Fault-Tolerant CSP

Shyan-Ming Yuan and Chin-Juan Chen

Department of Computer Science, National Chiao Tung University
Hsinchu, Taiwan, R. O. C. 30050

Abstract

In a network of communicating processes performing a distributed computation, we can replicate some or all of the communicating processes on different nodes to increase successful probability of the distributed computation against node failures. In this paper we use CSP to express this scheme by appropriately translating the commands which communicate with replicated processes. In order to make the translation scheme simple and easy to implement, the order of the replicas are deterministic.

1 Introduction

Consider a distributed computation being performed by a network of communicating processes. If even one of the nodes within the network fails the distributed computation will not be performed. The reliability can be increased by replicating all or some of the communicating processes on different nodes. As long as at least one copy of each communicating process is still active, the entire distributed computation will be performed.

Each copy of a communicating process is called an incarnation which can perform the task of the process alone. A process is k-resilient if it has (k + 1) incarnations so that it has the capability to resist up to k node failures. In this paper we use CSP[1] to express our scheme and focus on how to appropriately translate the CSP commands in one process which communicates with many incarnations.

Fault tolerance against node failures has been extensively studied such as atomic actions [2, 3, 4] and resilient data objects [5, 6, 7]. Most of these schemes employ "passive" backups, one of which "takes over" when the "primary" fails. Such schemes require some form of checkpointing and roll-back strategies. Our scheme employs "active" backups, by keeping all incarnations of processes simultaneously active on different nodes. Since there is no primary and secondary, check-pointing and/or roll-back strategies are not necessary and no domino effects [8] will occur.

Note that the major difference between our translation scheme[1] and Jalote's scheme[9] is that we use deterministic order for replicated incarnations. So, our scheme is much easy to implement.

2 Review CSP

We briefly review the pertinent concepts of CSP. A parallel command is of the form:

```
program_name ::= \{P_1||P_2||\ldots\}
```

which specifies concurrent execution of its constituent sequential processes (P_1, P_2, \ldots). CSP processes may only communicate with each other using the input and output commands. An output command is of the form:

```
destination!expression
```

where destination is the process name and expression is a simple or structured value. An input command has the form:

```
source?target
```

where source is a process name and target is a simple or structured variable. Communication between two processes of a parallel command occurs whenever the input command of one process and the output command of the other correspond. Commands which correspond are executed simultaneously, and their combined effect is to assign the value of the expression of the output command to the target variable of the input command(expression and target must have the same type). The i/o command is delayed until the other process is ready with the corresponding output or input command.

A guarded command has the form:

```
G \rightarrow C
```

if guard G is true then command list C will be executed. If an i/o command appears in a guard, it is called an i/o guard[10, 11]. We consider the failure of the i/o command in a guard as a failure of the guard, which merely implies that the guarded command is not executed.

An alternative command has the form:

```
\{G_1 \rightarrow C_1 ; G_2 \rightarrow C_2 ; \ldots ; G_n \rightarrow C_n\}
```

The execution of an alternative command selects exactly one of the constituent guarded commands to execute. In the case when more than one of the guards are true, a command is selected non-deterministically from those commands with successful guards.

A repetitive command has the form:

```
\ast [alternative command]
```

alternative command will be executed repeatedly until all guards are failed.
We are interested in node (or process) failures. A failure of an i/o command is considered to be either the failure of the source or destination processes or termination of these processes. The predicate failed(P) is defined as:
\[
\text{failed}(P) \equiv \text{"process P has failed".}
\]
The failure of an i/o command is considered as an exception condition and is denoted as
\[
C[\text{fail} \rightarrow EH]
\]
If the simple i/o command \(C\) fails, then the CSP commands in \(EH\) (exception handler) is executed.

3 Translation scheme for process with incarnations

3.1 Translation scheme for process with two incarnations

If process \(P\) has incarnations called \(P_1\) and \(P_2\) which are executed on different nodes. The process \(P\) can be expressed as \(P := [P_1][P_2]\). Due to different incarnations of \(P\), the original program in process \(Q\) which designates \(P\) as the source or target have to be appropriately translated. There are two cases:

3.1.1 A simple i/o command

We assume that a process contains a boolean array called "FAILED" which is indexed by the process name (we use "[]" for the index operation). The purpose of the array is to record process failures, and is initialized to be false. The command of the form \(P?z\) in process \(Q\) should be translated as figure 1.

\[
\begin{align*}
\text{FAZLED}(P_1) & \rightarrow \text{skip} \\
\lnot\text{FAZLED}(P_1) & \rightarrow \\
\text{PI?z}[\text{fail} \rightarrow \text{FAILED}(P_1) := \text{true}]; \\
\text{FAILED}(P_2) & \rightarrow \text{skip} \\
\lnot\text{FAILED}(P_2) & \rightarrow \\
\text{Pz?z}[\text{fail} \rightarrow \text{FAILED}(P_2) := \text{true}]; \\
\text{FAZLED}(P_2) & := \text{true}
\end{align*}
\]

Fig. 1: Translation of the command \(P?z\) in figure 1 is successful. For the future input/output commands of \(Q\) communicating with \(P_1\) will not be invoked because FAILED(\(P_1\)) in the first alternative command of figure 1 is true.

3.1.2 An i/o command in a guard

Now let us consider a guarded command in a process \(Q\) which has an input or output guard as the form of \(P?z \rightarrow C\) or \(P!z \rightarrow C\). The translations are shown in figure 3.

\[
\text{true} \rightarrow \\
\begin{align*}
\lnot\text{FAILED}(P_1) & \rightarrow \text{skip} \\
\lnot\lnot\text{FAILED}(P_1) & \rightarrow \\
P_1?z[\text{fail} \rightarrow \text{FAILED}(P_1) := \text{true}]; \\
\text{FAILED}(P_2) & \rightarrow \text{skip} \\
\lnot\text{FAILED}(P_2) & \rightarrow \\
P_2?z[\text{fail} \rightarrow \text{FAILED}(P_2) := \text{true}]; \\
\lnot\text{FAILED}(P_2) & \rightarrow \text{C}.
\end{align*}
\]

Fig. 3: Translation of the command \(P?z \rightarrow C\)

If there are multiple conditions in the guard, they can be anded with \(\land\). The translations of the guarded command with output guard can be derived in the similar manner.

3.2 Generalized scheme

Now we generalize the scheme described in section 3.1 for processes with \(k\) incarnations. If process \(P\) has \(k\) incarnations named \(P_1, P_2, \ldots, P_k\), then there are the following two cases to be considered.

3.2.1 A simple i/o command

The translation of the command of the form \(P?a\) in a process \(Q\) is shown in figure 4.

\[
\begin{align*}
\text{FAZLED}(P_1) & \rightarrow \text{skip} \\
\lnot\text{FAZLED}(P_1) & \rightarrow \\
P_1?a[\text{fail} \rightarrow \text{FAILED}(P_1) := \text{true}]; \\
\text{FAILED}(P_2) & \rightarrow \text{skip} \\
\lnot\text{FAILED}(P_2) & \rightarrow \\
P_2?a[\text{fail} \rightarrow \text{FAILED}(P_2) := \text{true}]; \\
\text{FAZLED}(P_2) & := \text{true}
\end{align*}
\]

Fig. 4: Translation scheme of \(P?z\) for process \(P\) with \(k\) incarnations

When \(Q\) performs an input command designates \(P\) as the source, \(Q\) receives the message from both \(P_1\) and \(P_2\). Consequently, the matching output commands in both \(P_1\) and \(P_2\) are satisfied, and both can continue their execution.

Assume that \(P_1\) fails, the first alternative command in figure 1 will invoke exception handler to record the failure information in FAILED(\(P_1\)) by changing its value from false into true. However \(Q\) will receive message from \(P_2\) because the second alternative command

3.2.2 An i/o command in a guard

The command of the form \(P?z \rightarrow C\) is translated into the form shown in figure 5.
true → 
\[
[\text{FAILED}(P_i) \rightarrow \text{skip} \\
\neg \text{FAILED}(P_i) \rightarrow \\
P_i?x\{\text{fail} \rightarrow \text{FAILED}(P_i) := \text{true}\}]
\]

[FAILED(P_n) → \\
P_n?z\{\text{fail} \rightarrow \text{FAILED}(P_n) := \text{true}\}]

... ...

[FAILED(P_k) → \\
P_k?z\{\text{fail} \rightarrow \text{FAILED}(P_k) := \text{true}\}]

... ...

[FAILED(P_a) → \\
P_a?z\{\text{fail} \rightarrow \text{FAILED}(P_a) := \text{true}\}]


LEMMA 4.1 Let post(S) represent the post condition of a statement S. If a command P?x is executed in a process P, then post(P?x) = io(P.Q) \land (z = \text{exp})

post(Q?exp) = io(P.Q)

The condition io(P.Q) states that the message was successfully passed between P and Q. Since a successful execution of an input or output command implies a successful execution of its corresponding output or input command, in considering a set of matching commands the clause io(P.Q) may be dropped from the post condition of either the input or output command, if it is included in the post condition of the other.

LEMMA 4.2 For a guarded command of the form: GC = \( G_1 \rightarrow C_1 \in G_2 \rightarrow C_2 \), post(GC) = (post(G_1) \land post(C_1)) \lor (post(G_2) \land post(C_2))

If \( G_1 \) has an i/o command then post(\( G_1 \)) has the form: boolean_condition \land post(i/o command)

Otherwise, it will simply be a boolean condition.

4. The correctness of our scheme

There are many papers provide techniques against CSP program failures, such as deadlock, design faults in [12, 13, 14]. But we are interested in node failure. So, we make two assumptions:

A1: The CSP program is correct.
A2: Each message passing between two processes is atomic.

We have the following lemmas to prove the correctness of the translation scheme.

LEMMA 4.3 Let C be of the form P?x or P!ezp, and let C' be C with the exception handler. Then post(C') = \( (\neg \text{failed}(P) \land post(C)) \lor (\text{failed}(P) \land post(\text{EH})) \)

LEMMA 4.4 We have the following assertion about the variable FAILED in any process Q.
\( \text{FAILED}(P_i) \Rightarrow \text{failed}(P_i) \)

LEMMA 4.5 For a 1-resilient processes, we have the following constraint:
\( \neg \text{failed}(P_1) \lor \neg \text{failed}(P_2) = \text{True} \)

The constraint states that at least one of the incarnations is always active.

4.1 The correctness of translation scheme for simple input command

Let P be the process to be made fault-tolerant, and \( P_1, \ldots, P_n \) be its incarnations. Let Q be any other process in the system that communicates with P. To ensure fault-tolerance there are two conditions that must be satisfied:

FT1: \( \text{failed}(P_i) \lor (\neg \text{failed}(P_i) \land post(P/P_i)) \), for \( i = 1 \ldots n. \)

FT2: \( (\neg \text{failed}(P_1) \lor \cdots \lor \neg \text{failed}(P_n)) \land \text{post}(Q) \)

Where, \( \text{post}(P/P_i) \) is the post condition of P with \( P_i \) substituted for P. If these two conditions hold for every command in P and Q, then the conditions for fault tolerance will automatically be satisfied. Since our scheme modifies only a few commands, we only need to show the above two conditions hold for the commands modified by our scheme.

Let us now consider a command P?x in Q where P has two incarnations and the corresponding command Q!ezp in P. The command in Q would be translated as shown in Fig. 1, and the commands in P will be executed by both the incarnations \( P_1 \) and \( P_2 \). Because there are only two incarnations, conditions FT1 and FT2 can be rewritten as follows:

\( \neg \text{failed}(P_1) \lor \neg \text{failed}(P_2) \lor \text{failed}(P_1) \land \text{failed}(P_2) \land \text{io}(P_1, Q) \land \text{io}(P_2, Q) \land \text{ Failed}(P_1) \land \text{ Failed}(P_2) \)

In order to show that the translated scheme shown in figure 1 is correct, we must prove that its post condition is as above. So, let's consider the post condition of the translated form of the command P?x in Q. The post condition of the first alternative command of the translation shown in Fig. 1 is
I I
[Image 1x0 to 621x799]

hence the post condition of the translated command is
items, and so

can be shown by using the similar manner
The post condition of the last line in figure
be performed automatically by a precompiler. So
which participate in distributed computation can be
easily implemented at application level.

Similar expression will be obtained in simplifying
the second alternative command. Hence the post
condition of the translated command is
\((\neg \text{failed}(P_1) \land \neg \text{failed}(P_2)) \land z = \text{exp}) \land \text{io}(P_1, Q) \lor \text{io}(P_2, Q))\)

If there are many incarnations with \(P\). The post
conditions of FT1 and FT2 will include more " \&" items, and so as post condition of the translated
commands. The correctness for more than two incarnations can be shown by using the similar manner as above.

4.2 The correctness of the translation for
guarded command with Input Guard

FT1 and FT2 can be rewritten as follows:
\((\neg \text{failed}(P_1) \land \text{io}(P_1, Q))\)
\((\neg \text{failed}(P_2) \land \text{io}(P_2, Q))\)
\(\land \text{io}(P_1, Q) \lor \text{io}(P_2, Q))\)

The post conditions of the first two alternative
commands of the translation shown in figure 3 is the same
as section 4.1.
\((\neg \text{failed}(P_1) \land \text{io}(P_1, Q))\)
\(\land \text{io}(P_2, Q))\)

The correctness for more than two incarnations can be
shown by using the similar manner as above.

5 Conclusion

The translation scheme we have presented can be
performed automatically by a precompiler. So
the fault-tolerant feature of communicating processes
which participate in distributed computation can be


J. E. Alchim and M. S. McKendry, "Synchronization
and recovery of actions," in Proceedings of symp. on principles of distributed computing,

B. H. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust, distributed

D. P. Reed, "Implementing atomic actions on de-
centralized data," ACM Transactions on Com-

P. A. Alsberg and J. D. Day, "A principle for re-
silient sharing of distributed resources," in Proc.

T. R. K. P. Birman, T. A. Joseph and A. E. Ab-
badi, "Implementing fault-tolerant distributed
objects," IEEE Transactions on Software Engi-

L. Svobodova, "Resilient distributed computing,
" IEEE Transactions on Software Engineering,

B. Randell, "System structure for software fault
tolerance," IEEE Transactions on Software En-

P. Jalote, "Fault-tolerant computation in com-
municating sequential processes," CS-TR-1653,
Computer Science Department, University of
Maryland College Park, MD 20742, May 1986.

A. J. Bernstein, "Output guards and nondeter-
minism in communicating sequential processes,
ACM Trans. on Prog. Languages and Systems,

A. Silberschatz, "Communicating and synchro-
nization in distributed systems," IEEE Transac-
tions on Software Engineering, vol. SE-5, no. 6,

F. Cristian, "Exception handling and software fault
tolerance," IEEE Transactions on Computers,

Software Engg., vol. SE-12, no. 1, pp. 59-68,
1986.

J. Misra and K. M. Chandy, "Proofs of networks
of processes," IEEE Trans. on Software Engg.,

References

[1] C. A. R. Hoare, "Communicating sequential pro-
cesses," Communications of the ACM, vol. 21,

and recovery of actions," in Proceedings of symp. on principles of distributed computing,


[4] D. P. Reed, "Implementing atomic actions on de-
centralized data," ACM Transactions on Com-

[5] P. A. Alsberg and J. D. Day, "A principle for re-
silient sharing of distributed resources," in Proc.

badi, "Implementing fault-tolerant distributed
objects," IEEE Transactions on Software Engi-

[7] L. Svobodova, "Resilient distributed computing,
" IEEE Transactions on Software Engineering,

[8] B. Randell, "System structure for software fault
tolerance," IEEE Transactions on Software En-

[9] P. Jalote, "Fault-tolerant computation in com-
municating sequential processes," CS-TR-1653,
Computer Science Department, University of
Maryland College Park, MD 20742, May 1986.

[10] A. J. Bernstein, "Output guards and nondeter-
minism in communicating sequential processes,
ACM Trans. on Prog. Languages and Systems,

[11] A. Silberschatz, "Communicating and synchro-
nization in distributed systems," IEEE Transac-
tions on Software Engineering, vol. SE-5, no. 6,

[12] F. Cristian, "Exception handling and software fault
tolerance," IEEE Transactions on Computers,

Software Engg., vol. SE-12, no. 1, pp. 59-68,
1986.

of processes," IEEE Trans. on Software Engg.,