttt{taken1}' = 1$ and $\texttt{cpu\_voter\_fifo}[0]' = 0$. So $\texttt{I10}'$ holds.

Ad3 Assume $\texttt{I10} \wedge \varphi_3$, clearly $\texttt{I10}'$ holds.

Ad4 For any transition from φ_4 to φ_9, since $\texttt{taken1}'$ is equal to 0, $\texttt{I10}'$ holds.

We conclude that I10 is an invariant. Because *synclk_xms* is generated in a T cycle, we have when $\texttt{y} > \texttt{T}$, taken1 is equal to 1. So, I1 is an invariant. We conclude the proof.

□

Lemma 5.2 $\texttt{I2} : \texttt{CPU}[0].\texttt{Startup} \wedge \texttt{CPU}[0].\texttt{y} > 2\texttt{T} \rightarrow \texttt{Voter}[0].\texttt{local\_flag}[0] = 1$

We can easily prove this lemma from lemma 4.1.

Lemma 5.3 $\texttt{I3} : \texttt{CPU}[0].\texttt{Idle} \rightarrow \texttt{Voter}[0].\texttt{local\_flag}[0] = 1$

Lemma 5.4 $\texttt{I4} : \texttt{CPU}[0].\texttt{Processing} \rightarrow \texttt{Voter}[0].\texttt{local\_flag}[0] = 1$

Lemma 5.3 and lemma 5.4 follow from lemma 5.2.

Lemma 5.5 $\texttt{I5} : \texttt{CPU}[0].\texttt{Startup} \rightarrow \texttt{Voter}[0].\texttt{cpu\_normal\_by\_voter}[0] = \alpha$

Proof. After the transition from location Reset to location Startup, the value of flag[0] is set to 1. By lemma 5.1 and constraint 1, $\texttt{cpu\_voter\_fifo}[0]$ is equal to 1 in location Startup. Since the value of $\texttt{cpu\_normal\_by\_voter}[0]$ can only be updated either in the case CPU[0] reads a correct data, namely $\texttt{cpu\_voter\_fifo}[0] = 1$, or in the case there is a value change of flag[0], namely $\texttt{flag}[0] = 0 \wedge \texttt{local\_flag}[0] = 1$, we have for any other discrete transitions or time delay transitions, $\texttt{cpu\_normal\_by\_voter}[0]' = \alpha$. □

Lemma 5.6 $\texttt{I6} : \texttt{CPU}[0].\texttt{Reset} \wedge \texttt{CPU}[0].\texttt{y} > 2\texttt{T} \rightarrow \texttt{cpu\_normal\_by\_voter}[0] = 0$

Proof. Analogous to the proof of the invariant I1, we also need to prove an additional invariant I60. Let φ_{10} be the transition in $\mathcal{A}$ that transits from location Processing to Idle. We introduce an auxiliary boolean variable taken2 to record if the transition φ_{10} is taken or not. The variable is initially set to 0. For all the discrete transitions from location Reset to the other locations, taken2 is set to 0. And the assignment of the transition φ_{10} has an extra part: $\texttt{taken2} := 1$. We now prove the below invariant:

$$\texttt{I60} : \texttt{CPU}[0].\texttt{Reset} \wedge \texttt{taken2} = 1 \rightarrow \texttt{cpu\_normal\_by\_voter}[0] = 0$$

Similarly, we consider all the transitions in $\mathcal{A}$ that arrive at location Reset, and all the transitions that update the value of taken2 or $\texttt{cpu\_normal\_by\_voter}[0]$ when

CPU0 stays in location Reset. For all the other transitions from state s, $s \models$ I60, since the left side of the formula is equivalent to false after the transition, it is trivial to observe that I60′ holds.

Then, we have the following implications:

(i) $\texttt{I60} \wedge \varphi_{1d} \rightarrow \texttt{I60}'$.
(ii) $\texttt{I60} \wedge \varphi_2 \rightarrow \texttt{I60}'$.
(iii) $\texttt{I60} \wedge \varphi_3 \rightarrow \texttt{I60}'$.
(iv) $\texttt{I60} \wedge \varphi_4 \rightarrow \texttt{I60}'$.
(v) $\texttt{I60} \wedge \varphi_5 \rightarrow \texttt{I60}'$.
(vi) $\texttt{I60} \wedge \varphi_6 \rightarrow \texttt{I60}'$.
(vii) $\texttt{I60} \wedge \varphi_7 \rightarrow \texttt{I60}'$.
(viii) $\texttt{I60} \wedge \varphi_8 \rightarrow \texttt{I60}'$.
(ix) $\texttt{I60} \wedge \varphi_9 \rightarrow \texttt{I60}'$.
(x) $\texttt{I60} \wedge \varphi_{10} \rightarrow \texttt{I60}'$.

Ad1 Since any of the transitions: φ_{1d}, φ_2 and φ_3, does not change the value of taken2, neither of cpu_normal_by_voter[0], clearly I60′ holds.
Ad2 For any of the transitions from φ_4 to φ_9, clearly I60′ holds since $\texttt{taken2}' = 0$.
Ad3 Assume $\texttt{I6} \wedge \varphi_{10}$. We consider two cases (1) $\texttt{taken2} = 0$ (2) $\texttt{taken2} = 1$.

Also, we introduce two auxiliary boolean variables from1 and from2, to indicate that from which location (excluding the location Reset itself) Reset is reached. Initially, these two variables are set to 0. For the transition from location Reset to Startup, it has an extra update $\texttt{from1} := 0 \wedge \texttt{from2} := 0$. If the location Reset is reached from location Startup, from1 is set to 1. So for the two transitions from location Startup to location Reset, they respectively have the update $\texttt{from1} := 1$. Similarly, if the location Reset is reached from location Idle or Processing, from2 is set to 1. Therefore, we have the update $\texttt{from2} := 1$ for any transition from location Idle or Processing to location Reset.

CASE 1. When $\texttt{taken2} = 0$. The combinational values of from1 and from2 have three cases:
(1) $\texttt{from1} = 0 \wedge \texttt{from2} = 0$.

Since the value of cpu_normal_by_voter[0] is initially 0, and any of the transitions φ_{1d}, φ_2 and φ_3 does not change this value, we observe that also after transition φ_{10}, $\texttt{cpu\_normal\_by\_voter[0]}' = 0$.
(2) $\texttt{from1} = 1 \wedge \texttt{from2} = 0$

Assume when CPU[0] stays in location Startup, $\texttt{cpu\_normal\_by\_voter[0]} = 1$. By lemma 5.5, cpu_normal_by_voter[0] is equal to 1 after the CPU transits to Reset from Startup. Since the value of cpu_normal_by_voter[0] is not changed by any of the transitions φ_{1d}, φ_2 and φ_3, we observe that also after transition φ_{10}, the value of $\texttt{cpu\_normal\_by\_voter[0]}'$ is 1. This result contradicts our assumption I60 when $\texttt{taken2} = 1$. So the case that when CPU[0] stays in location Startup, the value of

cpu_normal_by_voter[0] being 1 does not exist. Assume that when CPU[0] stays in location Startup, cpu_normal_by_voter[0] is equal to 0. We can prove I60′ holds.
(3) from1 = 0 ∧ from2 = 1

In this case, flag[0] is equal to 0. Further by lemmas 5.3 and 5.4, we have local_flag[0] = 1. Therefore Voter[0].cpu_normal_by_voter[0]′ = 0.

CASE 2. When taken2 = 1. In this case, Voter[0].local_flag[0] is equal to 0 and cpu_voter_fifo[0] = 0. Therefore for any value case of from1 and from2, the value of Voter[0].cpu_normal_by_voter[0] is not changed.
Further by the fact that I60 initially holds, we conclude that I60 is an invariant. By lemma 4.1, we have when y > 2T, taken2 is equal to 1. So I6 is an invariant.□

Lemma 5.7 I7 : CPU[0].Reset ∧ CPU[0].y > 2T → restart_flag_by_voter[0] = 0

Lemma 5.8 *When CPU[0] is in startup process, in other words, CPU[0] stays in location Startup in the CPU automaton, then CPU[0] should not get a restart signal from the arbitrator. That is, we have the invariant*
$\mathcal{T}1$: CPU[0].Startup → Arbitrator.restart_flag_by_arbi[0] = 0

This lemma follows from lemma 5.6 and 5.7, as well as constraint 1 and constraint 3.

Theorem 5.9 *When CPU[0] is in the startup phase, it should not restart again due to its watchdog timer overflow or a restart signal sent from the arbitrator. That is, property $\mathcal{T}$ holds.*
$\mathcal{T}$: $\mathcal{T}1 \wedge \mathcal{T}2$

Proof. When clock y equals WD_UB, CPU[0] will switch to location Reset. Because of constraint 2, watchdog timer will not expire the constant WD_UB in location Startup. So the transition to Reset can not be taken, $\mathcal{T}2$ holds. Further by lemma 5.8, we conclude the proof. □

6 Conclusion

We present our preliminary work on an improved triple modular fault tolerant system taken from the aerospace field. We expect that the improved system should satisfy the engineering requirements, one of the most important one is that " When a CPU is in the startup phase, it should not restart again due to its watchdog timer overflow or a restart signal sent from the arbitrator ". Rather than verifying the correctness of the system for a single, specific choice of parameter values like in UPPAAL, we derive (by hand) general constraints on the parameters which ensure correctness and give the correctness proof. Validity of the correctness property will give the engineers a guideline for choosing parameters values in real hardware implementation.

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*, 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)*, December 14 2003, Mumbai, India. pp. 57-70. Electronic Notes in Theoretical Computer Science 118.

[4] F. Schneider, S.M. Eastbrook, J.R. Callahan and G.J. Holzmann. Validating requirements for fault tolerant systems using model checking. In: *Int. Conf. on Requirements Engineering*, 1998.

[5] C. Bernardeschi, A. Fantechi and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability (STVR)*, 12(4):251C275, 2002.

[6] Miaomiao Zhang, Zhiming Liu and Dang Van Hung. Design and Verification of a Fault Tolerant System. *Technical Report, UNU-IIST, Macau, 2007.*

[7] Z. Liu and M. Joseph. Specification and verification of fault-folerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems*. 21(1): 46-89 (1999).

[8] Z. Liu, M. Joseph. Verification of fault folerance and real time.*FTCS 1996*: 220-229, IEEE Computer Society, 1996.

[9] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, pages 183–235, 1994.

[10] 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)*, Bertinoro, Italy, September 13-18, LNCS 3185, pages 200C236. Springer, 2004.

[11] E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Univ. of Twente, The Netherlands, 2004.

[12] Ansgar Fehnker, Frits W. Vaandrager, Miaomiao Zhang. Modeling and Verifying a Lego Car Using Hybrid I/O Automata. QSIC 2003: 280-289.

[13] M. Zhang, V.H. Dang. Formal analysis of streaming downloading protocol for system upgrading. *Electr. Notes Theor. Comput. Sci.* 164(3): 205-224 (2006)

[14] B. Gebremichael, F.W. Vaandrager, M. Zhang. Analysis of the Zeroconf Protocol Using UPPAAL. In *Proc. EMSOFT 2006*: 242-251.

[15] K. Yorav. *Exploiting syntactic structure for automatic verification*. PhD thesis, The Technion, Israel Insitute of Technology, 2000.