Aerospace Technology Congress, 8-9 October 2019, Stockholm, Sweden
Swedish Society of Aeronautics and Astronautics (FTF)

Triple Modular Redundancy based on Runtime Reconfiguration and Formal Models of Computation

Ricardo Bonna[1], Denis S. Loubach[2], Ingo Sander[3], and Ingemar Söderquist[4]

E-mail: rbonna@fem.unicamp.br; dloubach@ita.br; ingo@kth.se; ingemar.soderquist@saabgroup.com
[1]Advanced Computing, Control & Embedded Systems Lab, University of Campinas – UNICAMP, Campinas, SP, Brazil - 13083-860
[2]Department of Computer Systems, Computer Science Division, Aeronautics Institute of Technology – ITA, São José dos Campos, SP, Brazil - 12228-900
[3]Division of Electronics/School of EECS, KTH Royal Institute of Technology, SE-164 40, Kista, Sweden
[4]Business Area Aeronautics, Saab AB, Linköping, Sweden

Abstract

Runtime reconfiguration is one promising way to mitigate for increased failure rate and thereby it fulfills safety requirements needed for future safety-critical avionics systems. In case of a hardware fault, the system is able, during runtime, to automatically detect such fault and redirect the functionality from the defective module to a new safe reconfigured module, thus minimizing the effects of hardware faults. This paper introduces a high level abstraction architecture for safety-critical systems with runtime reconfiguration using the triple modular redundancy and the synchronous model of computation. A modeling strategy to be used in the design phase supported by formal models of computation is also addressed in the paper. The triple modular redundancy technique is used for detecting faults where, in case of inconsistency in one of the three processors caused by a fault, a new processor is reconfigured based on a software or hardware reconfiguration, and it assumes the tasks of the faulty processor. The introduced strategy considers that no other fault occurs during the reconfiguration of a new processor.

Keywords: safety-critical systems, triple modular redundancy, runtime reconfiguration, formal models of computation.

1 Introduction

The safety and reliability of modern avionics may be threatened by trends that are largely driven by high-volume commercial applications, *e.g.* environmental concerns as restriction of hazardous substances (RoHS) directive that forced the removal of lead from commercial electronics and solders.

Another trend arises from technological innovation in commercial electronics. The effort to place more functionality and performance in smaller packages and lower power has led to ever-shrinking device geometries down to deep submicron dimensions with new physical failure mechanism that affect the wear out of semiconductor devices. Additionally, small geometries negatively affect the susceptibility of the semiconductor device to atmospheric radiation.

One of the next big challenges for the avionics industry is to address these trends that increase failure rate and thereby affect safety and reliability. Bieber *et al.* [1] points out runtime reconfiguration as one of the big challenges for the future generation of integrated modular avionic (IMA) systems. In the event of a hardware failure, the system is able to reallocate the functionalities from the faulted module into a safe module, thus limiting the effects of a hardware failure on aircrafts.

Perhaps the most important component of a runtime reconfigurable safety-critical system is the *fault detection mechanism*. One of such mechanisms is the *triple modular redundancy* (TMR), capable of detecting and mask possible faults in a system, improving reliability [2]. In such architecture, depicted in Figure 1, three processes execute the same functionality, and a majority voting mechanism selects the output that most occurs. If one of the processors fails to produce the correct output, possibly due to a single event upset (SEU), the voting mechanism masks such fault with the output of the other two processes. Any number of processors can be used in modular redundancies, however the minimum number of redundant processors necessary to detect and mask a fault is three.

Systems with triple modular redundancy are tolerant to both *transient faults*, *i.e.* faults that appear for a very short period of time and then disappear, and *single permanent fault*, *i.e.* faults that remains active for a long or possibly indefinite

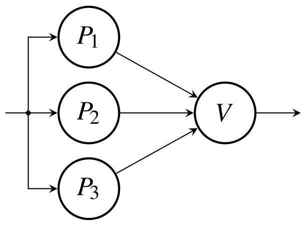


Figure 1: Triple modular redundancy architecture. V represents the voting mechanism and P_1, P_2 and P_3 represent three processes with the same functionality.

amount of time. However, faults in the voting mechanism lead to errors, making the voter a *single point of failure*. To improve reliability, three voters can be used instead of one. In that case if a fault occurs on the voters, the system can mask such fault, thus eliminating the single point of failure.

In view of this, this paper proposes a *high level abstraction architecture for safety-critical systems* with *runtime reconfiguration* (RTR) using the triple modular redundancy and the synchronous (SY) model of computation (MoC). Such architecture is composed of one fault detection mechanism, several runtime reconfigurable processes, and a control device to manage the reconfiguration process. Differently from the traditional triple modular redundancy architectures, the proposed architecture can mask multiple permanent faults, provided that no two faults occur in a small time interval defined by the reconfiguration time of a new module.

2 Models of Computation

Models of computation are a collection of rules dictating the semantics of execution and concurrency in computational systems. A common framework to classify and compare different MoCs is the *tagged signal model* [3]. In such framework, MoCs are a set of processes acting on signals, according to the following definitions.

Definition 1 (Signal)**.** *In the tagged signal model, a signal $s \in S$ is a set of events $e_i = (t_i, v_i)$ composed by a tag $t_i \in T$ and a value $v_i \in V$. The set of signals S is a subset of $T \times V$.*

Definition 2 (Process)**.** *In the tagged signal model, a process P is a set of possible behaviors that defines relations between input signals $s_i \in S^I$ and output signals $s_o \in S^O$. The set of output signals is given by the intersection between the set of input signals and the process $S^O = S^I \cap P$. A functional process is a process described by a single value mapping $f : S^I \to S^O$ and describes either one behavior or no behavior at all.*

The tagged signal model classifies MoCs as being *timed* or *untimed*. In a timed MoC, all events in all signals can be ordered based on its tags, *i.e.* the set of tags T is totally ordered. In an untimed MoC, the set of tags T is partially ordered, *i.e.* events can only be locally ordered.

2.1 Synchronous (SY) MoC

The synchronous MoC belongs to the class of timed MoCs and it is based on the *perfect synchrony hypothesis*, which states that neither computation nor communication consumes time. As a consequence, every signal is synchronized, meaning that for any event in any signal, there is an event with the same tag in every other signal. This allows the representation of signals as a list of values in which the position of each value in the list represents its tag, *i.e.*, $s[k] = v$ with $k \in T$ and $v \in V$. Another important property of the synchronous MoC is that the absence of an event is well defined. Such phenomenon is defined as an event, with some tag $t \in T$, whose value is the absent value $\perp \in V$, *i.e.* $e = (t, \perp)$.

Although the perfect synchrony hypothesis is not physically feasible, the synchronous MoC works well when modeling clocked-based systems, provided that both computation and communication are fast enough to fit within one evaluation cycle.

3 Modeling TMR with RTR

A triple redundancy architecture proposal using runtime reconfiguration is illustrated in Figure 2.

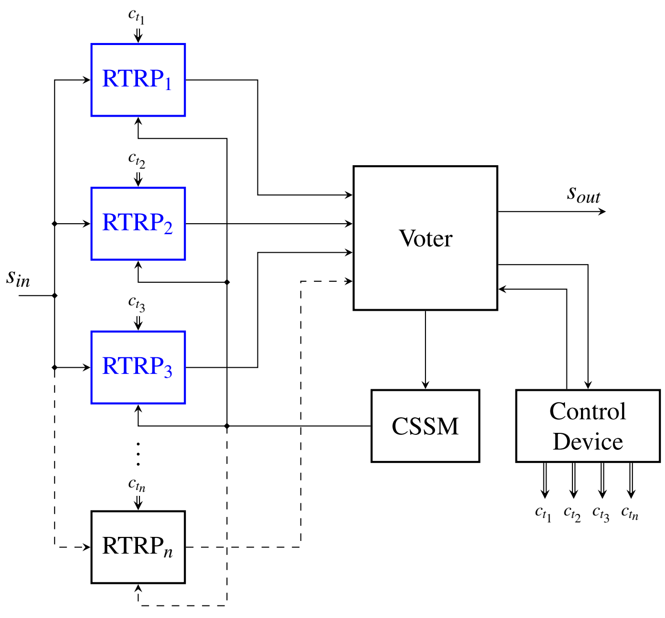


Figure 2: Triple redundancy architecture with runtime reconfiguration applicable to safety-critical systems.

It works as follows: three runtime reconfigurable processors, RTRP$_{1..3}$, are configured with the same functionality and, given the same input signal, should provide the same output signal. Knowing this property, the `Voter` compares the results and possibly the states outputted by each processor. If, by any chance, one of the processor's output differs from the other two, the `Voter` assumes that there must be a fault in such processor and, therefore, a new processor must take its place. In view of this, the `Voter` sends a signal to the `Control Device` informing which processor is malfunctioning, so that the `Control Device` can allocate a new RTRP$_x$ to assume the failed processor's task.

The newly allocated processor must then synchronize its states with the two remaining RTRP that are still executing in order to mask the fault. To do that, the first time a processor executes, it loads the current states from a *current state shared memory* (CSSM), which is represented as a *delay* using the SY MoC and can be physically implemented as a set of processor registers.

Similarly to N-modular redundancy (NMR) with spares, when one of the processors becomes unreliable, *i.e.* starts to produce inconsistent results, it is replaced by a spare processor. However, here the *spare processors*, represented by $\texttt{RTRP}_n$, with $n > 3$, can be initially loaded with less critical applications that can be overloaded when needed, providing better usability of resources.

For such architecture to work properly it is assumed that neither the `Voter` nor the `Control Device` fails, thus both of these devices are single point of failure for this system. Although it is possible to eliminate the `Voter`'s single point of failure by using three voters, there can only be one reconfiguration manager, represented by the `Control Device`.

When two `RTRP`s fail either at the same time, or in a time window smaller than the necessary time to reconfigure a new processor, we say the triple redundancy system fails. To show how likely such failure occurs, consider that all `RTRP` have the same failure rate and, for every cycle, the probability of failure of an $\texttt{RTRP}_j$ is $p(F_j) = \rho$. Consider also that it takes m clock cycles to reconfigure a new `RTRP` in case of a failure. Then, the probability $p(F_a|F_b)$ of some $\texttt{RTRP}_a$ to fail in a time window of $m+1$ cycles (including the cycle in which the fault was detected), provided that some $\texttt{RTRP}_b$ has already failed, is given by

$$p(F_a|F_b) = 1-(1-\rho)^{2(m+1)} \tag{1}$$

Therefore, the probability of failure of our triple redundancy architecture with RTR is given by

$$\begin{aligned} p(F_a \cap F_b) &= p(F_b)p(F_a|F_b) \\ &= \rho(1-(1-\rho)^{2(m+1)}) \end{aligned} \tag{2}$$

We define the *ratio of improvement* R_I as being the probability of failure of a single `RTRP` divided by the probability of failure of our triple redundancy architecture, given by (2). The larger the ratio of improvement, the more fail-safe the triple redundancy with RTR is when compared to an architecture with a single processor. Such ratio of improvement is given by

$$R_I = \frac{1}{1-(1-\rho)^{2(m+1)}} \tag{3}$$

The ratio of improvement R_I shows the importance of the reconfiguration time m in the robustness of the triple redundancy architecture with RTR. In case of a fault in one of the processors, the system can still continue to perform correctly with two processors while another `RTRP` is being reconfigured, however it becomes vulnerable to a second fault in this time window. Therefore, the fastest the reconfiguration, the less vulnerable the system is. Traditional triple redundancy architectures (without RTR) are immune to a single processor permanent fault, however they are vulnerable to multiple faults.

3.1 Runtime Reconfigurable Process (RTRP)

We start to model the triple redundancy architecture by modeling what we are calling a runtime reconfigurable process, similar to the architecture presented in [4]. We consider that for such process to be in its most general form, it must have some internal memory to store its states, and it must take into account reconfiguration time. When a new `RTRP` is being reconfigured, it takes a number $m \in \mathbb{N}$ of clock cycles, proportional to the size of the functionality bitstreams, to perform reconfiguration before it is able to execute for the first time. When the `RTRP` executes for the first time, it must synchronize its internal memory with the internal memory of the other two `RTRP` executing the same functionality. In order to achieve that, we added an extra input, a *synchronization input* $\bar{x}$, so that when the `RTRP` executes for the first time, it gets its initial state from the synchronization input.

We model an `RTRP` as a finite state machine with $x[k] \in \mathbb{S}$ being the state vector, $s_{in}[k] \in V^I$ the inputs, and $y[k] \in V^O$ the outputs at an instant k, with V^I and V^O the set of values from the input and output signals respectively. The functionality of an `RTRP` is represented by a state transition function $f : \mathbb{S} \times V^I \to \mathbb{S}$ and an output function $g : \mathbb{S} \times V^I \to V^O$. Such functionality is stored in a configuration memory and can be changed by a control input signal c_t that is responsible for the reconfiguration. The control signal c_t is responsible for changing both f and g when needed, and this change takes m clock cycles to finish. Finally, the processor can execute for the first time with the new configuration.

A representation of an `RTRP` is shown in Figure 3. Feedback loops, along with delay blocks (represented by z^{-1}), are used to represent memories following the pattern: the blue rounded delay represents configuration memory, the squared black delay represents `RTRP`'s internal memory, and the dashed delay represents a virtual count down to simulate reconfiguration time.

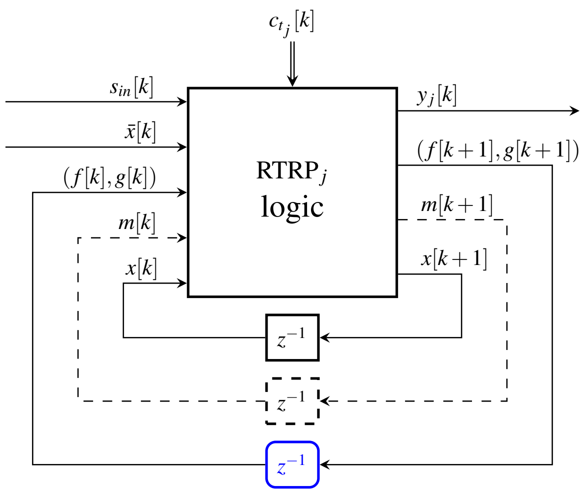


Figure 3: `RTRP` internal schematics, i.e. function application logic. The delays represented by z^{-1} are used to store state vector x, the pair of functions (f, g) and the reconfiguration countdown m.

Let $c_t[k]$ be the control signal in the instant k, which can be either the absent value $\perp$ or a 3-tuple $(\mathbf{f},\mathbf{g},\mathbf{m})$, with $\mathbf{f}:\mathbb{S}\times V^I\rightarrow\mathbb{S}$, $\mathbf{g}:\mathbb{S}\times V^I\rightarrow V^O$ and $\mathbf{m}\in\mathbb{N}$. $c_t[k]=\perp$ indicates no reconfiguration is needed and, therefore, the process can execute the current configuration normally, if able to. When $c_t[k]=(\mathbf{f},\mathbf{g},\mathbf{m})$, a reconfiguration must be performed and the functions $\mathbf{f}$ and $\mathbf{g}$ will replace the current configuration. Such reconfiguration process takes $\mathbf{m}$ cycles to finish.

At any instant k, the functionality of a runtime reconfigurable process is given by the pair $(f[k],g[k])$. Such pair is stored in the configuration memory until a reconfiguration request is received via c_t. To represent such behavior, the functionality transition function is given by

$$(f[k+1],g[k+1])=\begin{cases}(\mathbf{f},\mathbf{g}) & \text{if } c_t[k]=(\mathbf{f},\mathbf{g},\mathbf{m})\\ (f[k],g[k]) & \text{if } c_t[k]=\perp\end{cases}\tag{4}$$

To represent the time spent to perform the reconfiguration of a process, the countdown variable $m[k]\in\mathbb{N}$ stores how many cycles are left to finish the reconfiguration. $m[k]>0$ indicates that the process is reconfiguring at the instant k and, therefore, cannot execute. (5) represents the behavior of the countdown signal m.

$$m[k+1]=\begin{cases}\mathbf{m}-1 & \text{if } c_t[k]=(\mathbf{f},\mathbf{g},\mathbf{m})\\ m[k]-1 & \text{if } c_t[k]=\perp \text{ and } m[k]>0\\ 0 & \text{if } c_t[k]=\perp \text{ and } m[k]=0\end{cases}\tag{5}$$

When a reconfiguration is being performed, *i.e.* when $m>0$, the RTRP outputs the absent value $\perp$ for both the next state $x[k+1]$ and the output $y_j[k]$. The first time the RTRP executes after reconfiguration, it uses the state input $\bar{x}$ as initial states. Afterwards, it keeps executing with its internal state x. The state transition function at any instant k is given by

$$x[k+1]=\begin{cases}\perp & \text{if } c_t[k]\neq\perp \text{ or } m[k]>0\\ f[k](\bar{x}[k],s_{in}[k]) & \text{else if } x[k]=\perp\\ f[k](x[k],s_{in}[k]) & \text{otherwise}\end{cases}\tag{6}$$

with $\bar{x}[k]$ being the value of the states stored in CSSM in the instant k. The output of a runtime reconfigurable process at any instant k is given by

$$y_j[k]=\begin{cases}\perp & \text{if } c_t[k]\neq\perp \text{ or } m[k]>0\\ g[k](\bar{x}[k],s_{in}[k]) & \text{else if } x[k]=\perp\\ g[k](x[k],s_{in}[k]) & \text{otherwise}\end{cases}\tag{7}$$

Initial values $f[0]$, $g[0]$, $m[0]$ and $x[0]$, indicating the initial configuration and states of each RTRP, must be provided.

3.2 Voter

The Voter's task is to compare the outputs of the three RTRPs that are currently active, and alert the Control Device when one of the outputs differs from the other two. Figure 4 shows the voter inputs and outputs.

The input c_v is responsible to select the three currently active RTRP, so that the voter can compare their results and, in case of any inconsistency, it informs the Control Device about the failed RTRP through the signal r. The Voter outputs the most occurring of the RTRP results through s_{out} and send the current RTRP state to CSSM via x_v. Let s_j be the signal that carries the output and the states of RTRP$_j$. Events from c_v and s_j are defined as follows.

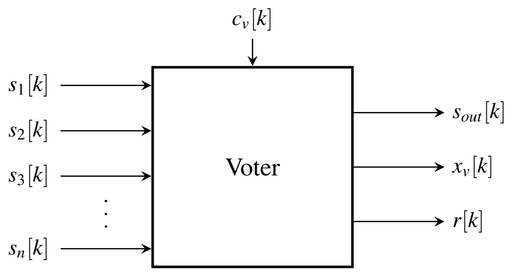


Figure 4: Voter with inputs and outputs.

$$c_v[k]=(a,b,c),\quad a,b,c\in\{1,2,\ldots,n\}\tag{8}$$
$$s_j[k]=(y_j[k],x_j[k]),\quad j\in\{1,2,\ldots,n\}\tag{9}$$

The outputs s_{out} and x_v are modeled as follows.

$$(s_{out}[k],x_v[k])=\begin{cases}(y_a[k],x_a[k]) & \text{if } y_a[k]=y_b[k]\\ & \text{or } y_a[k]=y_c[k]\\ (y_b[k],x_b[k]) & \text{if } y_b[k]=y_c[k]\end{cases}\tag{10}$$

The signal r is used to inform the Control Device, in case of a failure, which RTRP failed. If the results from the three active RTRPs are consistent in instant k, $r[k]$ assumes the absent value, otherwise it assumes the number of the faulted RTRP. Thus, the output r is modeled as follows.

$$r[k]=\begin{cases}\perp & \text{if } y_a[k]=y_b[k]=y_c[k]\\ a & \text{if } y_a[k]\neq y_b[k]=y_c[k]\\ b & \text{if } y_b[k]\neq y_a[k]=y_c[k]\\ c & \text{if } y_c[k]\neq y_a[k]=y_b[k]\end{cases}\tag{11}$$

3.3 Control Device

Finally, the Control Device is responsible for reconfiguring new RTRP based on the signal r received from the Voter, indicating which RTRP is not producing a correct answer. The Control Device keeps track of which RTRPs are active and, depending on the value received through the signal r, it performs a state transition to a new state indicating the active RTRP. Figure 5 shows the Control Device internal schematics.

Every time the Control Device performs a state transition, meaning an inconsistency was detected by the Voter, it sends the new configuration to the RTRP through signals c_{t_n}, and it waits m clocks, representing the time it takes to finish the reconfiguration of an RTRP, before being able to reconfigure a new RTRP in case of another inconsistency. The output/input signal m keeps track of how many clock cycles are left to finish a reconfiguration of a new RTRP, and it is modeled as

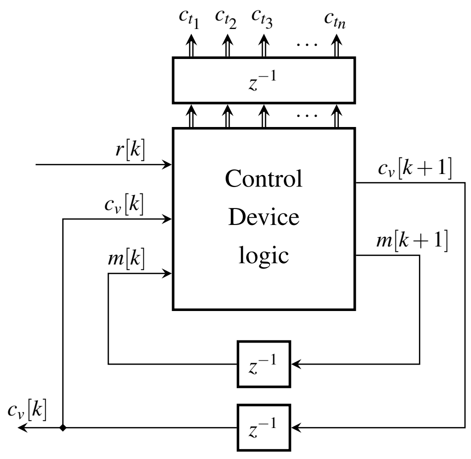


Figure 5: `Control Device` *internal schematics.*

follows.

$$m[k+1] = \begin{cases} \mathbf{m} & \text{if } r[k] \neq \perp \text{ and } m[k] = 0 \\ m[k]-1 & \text{if } m[k] > 0 \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

The output c_v carries the current three active RTRPs as a tuple, such as in (8). The behavior of the `Control Device` regarding the output c_v is modeled as follows.

$$c_v[k+1] = \begin{cases} c_v[k] & \text{if } r[k] = \perp \text{ or } m[k] > 0 \\ h(c_v[k], r[k]) & \text{if } r[k] \neq \perp \text{ and } m[k] = 0 \end{cases} \quad (13)$$

with $h((a,b,c),r)$ given by

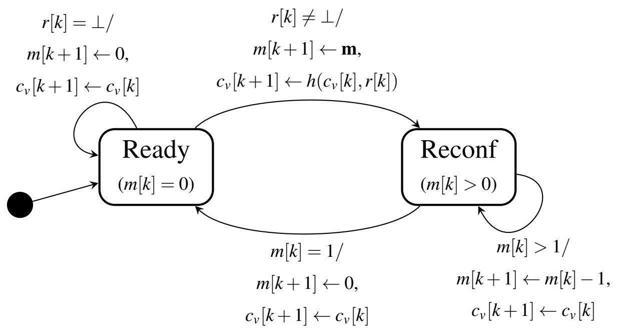
$$h((a,b,c),r) = \begin{cases} (\max(a,b,c)+1,b,c) & \text{if } r = a \\ (a,\max(a,b,c)+1,c) & \text{if } r = b \\ (a,b,\max(a,b,c)+1) & \text{if } r = c \end{cases} \quad (14)$$

(12) and (13) define a behavior that is represented graphically in Figure 6. While $m[k] > 0$, the `Control Device` is in the “Reconf” state, meaning a new RTRP is being reconfigured, and any reconfiguration request that is sent through r is ignored while in this state. After the reconfiguration is finished ($m[k] = 0$), the `Control Device` returns to the “Ready” state, awaiting for a new reconfiguration request.

Finally, the control outputs c_{t_j} behave as follows: when the transition from Ready to Reconf is *taken*, *i.e.* $r[k] \neq \perp$ and $m[k] = 0$, the `Control Device` outputs a reconfiguration signal given by the 3-tuple $(\mathbf{f},\mathbf{g},\mathbf{m})$ to the output $j = \max(c_v[k]) + 1$ (the next available spare RTRP); in any other case, it outputs $\perp$. Such behavior is given by

$$c_{t_j}[k+1] = \begin{cases} (\mathbf{f},\mathbf{g},\mathbf{m}) & \text{if } r[k] \neq \perp \text{ and } m[k] = 0 \\ & \quad \text{and } \max(c_v[k]) + 1 = j \\ \perp & \text{otherwise} \end{cases} \quad (15)$$

Initial conditions to m, c_v and c_{t_j} must be provided. As a general rule, we use the following initial conditions: $m[0] = 0$, $c_v[0] = (1,2,3)$ and $c_{t_j}[0] = \perp$. These initial conditions indicate the system starts with the three first RTRPs already configured and the `Control Device` in the Ready state.



$r[k] = \perp /$
$m[k+1] \leftarrow 0,$
$c_v[k+1] \leftarrow c_v[k]$
$r[k] \neq \perp /$
$m[k+1] \leftarrow \mathbf{m},$
$c_v[k+1] \leftarrow h(c_v[k], r[k])$
Ready
$(m[k] = 0)$
Reconf
$(m[k] > 0)$
$m[k] = 1/$
$m[k+1] \leftarrow 0,$
$c_v[k+1] \leftarrow c_v[k]$
$m[k] > 1/$
$m[k+1] \leftarrow m[k] - 1,$
$c_v[k+1] \leftarrow c_v[k]$

Figure 6: State chart ruling the behavior of the `Control Device`*.*

4 RTR Modeling with SY

An strategy and comparison of frameworks supporting formal-based development and models of computation is presented by Horita *et al.* [5]. Based on their result, we opt here for the use of ForSyDe [6] to model our TMR system. As ForSyDe is implemented in Haskell, a functional language, implementing (4) to (15) is considered an easy task, and we one does not need to worry about side effects either. Another advantage of functional languages is that functions can be used as normal data, allowing the exchange of control events such as $(\mathbf{f},\mathbf{g},\mathbf{m})$.

The ForSyDe SY library possesses a collection of process constructors, as well as delays, to implement all the processes presented so far. We use the process constructors comb*n*SY, with n indicating the number of inputs, unzip*m*SY, with m indicating the number of outputs, and `delaySY`.

Listing 1 shows the ForSyDe implementation of an RTRP process, where `rtrpFunc` is a Haskell function that implements (4) to (7). For this implementation, we consider an architecture with 5 RTRPs (three initially operating RTRPs and two spare ones). As mentioned in Section 3.1, the initial values $f[0]$, $g[0]$, $m[0]$ and $x[0]$ must be provided and are represented as `f0`, `g0`, `m0` and `x0`.

Listing 1: `RTRP` *process implemented in ForSyDe.*

```
rtrp (f0,g0,m0,x0) ct s_in x' = out
    where (out, fb) = unzipSY $ comb4SY
        rtrpFunc ct s_in x' fb'
          fb' = delaySY (f0,g0,m0,x0) fb
```

In a similar way, Listing 2 shows the ForSyDe implementation of the `Voter` process, where `voterFunc` is a Haskell function that implements (10) to (11).

Listing 2: `Voter` *process implemented in ForSyDe.*

```
voter cv s1 s2 s3 s4 s5 = unzip3SY $ comb2SY
    voterFunc cv (zip5SY s1 s2 s3 s4 s5)
```

Listing 3 shows the ForSyDe implementation of the `Control Device` process, where `ctrlDevLogic` is a Haskell function that implements (12) to (15), and `prosopon1` is a Haskell implementation of a 3-tuple $(\mathbf{f},\mathbf{g},\mathbf{m})$.

Listing 3: `Control Device` process implemented in ForSyDe.

```
ctrlDev r = (cv, cts)
    where (cv, m, ct) = unzip3SY $ comb3SY (
        ctrlDevLogic prosopon1) r m' cv'
          cts = unzip5SY ct
          m' = delaySY 0 m
          cv' = delaySY (1,2,3) cv
```

Finally, Listing 4 shows the TMR process network from Figure 2 implemented in ForSyDe.

Listing 4: TMR process network implemented in ForSyDe.

```
tmrPN s_in = (r, s_out, out2, out4)
    where out1 = rtrp1 ct1' s_in x'
          out2 = rtrp2 ct2' s_in x'
          out3 = rtrp3 ct3' s_in x'
          out4 = rtrp4 ct4' s_in x'
          out5 = rtrp5 ct5' s_in x'
          (s_out, x, r) = voter cv' out1 out2
              out3 out4 out5
          x' = delaySY (Prst 0) x
          (cv, (ct1,ct2,ct3,ct4,ct5)) =
              ctrlDev r
          cv' = delaySY (1,2,3) cv
          ct1' = delaySY Abst ct1
          ct2' = delaySY Abst ct2
          ct3' = delaySY Abst ct3
          ct4' = delaySY Abst ct4
          ct5' = delaySY Abst ct5
```

4.1 Simulation Results

To simulate the TMR architecture, we first need to define the functionalities of each RTRP. The first three RTRPs are implemented to behave as accumulators, *i.e.* each input is added to the result of the previous execution. Functions $\mathbf{f}$ and $\mathbf{g}$, from (16) and (17), are used to implement such accumulator. To simulate a failure in one of these three RTRPs (in this case, we chose to be RTRP_2) we implemented a faulted accumulator, replacing $\mathbf{f}$ for $\tilde{\mathbf{f}}$ given by (18), in which when the result of the previous execution is 3, instead of adding the input to it, it will subtract. RTRPs 4 and 5 are implemented using $\bar{\mathbf{f}}$ given by (19). We assume that it takes 2 clock cycles to reconfigure a new RTRP, *i.e.* $\mathbf{m} = 2$.

$$\mathbf{f}(x,u) = x + u \tag{16}$$

$$\mathbf{g}(x,u) = x \tag{17}$$

$$\tilde{\mathbf{f}}(x,u) = \begin{cases} x - u & \text{if } x = 3 \\ x + u & \text{otherwise} \end{cases} \tag{18}$$

$$\bar{\mathbf{f}}(x,u) = x - u \tag{19}$$

Table 1 shows the simulation results considering a constant input stream of ones. When $k = 4$, RTRP_2 outputs the wrong result, as seen in y_2. At the same instant, the `Voter` detects such fault and signals the `Control Device` that an error occurred in RTRP_2 and, therefore, it needs to be replaced. Then, the `Control Device` starts the reconfiguration procedure for RTRP_4, which takes 2 cycles to complete. When $k = 7$, RTRP_4 is fully reconfigured and matches the RTRP_1 and RTRP_3 outputs. As we can see, the output given by s_{out} is not affected by the fault in RTRP_2, nor the reconfiguration of RTRP_4. We can also notice that $r = 4$ when $k = 5$ and $k = 6$, indicating that the `Voter` is signaling the `Control Device` about an error in the output of RTRP_4. As RTRP_4 is being reconfigured in this interval, the `Control Device` is in the Reconf state and, therefore, is ignoring the values arriving through r.

Table 1: Simulation results in ForSyDe

k	0	1	2	3	4	5	6	7	8	9
s_{in}	1	1	1	1	1	1	1	1	1	1
s_{out}	0	1	2	3	4	5	6	7	8	9
y_1	0	1	2	3	4	5	6	7	8	9
y_2	0	1	2	3	2	3	2	3	2	3
y_3	0	1	2	3	4	5	6	7	8	9
y_4	0	-1	-2	-3	-4	$\perp$	$\perp$	7	8	9
y_5	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
r	$\perp$	$\perp$	$\perp$	$\perp$	2	4	4	$\perp$	$\perp$	$\perp$

5 Related Work

The idea of using triple modular redundancy with runtime reconfiguration is not new. SRAM field programmable gate arrays (FPGAs) must protect its configuration memory from SEUs, and TMR techniques are applied to such devices. However, when a majority voter is fed with two wrong answers, possibly caused by multiple independent SEUs, it produces the wrong result. One way to solve this issue is to periodically write back the whole bistream of each module, which is time consuming and leaves the modules inactive during this period. [7] proposes an optimization of the reconfiguration time in order to cope with this problem.

Another application of TMR using RTR is presented by [8], where an adaptive reconfigurable voting mechanism whose main goal is to extend the dynamic and partial reconfiguration SEU mitigation to the voter, which is usually the single point of failure in TMR architectures.

A novel technique for synchronizing the states of a newly reconfigured module is presented in [9]. Such technique consists on predicting the future state to which the system will soon converge (check point state) and presetting the reconfigured module to it. Therefore, only the reconfigured module will be set on hold until the check-point state is reached.

The research introduced in [10] claims an improvement of fault resilience, on up to 80%, by composing and applying space and time redundancy, *i.e.* multiprocessors and scheduling, with task migration among processors in hard real-time systems design. That architecture follows the multiple instruction, multiple data (MIMD) taxonomy, as proposed by [11].

6 Conclusion

This paper introduced a high level abstraction architecture for safety-critical systems with runtime reconfiguration (RTR) using the triple modular redundancy and the synchronous (SY) model of computation (MoC).

The triple modular redundancy was chosen to be the mechanism for detecting and masking faults. While the triple modular redundancy is a classic way to implement fail mitigation in safety-critical systems, in the event of a permanent fault, the system can mask such fault. However it gets vulnerable to a second fault.

A triple modular redundancy using RTR provides a way for the system to circumvent failures in the presence of multiple permanent faults, provided that no “two faults” happen in a time interval defined by the reconfiguration time of a new module.

We implemented the proposed high level architecture model in the framework ForSyDe and verified that a new `RTRP` can be correctly reconfigured in **m** cycles and can have its states synchronized with the other two `RTRP`s.

Acknowledgments

This research work is supported by the Research Development Foundation (*Fundação de Desenvolvimento da Pesquisa*) - FUNDEP/MCTIC/MDIC

References

[1] Pierre Bieber, Frédéric Boniol, Marc Boyer, Eric Noulard, and Claire Pagetti. New Challenges for Future Avionic Architectures. *AerospaceLab*, (4):p. 1–10, May 2012.

[2] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[3] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, Dec 1998.

[4] Denis S. Loubach. A runtime reconfiguration design targeting avionics systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–8, Sacramento, USA, September 2016. IEEE.

[5] Augusto Y. Horita, Ricardo Bonna, and Denis S. Loubach. Analysis and comparison of frameworks supporting formal system development based on models of computation. In Shahram Latifi, editor, *16th International Conference on Information Technology-New Generations (ITNG 2019). Advances in Intelligent Systems and Computing, vol 800*, pages 161–167, Cham, 2019. Springer International Publishing.

[6] Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. ForSyDe: System design using a functional language and models of computation. In Soonhoi Ha and Jürgen Teich, editors, *Handbook of Hardware/Software Codesign*, pages 99–140. Springer Netherlands, 2017.

[7] L. Sterpone and A. Ullah. On the optimal reconfiguration times for tmr circuits on sram based fpgas. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, pages 9–14, June 2013.

[8] F. Veljković, T. Riesgo, and E. de la Torre. Adaptive reconfigurable voting for enhanced reliability in medium-grained fault tolerant architectures. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2015)*, pages 1–8, June 2015.

[9] Conrado Pilotto, José Rodrigo Azambuja, and Fernanda Lima Kastensmidt. Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. In *Proceedings of the 21st Annual Symposium on Integrated Circuits and System Design*, SBCCI ’08, pages 199–204, New York, NY, USA, 2008. ACM.

[10] D. S. Loubach and A. M. da Cunha. Avionics hard real-time systems’ concerning fault tolerance. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6A2–1–6A2–18, Oct 2012.

[11] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972.