1 Example: Alternating Bit Protocol

In this section, we present an alternating bit protocol with a sender and a receiver processes that are subject to message loss faults. Using the synthesis method presented in [1], we synthesize an alternating bit protocol that is nonmasking fault-tolerant; i.e., when faults occur the program guarantees recovery to its invariant. However, during recovery, the nonmasking fault-tolerant protocol may temporarily violate it safety specification.

The alternating bit protocol (ABP). The fault-intolerant program consists of two processes: a sender and a receiver. The sender reads from an infinite input stream of data packets and sends the newly read packet to the receiver. The receiver copies each received packet into an infinite output stream. When the sender sends a data packet, it waits for an acknowledgement from the receiver before it sends the next packet. Also, when the receiver receives a new data packet, it sends an acknowledgment bit back to the sender. A one-bit message header suffices to identify the data packet currently being sent since at every moment there exists at most one unacknowledged data packet. Using this identifier bit, the sender (respectively, the receiver) does not need count the total number of packets sent (respectively, received).

Both processes have read/write access to a send channel and a receive channel. The send channel is represented by an integer variable cs and the variable cr models the receive channel. The domain of cs (respectively, cr) is $\{-1, 0, 1\}$, where 0 and 1 represent the value of the data bit in the channel and -1 represents an empty channel. Since we are only concerned about the synchronization between the sender and the receiver, we do not explicitly consider the actual data being sent. Thus, we consider the contents of cs and cr to be a single binary digit. The sender process has a Boolean variable bs that stores the data bit that identifies the data packet currently being sent to the receiver. Correspondingly, the receiver process has a Boolean variable br that represents the value that is supposed to be received. When the sender process transmits a data packet, it waits for a confirmation from the receiver before it sends the next packet. To represent the mode of operation, the sender process uses a Boolean variable rs. The value of rs is 0 iff the sender is waiting for an acknowledgement. Likewise, the receiver process uses a Boolean variable rr such that the value of rr is 0 iff the receiver is waiting for a new packet.

We represent a state s of the ABP program by a 6-tuple $\langle rs, bs, rr, br, cs, cr\rangle$. Thus, if we start from initial state $\langle 1, 1, 0, 0, -1, -1\rangle$, then the sender process begins to send a data bit 1 while the receiver waits to receive it. We represent the transitions of the sender process in the fault-intolerant program ABP by the following actions.

$Send_0: (rs = 1) \longrightarrow rs := 0; cs := bs;$
$Send_1: (cr \neq -1) \longrightarrow rs := 1; cr := -1; bs := (bs + 1) \mod 2;$

The actions of the receiver process in the fault-intolerant program ABP are as follows:

$Rec_0: (cs \neq -1) \longrightarrow cs := -1; rr := 1; br := (br + 1) \mod 2;$
$Rec_1: (rr = 1) \longrightarrow rr := 0; cr := br;$

Faults. Faults can remove a data bit from either one of the communication channels causing the loss of that data bit. Hence, we model faults by setting the value of cs (respectively, cr) to an unknown value -1.

$F_0: (cs \neq -1) \longrightarrow cs := -1;$
$F_1: (cr \neq -1) \longrightarrow cr := -1;$

We assume that the fault actions will be executed a finite number of times; i.e., eventually faults stop occurring.

Safety specification. The problem specification requires that the receiver receives no duplicate packets.

Invariant. The invariant of the ABP program is equal to the state predicate S_{ABP}, where

$$\begin{aligned} S_{ABP} = \{s \mid & (((rr(s) = 1) \vee (cr(s) \neq -1)) \Rightarrow (br(s) = bs(s))) \wedge \\ & (((rs(s) = 1) \vee (cs(s) \neq -1)) \Rightarrow (br(s) \neq bs(s))) \wedge ((cs(s) = -1) \vee (cs(s) = bs(s))) \wedge \\ & (((cs(s) = -1) \wedge (cr(s) = -1)) \Rightarrow ((rr(s) + rs(s)) = 1)) \wedge \\ & (((cs(s) \neq -1) \wedge (cr(s) = -1)) \Rightarrow ((rr(s) + rs(s)) = 0)) \wedge \\ & (((cs(s) = -1) \wedge (cr(s) \neq -1)) \Rightarrow ((rr(s) + rs(s)) = 0)) \} \end{aligned}$$

In the state predicate S_{ABP}, the notation $v(s)$, where v is a variable name, represents the value of v in state s.

Fault-span. The state of the ABP program may be perturbed to the state predicate T_{ABP} due to fault transitions, where

$$\begin{aligned} T_{ABP} = \{s \mid & ((cs(s) = -1) \vee (cs(s) = bs(s))) \wedge \\ & (((cs(s) = -1) \vee (cr(s) = -1)) \Rightarrow (((rr(s) + rs(s)) = 1) \vee ((rr(s) + rs(s)) = 0)))\} \end{aligned}$$

Adding the actions of the high atomicity pseudo process. Using our synthesis method, we have identified the following high atomicity actions that must be refined using pseudo processes that can read all program variables.

$HAC_0: (rs = 0) \wedge (rr = 0) \wedge (bs = 1) \wedge (br = 0) \wedge (cs = -1) \wedge (cr = -1) \longrightarrow cs := 1;$
$HAC_1: (rs = 0) \wedge (rr = 0) \wedge (bs = 0) \wedge (br = 1) \wedge (cs = -1) \wedge (cr = -1) \longrightarrow cs := 0;$
$HAC_2: (rs = 0) \wedge (rr = 0) \wedge (bs = 1) \wedge (br = 1) \wedge (cs = -1) \wedge (cr = -1) \longrightarrow cr := 1;$
$HAC_3: (rs = 0) \wedge (rr = 0) \wedge (bs = 0) \wedge (br = 0) \wedge (cs = -1) \wedge (cr = -1) \longrightarrow cr := 0;$

The guards of the above actions are global state predicates that we refine using linear distributed detectors. Let G_i be the guard of the action HAC_i, where $0 \leq i \leq 3$. For example, we have $G_0 \equiv ((rs = 0) \wedge (rr = 0) \wedge (bs = 1) \wedge (br = 0) \wedge (cs = -1) \wedge (cr = -1))$. Corresponding to each global state predicate G_i, we use a distributed detector with two elements ds_i and dr_i, where ds_i is the local detector installed in the sender side and dr_i is the local detector installed in the receiver side. Next, we show how we add a linear distributed detector for the detection of G_0. We omit the presentation of the refinement of G_1, G_2, and G_3 as the approach is similar to the refinement of G_0.

Adding fault-tolerance components. To illustrate the structure of the linear detectors that we add to the ABP program, we now describe the refinement of G_0. Due to read restriction the sender (respectively, the receiver) cannot atomically detect G_0. However, the sender can detect a local condition $LC_s \equiv ((rs = 0) \wedge (bs = 1) \wedge (cs = -1))$. Respectively, the receiver can detect a local condition $LC'_r \equiv ((rr = 0) \wedge (br = 0) \wedge (cr = -1))$, where $G_0 \equiv (LC_s \wedge LC'_r)$. Now, we instantiate the required distributed detector by reusing the code of the pre-synthesized linear detectors presented in [1].

$$
\begin{array}{lll}
DAr_0 : (LC'_r) \wedge (y'_r = false) & \longrightarrow & y'_r := true; \\
DAs_0 : (LC_s) \wedge (y_s = false) \wedge (y'_r = true) & \longrightarrow & y_s := true;
\end{array}
$$

The action DAs_0 belongs to detector ds_0 that is allowed to read the witness predicate y'_r of the detector element dr_0 in the receiver side. If the detector element dr_0 detects its local predicate LC'_r then it will set its witness predicate y'_r to true. Then, if the condition LC_s holds in the sender side then the detector element ds_0 will detect the global state predicate G_0 by setting its witness predicate y_s to true. Afterwards, the synthesis algorithm adds the following write action to the sender process.

$$
Cs_0 : (y_s = true) \longrightarrow cs := 1;\ y_s := false;
$$

The synthesis algorithm adds similar distributed detectors to ABP in order to refine the global state predicates G_1, G_2, and G_3. Given the local conditions $LC'_s \equiv ((rs = 0) \wedge (bs = 0) \wedge (cs = -1))$ and $LC_r \equiv ((rr = 0) \wedge (br = 1) \wedge (cr = -1))$, we have the following logical equivalences:

- $G_1 \equiv (LC'_s \wedge LC_r)$
- $G_2 \equiv (LC_s \wedge LC_r)$
- $G_3 \equiv (LC'_s \wedge LC'_r)$.

Corresponding to global detection predicates $G_1 \cdots G_3$, we respectively add the following linear distributed detectors and also the necessary correcting action for recovery to the invariant:
Detecting G_1. This linear detector refines the guard of the action HTR_1 added by our synthesis algorithm.

$$
\begin{array}{lll}
DAr_1 : (LC_r) \wedge (y_r = false) & \longrightarrow & y_r := true; \\
DAs_1 : (LC'_s) \wedge (y'_s = false) \wedge (y_r = true) & \longrightarrow & y'_s := true;
\end{array}
$$

Correcting G_1. After the detection of G_1, the following write action takes place.

$$
Cs_1 : (y'_s = true) \longrightarrow cs := 0;\ y'_s := false;
$$

Detecting G_2. We use the following linear detector to refine the guard of the action HTR_2.

$$
\begin{array}{lll}
DAr_2 : (LC_r) \wedge (u_r = false) \wedge (u_s = true) & \longrightarrow & u_r := true; \\
DAs_2 : (LC_s) \wedge (u_s = false) & \longrightarrow & u_s := true;
\end{array}
$$

Correcting G_2. The following action recovers the state of the ABP program to the invariant S_{ABP} after the detection of the global state predicate G_2.

$$
Cr_2 : (u_r = true) \longrightarrow cr := 1;\ u_r := false;
$$

Detecting G_3. To detect the global state predicate G_3, we add the following detector to ABP.

$$
\begin{array}{lll}
DAr_3 : (LC'_r) \wedge (u'_r = false) \wedge (u'_s = true) & \longrightarrow & u'_r := true; \\
DAs_3 : (LC'_s) \wedge (u'_s = false) & \longrightarrow & u'_s := true;
\end{array}
$$

Correcting G_3. This action changes the state of the ABP program to a state in S_{ABP} after the detection of G_3.

$$
Cr_3 : (u'_r = true) \longrightarrow cr := 0;\ u'_r := false;
$$

The fault-tolerant ABP program. Next, we present the actions of the sender process in the resulting nonmasking fault-tolerant program.

$$
\begin{array}{lll}
Send'_0: & (rs = 1) & \longrightarrow \; rs := 0; cs := bs; cs := bs; \\
 & & \quad y_s := false; y_s := false; u'_s := false; u_s := false; \\
Send'_1: & (cr \neq -1) & \longrightarrow \; rs := 1; cr := -1; bs := (bs + 1) \mod 2; \\
 & & \quad y_s := false; y_s := false; u'_s := false; u_s := false; \\
DAs_0: & (LC_s) \wedge (y_s = false) \wedge (y'_r = true) & \longrightarrow \; y_s := true; \\
Cs_0: & (y_s = true) & \longrightarrow \; cs := 1; \; y_s := false; \\
DAs_1: & (LC'_s) \wedge (y'_s = false) \wedge (y_r = true) & \longrightarrow \; y'_s := true; \\
Cs_1: & (y'_s = true) & \longrightarrow \; cs := 0; \; y'_s := false; \\
DAs_2: & (LC_s) \wedge (u_s = false) & \longrightarrow \; u_s := true; \\
DAs_3: & (LC'_s) \wedge (u'_s = false) & \longrightarrow \; u'_s := true;
\end{array}
$$

The actions of the receiver in the synthesized fault-tolerant program are as follows:

$$
\begin{array}{lll}
Rec_0: & (cs \neq -1) & \longrightarrow \; cs := -1; rr := 1; br := (br + 1) \mod 2; \\
 & & \quad y_r := true; y'_r := true; u_r := true; u'_r := true; \\
Rec_1: & (rr = 1) & \longrightarrow \; rr := 0; cr := br; \\
 & & \quad y_r := true; y'_r := true; u_r := true; u'_r := true; \\
DAr_0: & (LC'_r) \wedge (y'_r = false) & \longrightarrow \; y'_r := true; \\
DAr_1: & (LC_r) \wedge (y_r = false) & \longrightarrow \; y_r := true; \\
DAr_2: & (LC_r) \wedge (u_r = false) \wedge (u_s = true) & \longrightarrow \; u_r := true; \\
Cr_2: & (u_r = true) & \longrightarrow \; cr := 1; \; u_r := false; \\
DAr_3: & (LC'_r) \wedge (u'_r = false) \wedge (u'_s = true) & \longrightarrow \; u'_r := true; \\
Cr_3: & (u'_r = true) & \longrightarrow \; cr := 0; \; u'_r := false;
\end{array}
$$

References

[1] S. S. Kulkarni and Ali Ebnenasir. Adding fault-tolerance using pre-synthesized components. *Technical report MSU-CSE-03-28, Department of Computer Science, Michigan State University, East Lansing, Michigan, USA. A revised version is available at* `http://www.cse.msu.edu/~sandeep/auto_component_techreport.ps`, 2003.