Specification and Verification of the Triple-Modular Redundancy Fault Tolerant System using CSP

Lanfang Tan, Qingping Tan, Jianli Li
School of Computer
National University of Defense Technology
Changsha, China
tlf1022@126.com, eric.tan.6508@gmail.com, ljl_003@163.com

***Abstract*—A case study on the application of Communicating Sequential Processes (CSP) to the specification and verification of fault tolerant systems is presented. The Triple-Modular Redundancy (TMR) mechanism is a classical design technique for tolerating hardware errors. By specifying the behavior of the faultless module as a CSP process, the behavior of TMR system suffering from hardware errors can be verified as a refinement of the one of the faultless module.**

***Index Terms*—TMR System; fault tolerance; verification; CSP**

I. INTRODUCTION

Fault tolerance is generally accomplished by using redundancy in hardware, software or combination. There are three basic types of redundancy in hardware and software: spatial redundancy, time redundancy, and hybrid. TMR scheme has been one of the most popular fault tolerant mechanisms using spatial redundancy [1]. In TMR systems, computions are replicated into three modules running in parallel and their outputs are voted using a voter circuit. A single fault in any of the redundant modules will not produce an error at the output as the voter will select the correct result from the two working modules and recover the fault. There are numerous examples of dependable systems using the TMR technique [2].

Though the principle of TMR fault tolerant system is straightforward, evaluating system's behavior in the presence of faults constitutes another significant problem, especially for the complicated systems[8][9][10]. In other words, it was not possible to determine whether the behavior described in these requirements would provide the desired level of fault tolerance. Fault injection techniques have emerged as an important method for evaluating fault tolerant systems. However, they cannot cover all fault scenarios. Therefore, they cannot guarantee that all fault consequence has been investigated. This motivates us to explore a formal verification approach that targets a complete validation.

In [3], temporal logic of actions (TLA) [4] is used to specify and validate TMR fault tolerant system. The programs running on three processors are represented as transition systems. Physical faults in the system are modelled as a set of "fault actions" which perform state transformations in the same way as the other program actions. Assuming that errors will not occur on two modules synchronously, the fault tolerance property of TMR system can be verified as the refinement of a non-fault-tolerant program.

In this paper, we propose an approach for the formal verification of TMR fault tolerant systems using CSP [5], which is a member of a class of formal methods known as process algebras. By specifying the property of a faultless module with a CSP process, we prove that TMR fault tolerance system can still satisfy the property in spite of hardware errors. The verification process can be absolutely automatic based on model checking support tool FDR2 [6] of CSP.

The rest of this article is organised as follows. The next section briefly introduces the language of CSP used in this paper; Section III considers the specification for a faultless module; Section IV describes the TMR fault tolerance system suffering from hardware errors; Section V verifies the effectiveness of TMR mechanisms and discusses the use of model checking tool FDR as an automated means of verifying the fault-tolerant design; Section VI concludes with some remarks on the use of CSP in formal verification of fault tolerant systems.

II. THE LANGUAGE OF CSP

CSP is a language where processes proceed from one state to another by engaging in (instantaneous) events [5]. A process is a component that encapsulates some data structures and algorithms for manipulating that data. It interacts with enviroment through synchronised message passing along channels, or events. The set of all events in the interface of a process P, written α.P, is called its alphabet. However, the interface events are not as autonomous actions under the control of a single process but intended as synchronization between the participating processes.

The language of CSP used in this paper is described in Figure 1, which is defined by the following pseudo Backus-Naur form definitions. In Figure 1, c denotes an event, A is a set of events and b is a Boolean expression. The Skip is the process that does nothing but terminates successfully. The prefixprocess c -> P is ready to perform event c and waits until the environment prepares well event c. Once the event c is performed, the subsequent behaviour of c -> P will be that of process P. In the

 ISBN: 978-1-61208-149-6

sequential composition P; Q, the combined process firstly behaves as P and then Q becomes active immediately after the termination of P. The internal choice P ⊓ Q waits to perform events that either P or Q is ready to engage in. Once an event of a component is performed, the subsequent behaviour is given by this component. Selecting either P or Q depends on its internal environment.

The parallel combination P |A| Q only synchronises on events in A, and interleaves on all other events. The hiding operator P \ A makes a given set of events in A internal, thus beyond the control of its environment. The prefix choice a: A -> Pa is ready to perform any event from set A until one is chosen. Pa is its subsequent behaviour, which is dependent on the chosen event a. A process can be defined to allow the input on channel *in* of any item *x* in a set *M*, and the value *x* determines the subsequent behavior [5]:

$$in?x{:}M \rightarrow Q(x) \cong a{:}in.M \rightarrow \mathrm{Pa}$$

where the set $in.M=\{in.m \mid m \in M\}$ and $Pin.m=Q(m)$ for every $m \in M$. The atomic synchronization events here are of the form *in.m*. The output prefix has the form $out!x \rightarrow P$ and this is simply a shorthand for $out!x \rightarrow P$.

The indefinite loop process P* repeats the actions of P after the successful termination of P. The condition operator if b then P else Q fi selects either P or Q according to the Boolean expression b.

P := c -> P | P ; P | P ⊓ P |

P |A| P | P \ A | a: A -> P_a |

P* | if b then P else P □

Figure 1. The CSP operators

III. SPECIFICATION OF A FAULTLESS MODULE

In this section, we want to model the faultless module as a CSP process that represents the general computing model. As Figure 2 illustrates, a faultless module can be abstracted as a computation process, which consists of a processor and a memory. Assume that the program to be performed in the module is deterministic and sequential, consisting of data segment and text segment. During program executing, the processor either executes an instruction in text segment to change the content of data segment, or issues write operation to the memory.

Therefore, the processor can be abstracted as a function $d=funp(l)$, where *d* denotes the content of data segment and *l* denotes the next instruction to be executed. For each input program, the mapping function *funp* is determined. The write operations can be performed just as certain instructions are executed, such as store instructions. So we specify the behaviour that the processor needs write data *d* to the memory by an assertion NeedWrite(*d*), whose value is true if and only if the store instruction is performed. When the processor issues write operations defined as Output (*d*), the memory updates data segment, which is represented by a function Update(*d*).

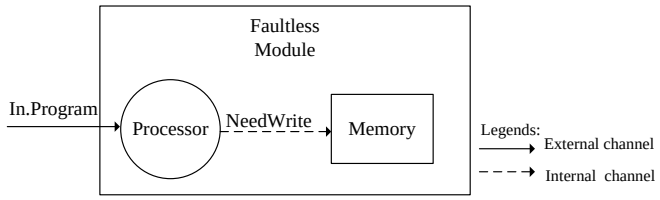

Figure 2. The faultless module

As is mentioned above, we can illustrate the behaviour of the faultless module in the CSP language in Table I.

TABLE I SPECIFICATION FOR THE FAULTLESS MODULE

1	Faultless Module: = Processor | { Output(*d*) } | Memory \ { Output(*d*) }
2	Processor: =*in*? Program → (if (Exited(*l*)) then Skip else (if (NeedWrite(*d*)) then *out*! Output (*d*) else *funp* (*l*) fi.)fi.)* →Processor
3	Memory: = (*in*? Output(*d*) → Update (*d*)) → Memory

The faultless module is encoded as a parallel combination construction, with two processes synchronizing on the event Output (*d*). The CSP process is defined by the expression on the right-hand-side of the definition ":=" symbol. Processor specifies the module initially inputs a program through channel *in* and then executes in terms of the input program. During executing, the Processor either exits the program and prepares for the next input program, or continues executing depending on the Boolean expression Exited(*l*). When Processor output *d* through channel in, Memory updates the values of the data segment and then returns to its initial state preparing for the next write operation.

IV. SPECIFICATION OF THE TMR SYSTEM

The TMR system allows parallel execution of the three modules on three processors thereby providing tolerance of certain permanent and transient hardware faults. Suppose that the hardware faults only occur on processors and memories are protected by error correcting codes (ECC) mechanism [7].

The principle of the TMR system is shown in Figure 3. It works as follows. Once a program is input, the three processors start executing. When a processor needs writing data in memory, it issues a signal "Needwrite" to the Voter. When all signals to write memory arrive, the voter chooses the correct data to be written into memory and then sends the answer message to the three processors.

We may specify the behavior of the Voter by the following CSP process in Table II. It is a sequential composition, where the process first waits the signals to write data and then choose the correct data. The notations of answer message are defined as follows:

1) Ack !"0": denotes the data to be written into memory if three processors are equal.
2) Ack !"i"(i=1, 2,3): denotes the data to be written of $processor_i$ is not equal to the data of the other two

processors.

3) Ack!"4": denotes the data to be written of three processors are not equal to each other. Of course, this case is not possible unless the Voter works abnormally.

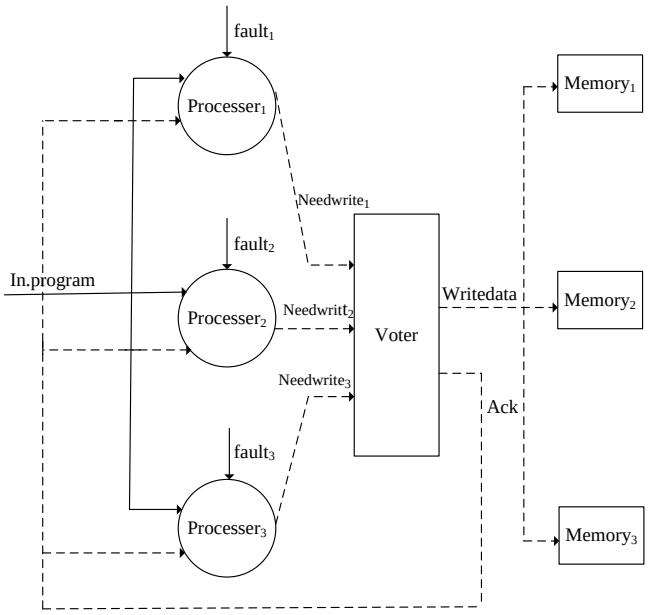

Figure 3. The TMR system

TABLE II SPECIFICATION FOR THE VOTER OF THE TMR SYSTEM

	Voter: = (*in*? Needwrite$_1$ → SKIP ‖*in*? Needwrite$_2$ → Skip ‖
1	*in*?Needwrite$_3$ → SKIP);
2	If (d1==d2 and d2==d3) then (Writedata!(d1) ‖ Ack ! 0)
3	else if (d1==d2 and d2!=d3) then (Writedata!(d2) ‖ Ack ! 3)
4	else if (d1==d3 and d2!=d3) then (Writedata!(d1) ‖ Ack ! 2)
5	else if d3==d2 and d1!=d3) then (Writedata!(d3) ‖ Ack ! 1)
6	else (Ack !4)
7	→ Voter

Similarly, the behavior of the Modules can be expressed as Processor$_i$ and Memory$_i$ in CSP language, which is shown in Table III. The Process$_i$ is designed as a sequential composition, which begins with an internal choice waiting to perform events. If a fault occurs, the data values will be corrupt, which is expressed by a function Wrong (d). We can speculate that once a fault occurs, it will be responded by the answer message and then the data will be recovered, which is described as Recover(d). Memory$_i$ process is similar to the Memory process of the faultless module.

TABLE III SPECIFICATION FOR THE PROCESSOR OF THE TMR SYSTEM

1	Processor$_i$:= (*in*?fault$_i$ → (Wrong$_i$ (*d*) → SKIP)
	⊓ (*in*? Program →
	if (Exited(*l*)) then Skip else
2	(if (NeedWrite$_i$) then *out*! Output$_i$(*d*)
3	else *fun*$_p$(*l*) fi.)*→ SKIP) ⊓
4	(Ack?x →(if *in*?Ack.x= i then Recover$_i$(*d*)
5	else SKIP))
6	→Processor$_i$ (i=1,2,3)
7	Memory$_i$: = (In? Output(*d*) → Update (*d*)) → Memory$_i$ (i=1,2,3)

As mentioned above, the TMR system can be illustrated in Table IV. It is designed as a parallel combination construction containing two processes. The first process is that three processors and Voter synchronize on events "Output" and "Ack". The second process describes the synchronizing between the first process and the three memories on event "Writedata". All the events are internal to the TMR system, thus the hiding operator is adopted.

TABLE IV SPECIFICATION FOR THE TMR SYSTEM

1	TMR System := (Processor$_1$ ‖ Processor$_2$ ‖ Processor$_3$
2	|{ Output$_1$, Output$_2$, Output$_3$, Ack } | Voter)
3	|{Writedata}|
4	(Memory$_1$ ‖ Memory$_2$‖ Memory$_3$) \
5	\{ Output$_1$, Output$_2$, Output$_3$,WriteData, Ack }

V. VERIFICATION OF FAULT TOLERANCE

The verification of fault tolerance for the TMR system amounts to showing that the behaviour of the TMR system suffering from hardware faults is a refinement of the behavior of a faultless module, as stated in the following lemma:

Lemma 1: Faultless Module ≦ TMR System

Proof: One straightforward way to show the refinement relationship is to apply semantics preserving transformation upon the process definition based on the algebraic rules associated with CSP operators [5]. This approach explores or enumerates manually all possible states of a process defined by parallel combination. Also, since here we target at the behavioral properties rather than the functional property of the process, an abstract version of TMR system which ignores the functional aspect including values, variables, and Boolean expressions can be obtained. Table V shows the proof for Lemma1.

TABLE V THE PROOF OF THE TMR SYSTEM

1	TMR system= (Process$_i$|{ NeedWrite$_i$, Ack } | Voter) |{WriteData}| Memory$_i$\{ NeedWrite$_i$, Ack,WriteData } i=1,2,3 (1)
2	Fill the definitions of Processor$_i$,Memory$_i$ and Voter in (1)
3	Apply the following algebraic laws (2) to simply (1)
4	P ‖ (R; Q) = (P ‖ R); (P‖ Q) (2)
5	Apply the following algebraic laws (2) and (3) to simply (1)
6	(P ⊓ Q)‖ R= (P ‖ R) ⊓ (Q‖ R) (3)
	Assume that two faults cannot occur synchronously, (1) can be simplified as (4)
8	Faultless Module$_{i=1,2}$; Ack?x->SKIP‖ (fault$_i$? → (Wrong (*d*) →SKIP)‖ Voter ; (4)
9	Assume that the Recover(*d*) can restore the program effectively, apply the following laws to simply(4)
10	SKIP ; P=P (a→P);Q=a→(P;Q)
11	(4) can be simplified as following :
12	Faultless Module$_{i=1,2,3}$

However, it is too laborious to verify the property of the TMR system manually. Fortunately, the FDR model checking tool [6] can be used to verify the above lemma automatically. To verify whether the TMR system suffering from hardware fault is a refinement of the faultless module, the failures-divergence model [6] in which the possible behaviours of a process are denoted by a set called its failures-divergence is adopted. A process Q is a

refinement of another process P, written as P [FD= Q, if (and only if) the failures-divergence set of the former is contained in the failures-divergence set of the latter. Firstly, the system is defined as a CSP process. Then, it is translated into the input language of FDR. A listing of the FDR2-compatible source can be found in the appendix. While a detailed introduction to the semantics of FDR is beyond the scope of this paper, the appendix specifies the TMR system obtained by translating the CSP process into the input language of FDR.

VI. CONCLUSION AND FUTURE WORK

This paper has shown how the FDR refinement checker for CSP can be used to specify and verify the fault tolerant system. Firstly, a faultless module that is abstracted as a computation model is encoded as a CSP process. Then the TMR system suffering from hardware errors is illustrated in CSP. By verifying the TMR system is a refinement of a faultless module, the fault tolerance property of the TMR system can be verified. Moreover, by the model checking tool FDR, the verification is performed automatically.

The work with more similarity to ours is described in [3], but ours is much more general and practical. Our work just needs to encode the system in CSP processes, and then the verification can be performed automatically by FDR. FDR searches the state space of the system until it either finds an undetected error or exhausts the state space. This search is automatic in the sense that it does not require user guidance once the system has been modeled in CSP. On the contrary, the work in [3] validates the correctness of fault tolerant systems using axioms and proof rules. It can only be used by experts who are well-drained in logical reasoning and have considerable experience. Thus it is not practical.

However, we only focus on the write operation between the processor and the memory. In order to simplify the process description, the read operation that the processor reads data from the memory is not considered. In general, it is just an early work. In the future, we will investigate the read operation and apply the method to some complicated system.

APPENDIX

{- The idea of this script is to prove that the TMR system is a refinement of the behavior of a faultless module -}
-- Event definitions
Output (d) =yes|no
Tags_ack= {0, 1, 2, 3}
NeedWrite= $NeedWrite_1$| $NeedWrite_2$| $NeedWrite_3$
Data=d_1|d_2|d_3
--channel declarations
Channel Processor_in, $Proceesor\_in_1$, $Proceesor\_in_2$, $Proceesor\_in_3$
channel Memory_in, $Memory\_in_1$, $Memory\_in_2$, $Memory\_in_3$: Output
channel Voter_ack : Tags_ack
channel Voter_in: NeedWrite
channel Voter_WriteData
-- The specification is for the processor of the faultless module
Processor=Processor_in? Program-> (if (NeedWrite) out! Output (d).yes else funp(l) fi.) ->Processor
-- The specification is for the memory of the faultless module
Memory = (Memory_in? Output(d).yes > Update (d)) -> Memory
-- The specification is the faultless module
Faultless Module= (Processor [| {| Output(d)|} |] Memory) \ {| Output(d) |}
-- The specification is the Voter of the faultless module
Voter: = (Voter_in? $Needwrite_1$ --> SKIP ‖ Voter_in? $Needwrite_2$ --> SKIP ‖ Voter_in?$Needwrite_3$ -> SKIP);
if (d_1==d_2 and d_2==d_3) then (Voter_WriteData!d_1 ‖ Voter_ack ! 0)
else if (d_1==d_2 and d_2! =d_3) then (Writedata! d2 ‖ Voter_ack! 3)
else if (d_1==d_3 and d_2! =d_3) then (Writedata! d1 ‖ Voter_ack! 2)
else d_3==d_2 and d_1!=d_3) then (Writedata!d_3 ‖ Voter_ack ! 1)
--> Voter
-- The specification is the Processor1 of the faultless module, it is similar to Processor2 and Processor3
$Processor_i$ = ((in?$fault_i$ -> (Wrong (d_i) -> SKIP) [] In? Program-> (if ($NeedWrite_i$) Out! $Output_i$(d_i)
else funp(l) fi. --> SKIP);
Ack? ->(if in?ack.x= i then $Recover_i$(d)
else SKIP);
$Processor_i$ (i=1,2,3)
$Memory_i$= (In? Output(d) -> Update (d)) -> $Memory_i$ (i=1,2,3)
-- Finally we put it all together, and hide internal communication
TMR System = ($Processor_1$ ‖ $Processor_2$‖ $Processor_3$ |{$Output_1$, $Output_2$, $Output_3$,Ack}| Voter) |{Writedata}|($Memory_1$‖$Memory_2$ ‖ $Memory_3$)\{ $Output_1$, $Output_2$, $Output_3$,Ack, Writedata}
--The Specification of Faultless Module ≦ TMR System
assert Faultless Module [FD= TMR System

REFERENCES

[1] Kang G. S. and Hagbae K., "A Time Redundancy Approach to TMR Failures Using Fault-State Likelihoods," IEEE Trans. on Computers, vol. 43, pp. 1151-1162, Oct. 1994.

[2] Siewiorek D. P. and R. S. Swarz, "Reliable Computer Systems: Design and Evaluation," Digital Press, 1992.

[3] Liu Z.M. and Joseph M., "Specification and verification of fault tolerance, timing and scheduling," ACM Trans. on Programming language and systems, vol. 21, pp. 46-49, Jun. 1999.

[4] Lamport L, "The temporal logic of actions," ACM Trans. on Programming language and systems, vol. 16, pp. 872-923, Nov. 1994.

[5] Hoare C.A.R, "Communicating Sequential Processes," Prentice Hall, 1985.

[6] Formal Systems (Europe) Ltd, FDR2 User Manual, 2005.

[7] Lin S. and Costello D. J., "Error Control Coding: Fundamentals and Applications", second edition, Prentice Hall: Englewood Cliffs, 2004.

[8] Subhasish M. and Edward J. M.," Word-Voter: A New Voter Design for triple Modular Redundant systems", Proc. IEEE Symp. VLSI Test, IEEE Press, pp. 465-470, Aug. 2000.

[9] Kang G. Shin and Hagbae Kim, "A Time Redundancy Approach to TMR Failures Using Fault-State Likelihoods", IEEE Trans. on Computers, vol. 43, pp. 1151-1162, Oct. 1994.

[10] Lisboa C. A. L., Schuler E. and Carro L., "Going beyond TMR for Protection against Multiple Faults", IEEE Symp. Integrated Circuits and Systems Design, IEEE Press, pp. 80-85, Sept. 2005.

 ISBN: 978-1-61208-149-6