AN APPROACH TO SOFTWARE SAFETY ANALYSIS IN A DISTRIBUTED REAL-TIME SYSTEM

Paul V. Shebalin
Sang H. Son
Chun-Hyon Chang

Computer Science Report No. TR-88-13
May 20, 1988

AN APPROACH TO SOFTWARE SAFETY ANALYSIS IN A DISTRIBUTED REAL-TIME SYSTEM

Paul V. Shebalin
ORI, Inc.
601 Caroline Street
Fredericksburg, VA 22401

Sang H. Son and Chun-Hyon Chang
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

In many real-time applications, software systems have to cope with faults and failures to avoid disastrous results. Several techniques have been proposed for analyzing software safety in real-time systems, but, very few of them deal with distributed environments. In this paper we present a new software safety analysis approach for distributed systems based on a technique called *component message fault analysis*. This technique checks the safety-critical logic using the safety specification for different categories of component fault events, to uncover ambiguous safety requirements or design deficiencies. To demonstrate the power of this technique, a dual-purpose missile system is introduced and analyzed with regard to its software safety requirements.

1. INTRODUCTION

The use of computers for the real-time control and monitoring of physical processes has long been recognized as both the basis for unprecedented technological achievement and the potential source of concerns heretofore unknown in our society. One area of concern is the ability of computer-controlled systems to cause inadvertent damage of possibly enormous proportions. Based, in part, on the work of Leveson ([4],[5],[6],[7]), a great deal of interest has developed on the subject of software safety. The federal government has placed an increased emphasis on the safety of embedded software in such applications as air traffic control, avionics and military weapons systems ([1],[3],[4],[7]).

This work was partially supported by the Office of Naval Research under contract No. N00014-86-K-0245 to the Department of Computer Science, University of Virginia.

This emphasis on safety is indeed warranted. Because of software faults, people have been killed, equipment has been destroyed, and this country has been brought too close to the brink of nuclear war. [3] describes three cases, in 1960, 1979, and 1980 where ballistic missile early warning systems alerted operators to imaginary attacks. The incident in 1980 resulted in an increased defense condition and the scrambling of strategic bomber crews.

This paper presents a software safety analysis technique, called component message fault analysis(CMFA), for distributed systems based on fault tree analysis[1] and a modified approach to program control flow diagrams. Following a definition of distributed system software safety, the CMFA approach is described. Next, the paper provides a case study of a hypothetical distributed system and demonstrates an application of CMFA. Finally, observations are provided on the general applicability and usefulness of the CMFA approach.

2. SOFTWARE SAFETY IN A DISTRIBUTED SYSTEM

The development of safe real-time systems controlled by a single embedded computer is a complex and not well understood activity. The problem of ensuring the development of a safe system becomes even more complex and error-prone in the case of distributed computing systems. Because the control of safety-critical devices such as robot-arms, guided missiles and aircraft control surfaces becomes distributed in such systems, the ability of relatively remote computers to cause an undesired event to occur is a real possibility. Every software component in a safety-critical distributed system must be evaluated for its contribution to the safety of the entire system.

As shown in Figure 1 below, a distributed system consists of various autonomous components interconnected by communications, or message-passing, channels.

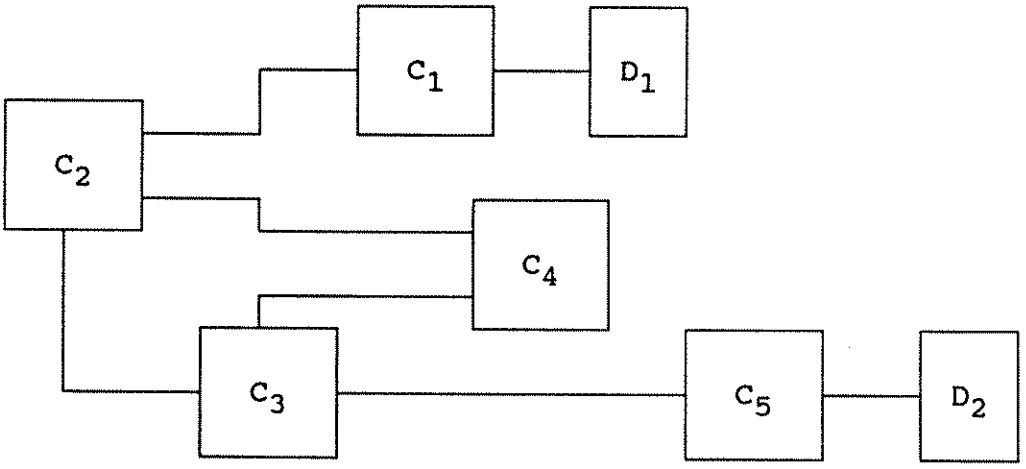

Figure 1. Distributed System

Formally, a distributed system, for the purposes of this paper, is defined to consist of:

1. A set of components $\{C_i\}$.

2. A **connected** relation, $\{(C_i,C_j)\}$, indicating a channel, or connection, between components C_i and C_j. The connection relation is symmetric.

3. A set of **safety critical devices**, $\{D_k\}$.

4. A **control** relation, $\{(C_i,D_k)\}$, indicating the immediate control of safety critical device D_k by component C_i.

5. A set of **messages**, $\{(M_n)\}$, which components may send to other components. A message may be a **command** message, an **information** message, or both a **command** and **information** message. A message consists of four components: (Originator, Destination, Action, Data). Separate messages sent from C_i to C_j will always arrive in the order sent if C_i is connected to C_j.

6. A set of **rules** for component behavior, which collectively provide the rules for system behavior. these rules are often represented as system requirement specifications, interface requirement specifications and other, similar, documents.

7. A **system state** definition which consists of a set of variable data items whose values are known, collectively, by the components of the system. A particular component may not know (have access to) all of the data items comprising the system state definition. At any time, the system state, s, is represented by the value of the data items. The set of all possible system states, $\{s\}$, is denoted as **S**.

8. A **component view** is the subset of system state data items which the component is privy to, through information in messages received by the component. For a component, C_i, the substate of the system offered by the component view is denoted as s^i. The set of all possible substates offered by the component view, $\{s^i\}$, is denoted as S^i.

Software by itself is not inherently safe or unsafe, but must be considered as being safe or unsafe within the context of a real-time system. Because a component, C_i, of a distributed system communicates with other components within the system by passing messages, the unsafe actions which C_i may take can be expressed in terms of messages and their contents. There are three categories of component fault events which may occur at C_i and which may contribute to a system mishap. These component fault event categories are general and may be applied to any component within the distributed system:

Inappropriate Command Transmission: C_i inappropriately sends a command, CMD_j, which may cause the system to move into or remain in a hazardous state.

Command Failure: C_i fails to send a command, CMD_k, when conditions warrant, which may cause the system to move into or remain in a hazardous state.

Hazardous Data: C_i sends, within an information message, incorrect data which may cause the system to move into or remain in a hazardous state.

The purpose of the component message fault analysis (CMFA) approach is to uncover errors in distributed software which could contribute to the component fault events listed above. The steps in conducting CMFA are described below.

3. CMFA APPROACH

Step 1

In Step 1, identify undesired events which the distributed system, as a whole, may cause.

Step 2

In Step 2, conduct a system fault tree analysis for each of the undesired system events, as described in [1].

Step 3

In Step 3, for each of the resulting system fault trees, identify the fault tree nodes which represent a hazardous action taken by a component of the distributed system, and define the system states which could exist when the hazardous action was taken. For each component, C_i, a set A_i, of (hazardous action, (system state)) pairs will have been determined.

Step 4

In Step 4, select a component, C_i. For each member of A_i, complete a fault tree, as described in [1]. This component fault tree will be very similar to the initial system fault tree. Within the component fault tree, identify each node which indicates that the component software takes an action corresponding to one of the three types of unsafe actions defined above in Section 4.2. Software fault events will fall into one of the following three software fault event categories:

Inappropriate Software Command Order: The software at C_i inappropriately orders a command message to be sent to another system component. The relevant component fault tree node will indicate "software inappropriately sends a <command> to component C_j."

Software Command Failure: The software at C_i fails to send a command that was warranted and required by system rules. The relevant component fault tree node will indicate "software fails to send <command> to component C_j."

Software Data Hazard: The software incorrectly modifies the value of a data item. The incorrect data value, when sent as part of a message, may cause another component to take a hazardous action. The relevant component fault tree node will indicate "software sends <data item> with incorrect value to component C_j."

The procedures for analyzing each of these three types of software fault events is described under Step 5, below.

Step 5

In Step 5, analyze each of the identified software fault events. The general procedure for analyzing a component's real-time program for safety follows:

1. Determine the software safety requirement associated with the software fault event. Before we can analyze software for safety, we need to understand the software safety requirement. This is accomplished by first defining, if not previously done, the component safety requirement (CSR) associated with the component fault event and then by defining the corresponding software safety requirement (SSR). The CSR is stated in terms of system parameters, and the SSR is stated, more precisely, in terms of program data items and their values. Both CSRs and SSRs consist of an <event> part and an <assertion> part. A CSR-SSR pair will fall into one of three categories:

   Type I Safety Requirements. Referred to as **SEND_ONLY_WHEN** requirements, requirements of this type specify the only conditions under which a message (command) may be sent.

   Type II Safety Requirements. Referred to as **SEND_IF** requirements, requirements of this type specify that certain messages must be sent if certain conditions exist.

   Type III Safety Requirements. Referred to as **DATA_REPORTING** requirements, requirements of this type specify that a safety-critical data item must be accurate when sent.

   As depicted in Figure 2, system(component) and software safety requirements are often not explicitly identified in either a system requirements specification or software requirements specification. These two specifications, whatever their formal names, are usually oriented toward functional and performance requirements. In many cases, safety requirements are an after-thought and often are not stated at all. The importance of properly defining CSRs and SSRs, therefore, cannot be understated.

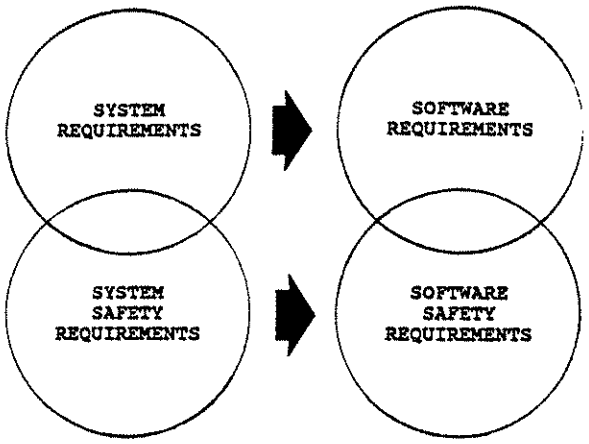

Figure 2. 'Adding-on' Safety Requirements.

2. Perform a safety-critical logic check. The purpose of the safety-critical logic check is to uncover logical errors which would be inconsistent with the software safety requirement and to determine where the program fails to verify software safety assertions. The check is performed by constructing a flow graph of the processing required for the safety-critical event. Depending on the software safety requirement type (i.e., I, II, III), either a forward flow graph(FFG) or a backward flow graph(BFG) is constructed. The use of an FFG or BFG allows the detection of program logic errors and the verification of software safety assertions.

   Definition: FFG - An FFG is a tree which contains two type of nodes, assertion nodes (rectangles) and event nodes, as shown in Figure 3. The direction of arcs in the FFG indicate movement away from the root of the tree, and represent the paths the program may logically take after the initiating external event has occurred. The assertion nodes represent assertions which need to be true (necessary conditions) in order for the path to continue to a terminating event. Where the necessary conditions do not exist (i.e., assertions are not true), the path ends in the null event, 'Ø'. The leaves of an FFG may be either a terminating event (e.g., message sent) or a null event. On a path, event nodes between the root and a leaf typically represent an executable program statement.

   Definition: BFG - A BFG, as with an FFG, is a tree which contains two types of nodes, assertion nodes (rectangles) and event nodes, as shown in Figure 4. The direction of arcs in the BFG indicate a backward flow from the leaves of the tree to the root, representing the backward tracing, from the root, of flow through the program and representing the paths the program may have taken to reach the terminating event. The assertion nodes represent assertions which must have been true (necessary conditions) for the path to have been taken. Where the necessary conditions should not have existed, the node pointed to (previous

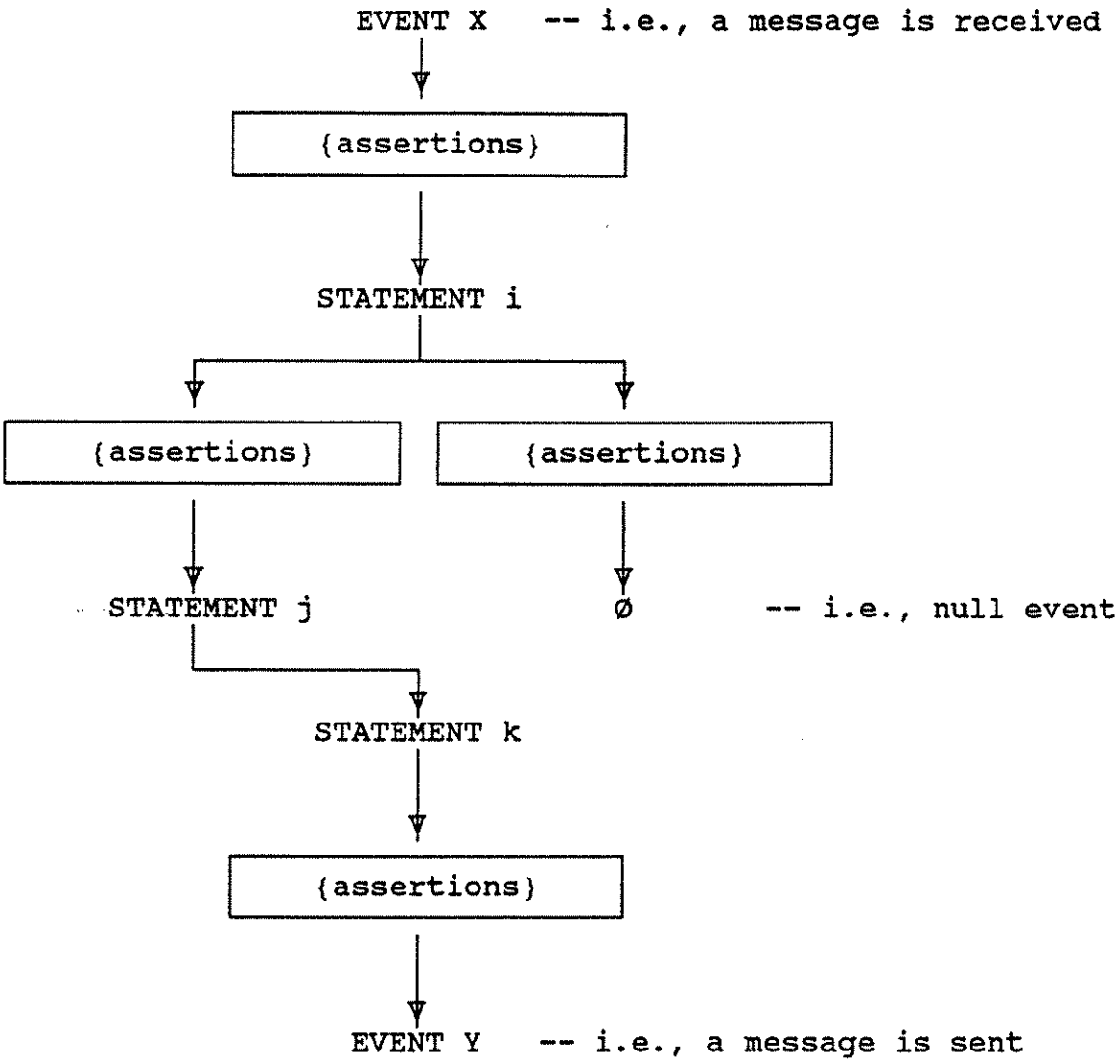

Figure 3. General format for an FFG.

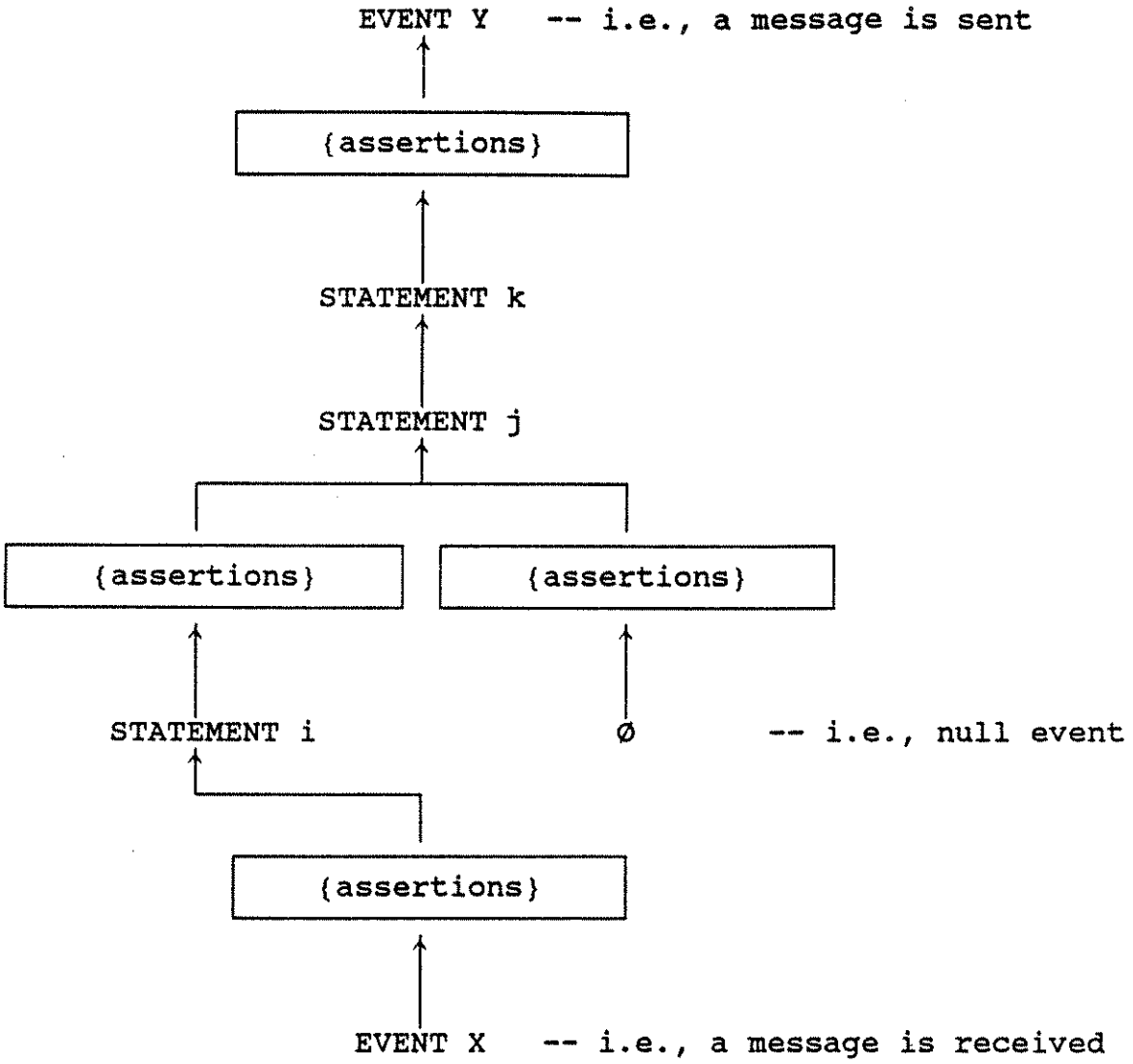

Figure 4. General format for a BFG.

event) is a null event, 'Ø'. The leaves of a BFG may be either an initiating event (e.g., message received) or a null event. On a path, event nodes between the root and a leaf typically represent a reverse sequence of program statements.

For each type of software fault event, there is a specific analysis procedure, as described below.

Step 5.1 Conduct a software safety analysis for Inappropriate Software Command Order, or Type I, events in two parts:

Part 1 A Type I, or SEND_ONLY_WHEN, safety requirement is associated with each Inappropriate Command Transmission event. As previously stated, a component safety requirement is first defined, followed by a software safety requirement. The general syntax for these Type I requirements is:

Type I CSR: <send message event> ONLY WHEN <system assertions>

Type I SSR: <execute statement to send message> ONLY WHEN <program assertions>

The format for <system assertions> is generally less restrictive then for <program assertions>. The <program assertions> are given in terms of program variables and their possible values.

Part 2 For a Type I SSR, a safety-critical logic check is performed by analyzing the processing flow for the <execute statement to send message> event with a BFG. The purpose of the check is to determine if the <execute statement to send message> event could occur when the <program assertions> were not true.

Step 5.2 Conduct a software safety analysis for Software Command Failure, or Type II, events in two parts:

Part 1 A Type II, or SEND_IF, safety requirement is associated with each Software Command Failure event. As previously stated, a component safety requirement is first defined, followed by a software safety requirement. The general syntax for these Type II requirements is:

Type II CSR: <send message event> IF <system assertions>

Type II SSR: <execute statement to send message> IF <program assertions>

As with Type I safety requirements, the format for <system assertions> is generally less restrictive then for <program assertions>. Again, the <program assertions> are given in terms of program variables and their possible values.

Part 2 For a Type II SSR, a safety-critical logic check is performed by analyzing the processing flow for the <execute statement to send message> event with an FFG, starting with the initiating event. The purpose of the check is to verify that the <execute statement to send message> event will occur when the <program assertions> are true.

Step 5.3: Conduct a software safety analysis for Software Data Hazard, or Type III, events is conducted in two parts:

Part 1 A Type III, or DATA_REPORTING safety requirement is associated with each Hazardous Data event. As previously stated, a component safety requirement is first defined, followed by a software safety requirement. The general syntax for these Type III requirements is:

Type III CSR: <data item reporting event> ONLY IF <system assertions>

Type III SSR: <data item update event> ONLY IF <program assertions>

As with Type I and II safety requirements, the format for <system assertions> is generally less restrictive then for <program assertions>. Again, the <program assertions> are given in terms of program variables and their possible values.

Part 2 For a Type III SSR, a safety-critical logic check is performed by analyzing the definition of safety-critical data items (the <data item update event>) with BFGs. The purpose of the check is to verify that the <data item update event> will occur only when the <program assertions> are true. The primary concern is that when data is sent to another component, the data must be accurate (consistent with the system state). A Type III safety requirement may have a companion Type I safety requirement.

Step 6

Perform an environmental dependence check. The purpose of the environmental dependence check is to minimize the program's susceptability to inconsistent environmental conditions. The assertions defined in an FFG or BFG contain data items which are critical to meeting a software safety requirement. Those data items whose values are set externally (defined by an incoming message) must be checked for consistency with the component view of the system. Of course, some external data must be taken as arbitrarily correct; there may be, however, opportunities to check external data for consistency. If the program fails to check for this consistency, it may violate a software safety requirement.

Based on the software fault events and associated software safety requirements determined in Step 5, identify the externally defined safety-critical data items used by the software. For each externally defined safety-critical data item, determine the consistency relations. A

consistency relation is one in which the data item is logically or arithmetically related to other data items using predicate calculus. Next, the software is examined to verify that the required consistency checks are in place. Each consistency relation not checked is noted and set aside for feedback to the software developer. By ensuring that proper consistency checks are made, the negative effects of inconsistent or conflicting external data can be minimized and the program made more safe.

4. HYPOTHETICAL CASE STUDY - DUAL-PURPOSE MISSILE SYSTEM

For the purpose of demonstrating CMFA, a hypothetical system has been defined. This system, the **Dual-Purpose Missile System** (DPMS), is concisely described below. Many of the details which would exist for an actual system are not provided. The DPMS is intended to serve as a pedagogical tool for demonstrating the CMFA approach to software safety analysis and may have ambiguous requirements or design deficiencies.

As seen in Figure 5, the DPMS is a distributed system.

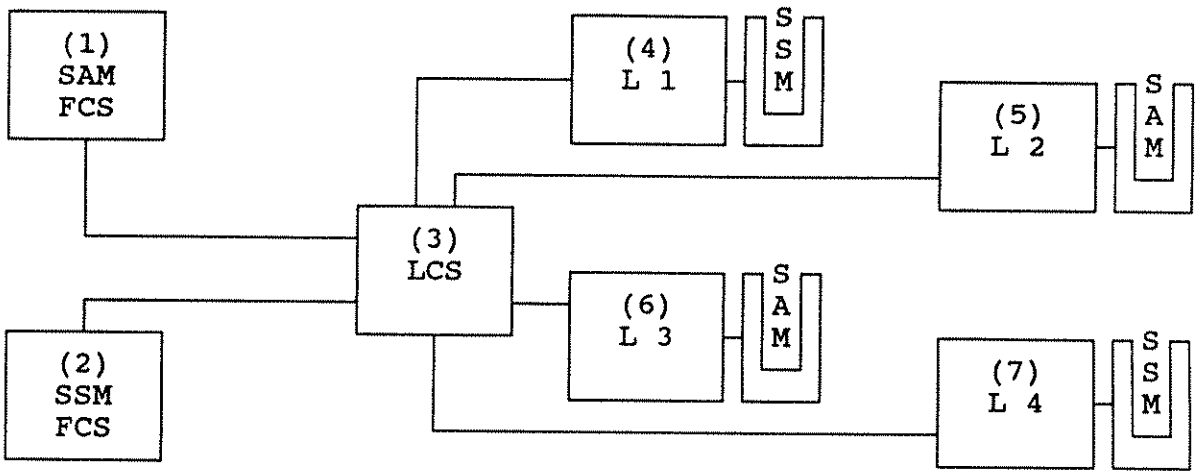

Figure 5. Dual-Purpose Missile System.

The DPMS is described as follows:

1. The components of the DPMS are

Name	ID
a. Surface-to-Air Missile Fire Control System (SAMFCS)	1
b. Surface-to-Surface Missile Fire Control System (SSMFCS)	2
c. Launch Control System (LCS)	3
d. Launcher 1 (L1)	4
e. Launcher 2 (L2)	5
f. Launcher 3 (L3)	6
g. Launcher 4 (L4)	7

2. The connected relation for the DPMS, in terms of component ID's, is:

{(1,3),(2,3),(3,4),(3,5),(3,6),(3,7)}

3. The Safety Critical Devices, $\{D_k\}$, in the DPMS are

   a. Missile Canister 1 (D_1)
   b. Missile Canister 2 (D_2)
   c. Missile Canister 3 (D_3)
   d. Missile Canister 4 (D_4)

4. The control relation for the DPMS is:

   $\{(4,D_1),(5,D_2),(6,D_3),(7,D_4)\}$

5. The messages, $\{M_n\}$, available to components within the DPMS are:

Message	Name(Action)	Type
M_1	PREPARE	Command, Information
M_2	DATA_XFER	Command, Information
M_3	LAUNCH	Command, Information
M_4	SAFE	Command, Information
M_5	STATUS_REQUEST	Command, Information
M_6	STATUS_RESPONSE	Command, Information

6. The following rules apply to the DPMS:

   a. SAMFCS (Component Number = 1)

      (1) The SAMFCS is responsible for launching SAMs.

      (2) Except for STATUS_REQUEST messages, the SAMFCS may send COMMAND messages only to a LAUNCHER with a SAM in its canister.

      (3) The SAMFCS may not send DATA_XFER or STATUS_RESPONSE messages.

      (4) The SAMFCS may send a STATUS_REQUEST message to any LAUNCHER.

      (5) To launch a SAM, the SAMFCS sends a PREPARE message followed by a LAUNCH message.

      (6) The SAMFCS sends a LAUNCH message to a LAUNCHER only after receiving a STATUS_RESPONSE message for that LAUNCHER indicating READY.

      (7) The SAMFCS may send a SAFE message or STATUS_REQUEST message at any time.

   b. SSMFCS (Component Number = 2)

      (1) The SAMFCS is responsible for launching SSMs.

      (2) Except for STATUS_REQUEST messages, the SSMFCS may send COMMAND messages only to a LAUNCHER with a SSM in its canister.

(3) The SSMFCS may not send PREPARE or STATUS_RESPONSE messages.

(4) The SSMFCS may send a STATUS_REQUEST message to any LAUNCHER.

(5) To launch a SSM, the SSMFCS sends one or more DATA_XFER messages followed by a LAUNCH message. To safely launch an SSM, the SSMFCS must download the missile software into the onboard computer, establish SSM inertial reference by initializing the missile inertial platform, and load target data into the onboard computer. The onboard program must be loaded before commencing inertial platform initialization or target data loading. (6) The SSMFCS sends a LAUNCH message to a LAUNCHER only after receiving a STATUS_RESPONSE message for that LAUNCHER indicating READY.

(7) The SSMFCS may send a SAFE message or STATUS_REQUEST message at any time.

c. LCS (Component Number = 3)

(1) The LCS relays COMMAND messages from one of the fire control systems (SAMFCS or SSMFCS) to a LAUNCHER when the missile type and COMMAND(action) match the sending fire control system and missile state. Otherwise, the LCS ignores the COMMAND.

(2) The LCS relays STATUS_REQUEST messages to the indicated LAUNCHER.

(3) The LCS relays STATUS_RESPONSE messages to the indicated fire control system.

d. LAUNCHERs (Component Numbers 4, 5, 6, 7)

(1) The LAUNCHER Canister may be loaded with either a SAM or SSM (unless it is empty).

(2) LAUNCHERs take the Action indicated in COMMAND messages.

(3) Upon receiving STATUS_REQUEST messages, LAUNCHERs determine missile status and transmit a STATUS_RESPONSE message to the Originator. The first data word of the STATUS_RESPONSE message is loaded with missile status, while the second data word is loaded with missile type.

7. The system state definition consists of the following:

Data Item	Initial Value
LAUNCHER 1 Missile Type	NONE
LAUNCHER 1 Missile State	SAFE
LAUNCHER 2 Missile Type	NONE
LAUNCHER 2 Missile State	SAFE

LAUNCHER 3 Missile Type	NONE
LAUNCHER 3 Missile State	SAFE
LAUNCHER 4 Missile Type	NONE
LAUNCHER 4 Missile State	SAFE

A real system would have many more data items to define the state of the system. The limited number of data items above are sufficient for the purposes of illustration.

8. Table 1 provides the component views within the DPMS. Data items within the view of a component are indicated by an 'X' in the appropriate row and column.

Table 1. DPMS Component Views

	Component						
	SAM FCS	SSM FCS	LCS	L1	L2	L3	L4
LAUNCHER 1							
Missile Type	X	X	X	X			
Missile State	X	X	X	X			
LAUNCHER 2							
Missile Type	X	X	X		X		
Missile State	X	X	X		X		
LAUNCHER 3							
Missile Type	X	X	X			X	
Missile State	X	X	X			X	
LAUNCHER 4							
Missile Type	X	X	X				X
Missile State	X	X	X				X

Specific algorithms and software architecture will only be provided for one component of the DPMS, the LCS component. As described below, the LCS is the central component in the DPMS and acts as a type of message switching center for the subscriber fire control systems.

As shown below in Figure 6, the LCS has six full-duplex input/output channels, designated 1, 2, 4, 5, 6, and 7, which, for convenience, correspond to DPMS components having those same IDs. The software within the LCS consist of four processes implemented as ADA[9] tasks. Although not described, the six channel devices can also be considered processes (tasks) in that they operate in parallel with the LCS tasks and non-deterministically notify the RECEIVER task of messages which have arrived. The four ADA tasks in the LCS are SENDER, RECEIVER, SAM_LAUNCH_CONTROL, and SSM_LAUNCH_CONTROL. Although a

reasonable attempt has been made to construct valid ADA program segments, there may be a few bugs. Nonetheless, the ADA package and tasks given below(Figures 7 through 11) provide an example with which to demonstrate the CMFA approach.

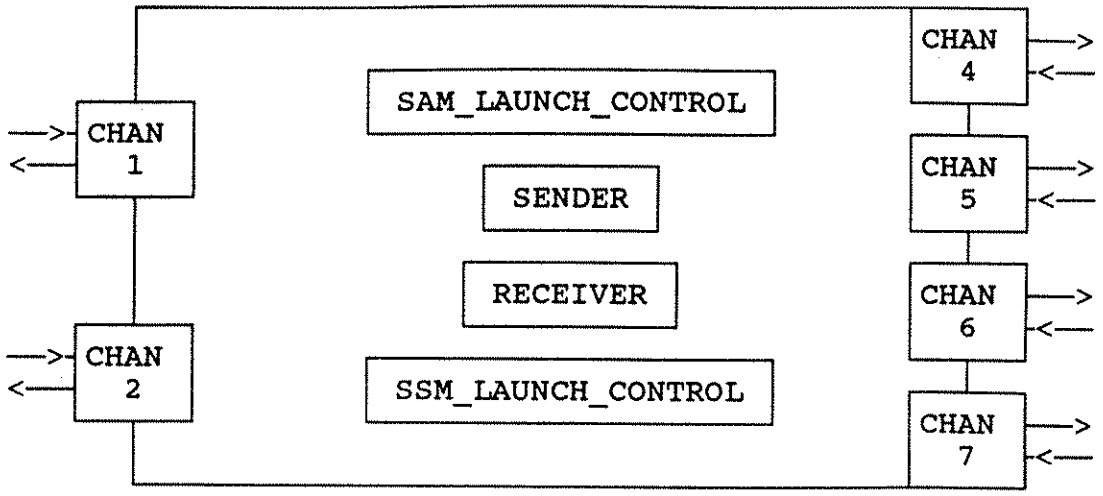

Figure 6. Launch Control System.

5. APPLICATION OF CMFA

Using the DPMS as an example, a sample application of CMFA is described below:

Step 1

This step of the CMFA procedure requires that the undesired events in the DPMS be identified. For our purposes, only one undesired event needs to be identified: The premature launch of an SSM. The event is considered a safety problem because it may result in a missile which takes an unpredictable flight path, possibly returning to the launch site or striking an unintentional target. By premature, we mean that the SSM is not ready, that is, it may not have received its entire guidance program, the inertial reference may not have been established, or target data may not have been completely loaded.

Step 2

This step requires that a system fault tree be constructed for the undesired event. This event is formally called LAUNCHER_N PREMATURELY LAUNCHES SSM. LAUNCHER_N represents any of the four possible launchers. It is assumed that LAUNCHER_N holds an SSM. The system fault tree for the event, LAUNCHER_N PREMATURELY LAUNCHES SSM, is provided in Figure 12. Several symbols used in Figure 12 and subsequent figures are slightly different than in [1], but their meaning should be clear. The 'AND' and 'OR' gates, and the pointers to other fault trees are depicted as small boxes. The AND and OR boxes are marked clearly. Pointers to other fault trees are marked with 'Ci', where i is an integer, or are left blank. 'C' signifies a component fault tree and a blank pointer box indicates

```
package LAUNCH_DATA is

   type MISSILE_TYPE is (SAM,   -- surface to air missile
                         SSM,   -- surface to surface missile
                         NONE);-- no missile

   type STATE_TYPE is (SAFE,            -- missile is in a safe condition
                       IN_PREP,         -- SAM is in prep for launching
                       PGM_LOADED,      -- SSM guidance program loaded
                       INER_REF_EST,    -- SSM inertial reference established
                       TGT_DATA_LOADED,-- SSM target data loaded
                       READY,           -- missile is ready for launch
                       MSL_AWAY,        -- missile has left the launcher
                       IN_SAFING,       -- missile is being safed
                       INOP,            -- missile is inoperative
                       NOT_LAUNCHER);   -- component is not a launcher


   type LAUNCHER_TYPE is
     record
       MISSILE: MISSILE_TYPE;
       STATE: STATE_TYPE;
     end record;

   type ACTION_TYPE is (PREPARE,          -- prepare the SAM for launch
                        DATA_XFER,        -- transfer data to SSM
                        LAUNCH,           -- launch the missile
                        SAFE,             -- safe the missile
                        STATUS_REQUEST,   -- request for status
                        STATUS_RESPONSE);-- response to status request

   subtype CS_TYPE is INTEGER range 1..7;

   CHANNEL: array(CS_TYPE) of CS_TYPE := (1,2,3,4,5,6,7);

   subtype RTC_TYPE is LONG_INTEGER;  -- type of real-time clock value

   type MESSAGE_TYPE(N: INTEGER) is
     record
       ORIGINATOR: CS_TYPE;           -- who sent the message
       DESTINATION: CS_TYPE;          -- who is to receive the message
       ACTION: ACTION_TYPE;           -- what action is the dest. to perform
       DATA: array(1..N) of INTEGER; -- data relevant to the action
       CHANNEL: CS_TYPE;              -- channel over which message was recd
       TIME: RTC_TYPE;                -- time of receipt
     end record;

   LAUNCHER: array(CS_TYPE) of LAUNCHER_TYPE; -- missile type and state of
                                              -- missile, if launcher
   CHANNEL_MSL: array(CS_TYPE) of MISSILE_TYPE;
```

Figure 7. Data Definitions for the LCS program. (continued on next page).

```
   SAM_FCS: constant INTEGER := 1;
   SSM_FCS: constant INTEGER := 2;
   LAUNCHER_1: constant INTEGER := 4;
   LAUNCHER_2: constant INTEGER := 5;
   LAUNCHER_3: constant INTEGER := 6;
   LAUNCHER_4: constant INTEGER := 7;

   procedure IGNORE(MSG: in MESSAGE_TYPE) is    end IGNORE;
     -- ignore message and release message record


   function RTC return RTC_TYPE is begin    end;


end; -- end of LAUNCH_DATA package specification

package body LAUNCH_DATA is

--

end LAUNCH_DATA;
```

Figure 7(continued). Data Definitions for the LCS program.

```
   task SAM_LAUNCH_CONTROL is
   use LAUNCH_DATA;

     entry MESSAGE(MSG: in MESSAGE_TYPE);

   end;

 1 task body SAM_LAUNCH_CONTROL is
 2 use LAUNCH_DATA;
 3
 4 begin
 5   loop
 6     accept MESSAGE(MSG: in MESSAGE_TYPE)
 7       do
 8         case MSG.ACTION is
 9           when PREPARE =>
10             case LAUNCHER(MSG.DESTINATION).STATE is
11               when SAFE => SENDER.SEND(MSG);
12               when others => IGNORE(MSG);
13             end case;
14           when LAUNCH =>
15             case LAUNCHER(MSG.DESTINATION).STATE is
16               when READY => SENDER.SEND(MSG);
17               when others => IGNORE(MSG);
18             end case;
19           when SAFE =>
20             case LAUNCHER(MSG.DESTINATION).STATE is
21               when MSL_AWAY, IN_SAFING => IGNORE(MSG);
22               when others => SENDER.SEND(MSG);
23             end case;
24           when STATUS_REQUEST => SENDER.SEND(MSG);
25           when others => IGNORE(MSG);
26         end case;
27       RECEIVER.RESET(MSG.CHANNEL);
28     end MESSAGE;
29   end loop;
30 end SAM_LAUNCH_CONTROL;
```

Figure 8. SAM_LAUNCH_CONTROL Task (TASK A).

```
   task SSM_LAUNCH_CONTROL is
   use LAUNCH_DATA;

     entry MESSAGE(MSG: in MESSAGE_TYPE);

   end;

 1 task body SSM_LAUNCH_CONTROL is
 2 use LAUNCH_DATA;
 3
 4 begin
 5   loop
 6     accept MESSAGE(MSG: in MESSAGE_TYPE)
 7       do
 8         case MSG.ACTION is
 9           when DATA_XFER =>
10             case LAUNCHER(MSG.DESTINATION).STATE is
11               when SAFE, PGM_LOADED, INER_REF_EST, TGT_DATA_LOADED
12                 => SENDER.SEND(MSG);
13               when others => IGNORE(MSG);
14             end case;
15           when LAUNCH =>
16             case LAUNCHER(MSG.DESTINATION).STATE is
17               when READY => SENDER.SEND(MSG);
18               when others => IGNORE(MSG);
19             end case;
20           when SAFE =>
21             case LAUNCHER(MSG.DESTINATION).STATE is
22               when MSL_AWAY, IN_SAFING => IGNORE(MSG);
23               when others => SENDER.SEND(MSG);
24             end case;
25           when STATUS_REQUEST => SENDER.SEND(MSG);
26           when others => IGNORE(MSG);
27         end case;
28       RECEIVER.RESET(MSG.CHANNEL);
29     end MESSAGE;
30   end loop;
31 end SSM_LAUNCH_CONTROL;
```

Figure 9. SSM_LAUNCH_CONTROL Task (TASK B).

```
   task RECEIVER is
   use LAUNCH.DATA;

     entry RESET(CHANNEL: in CS_TYPE); -- from SAM_LAUNCH_CONTROL or
                                       --      SSM_LAUNCH_CONTROL

     entry RECEIVE_MESSAGE(CHANNEL: in CS_TYPE; MSG: out MESSAGE_TYPE);
                                       -- from one of the input devices
   end;

 1 task body RECEIVER is
 2 use LAUNCH.DATA;
 3
 4 begin
 5   loop
 6     select
 7       accept RESET(CHANNEL: in CS_TYPE); -- reset input device to
 8         do                               -- accept a message
 9           -- reset the channel identified by CHANNEL
10         end RESET;
11     or
12       accept RECEIVE_MESSAGE(MSG: in MESSAGE_TYPE);
13         do             -- get a message from one of the input devices
14           case MSG.ORIGINATOR is
15             when SAM_FCS => SAM_LAUNCH_CONTROL.MESSAGE(MSG);
16             when SSM_FCS => SSM_LAUNCH_CONTROL.MESSAGE(MSG);
17             when LAUNCHER_1, LAUNCHER_2, LAUNCHER_3, LAUNCHER_4 =>
18               if MSG.ACTION = STATUS_RESPONSE then
19                 LAUNCHER(MSG.ORIGINATOR).STATE :=
20                                 STATE_TYPE'VAL(MSG.DATA(1));
21                 SENDER.SEND(MSG);
22               end if;
23             when others => IGNORE(MSG);
24           end case;
25         end RECEIVE_MESSAGE;
26     end select;
27   end loop;
28 end RECEIVER;
```

Figure 10. RECEIVER Task (TASK C).

```
   task SENDER is
   use LAUNCH_DATA;

     entry SEND(MSG: in MESSAGE_TYPE);

   end;

 1 task body SENDER is
 2 use LAUNCH_DATA;
 4
 5 begin
 6   loop
 7     accept SEND(MSG: in MESSAGE_TYPE);
 8       -- reset the output device identified by CHANNEL(MSG.DESTINATION);
 9       -- set the output device to send the MSG to MSG.DESTINATION;
10   end loop;
11 end SENDER;
```

Figure 11. SENDER Task (TASK D).

only that the elaboration of that fault path was omitted in the example. A box with 'Si' indicates a software fault event which is analyzed further with flow graphs.

Step 3

This step requires that the nodes representing hazardous action by components of the DPMS be identified. For the purpose of demonstration, the LCS component will be used. For each component fault type (i.e., I,II,III), one representative component fault will be analyzed. These component faults, found initially in Figures 12 and 13, are:

Fault	Type	Fault Event
C1	I	LCS inappropriately sends LAUNCH command to LAUNCHER_N.
C2	II	LCS fails to send DATA_XFER message to LAUNCHER_N.
C3	III	LCS sends incorrect missile READY status for LAUNCHER_N to SSM FCS.

The system states for each of these events is described below in Table 2.

Table 2. System states for component fault events C1, C2, C3

Event	System State
C1	{*s* | HAS_SSM(LAUNCHER_N) AND NOT_READY(LAUNCHER_N)} In other words, the required system associated with the component fault event C1 is one in which LAUNCHER_N has an SSM which is not fully prepared(not ready) for launch. The predicate, HAS_SSM, specifies that LAUNCHER_N has an SSM and the predicate, NOT_READY, specifies that the missile in LAUNCHER_N is not ready for launch.
C2	{*s* | HAS_SSM(LAUNCHER_N) AND [SAFE(LAUNCHER_N) OR PGM_LOADED(LAUNCHER_N) OR INER_REF_EST(LAUNCHER_N) OR TGT_DATA_LOADED(LAUNCHER_N)]} The required system state associated with component fault event C2 is one in which LAUNCHER_N has an SSM which may be in the process of being prepared for launch; the SSM state is either SAFE, PGM_LOADED, INER_REF_EST, or TGT_DATA_LOADED.
C3	{*s* | HAS_SSM(LAUNCHER_N)} The only requirement for a system state for component fault event C3 is that LAUNCHER_N have an SSM. No matter what the state of the SSM, it is unsafe for the LCS to send an incorrect missile status to the SSM_FCS.

Step 4

This step requires that for the component, LCS, a component fault tree be produced for each hazardous action pair. For our example, the three component fault events, C1, C2, C3, will be considered. The system states associated with each component fault event (hazardous action) are used in preparing the component fault trees, which are shown as Figures 13, 14, and 15. In Figure 13, the fault tree for the component fault event, "LCS INAPPROPRIATELY SENDS LAUNCH COMMAND TO LAUNCHER_N," identifies the Type I software fault event S1, "LCS SOFTWARE INAPPROPRIATELY ORDERS LAUNCH COMMAND TO LAUNCHER_N." In Figure 14, the fault tree for the component fault event, "LCS FAILS TO SEND DATA_XFER MESSAGE TO SSM IN LAUNCHER_N," identifies the Type II software fault event S2, "LCS SOFTWARE FAILS TO ORDER DATA_XFER MESSAGE TO LAUNCHER_N." In Figure 15, the fault tree for the component fault event, "LCS SENDS INCORRECT MISSILE READY STATUS FOR LAUNCHER_N TO SSM_FCS," identifies the Type III software fault event S3, "LCS SOFTWARE SENDS STATUS RESPONSE MESSAGE INCORRECTLY INDICATING 'READY'." The three software fault events of interest are summarized in Table 3.

Table 3. Software Fault Events S1, S2, S3

Component Fault Event	Software Fault Event	Type	Description
C1	S1	I	LCS SOFTWARE INAPPROPRIATELY ORDERS LAUNCH COMMAND TO LAUNCHER_N
C2	S2	II	LCS SOFTWARE FAILS TO ORDER DATA_XFER MESSAGE TO LAUNCHER_N
C3	S3	III	LCS SOFTWARE SENDS STATUS RESPONSE MESSAGE INCORRECTLY INDICATING 'READY'

Step 5

This step requires that each of the identified software fault events be analyzed according to its type. For our example, events S1, S2, and S3 are analyzed using Steps 5.1, 5.2, and 5.3, respectively.

Event S1 Since S1 is a Type I software fault event, Step 5.1 is taken.

Step 5.1 Part I Part I of Step 5.1 requires that component and software safety requirements be defined for event S1; these are shown below in Table 4.

Table 4. Safety Requirements for Event S1

Software Fault Event:	LCS SOFTWARE INAPPROPRIATELY ORDERS LAUNCH COMMAND TO LAUNCHER_N
Type: I (SEND_ONLY_WHEN)	
Component Safety Requirement:	SEND (SSM) LAUNCH Command to LAUNCHER_N with MISSILE(LAUNCHER_N) = SSM ONLY WHEN STATE(LAUNCHER_N) = READY AND MESSAGE ORIGINATOR = SSM_FCS AND MESSAGE DELAY < MAX ALLOWABLE
Software Safety Requirement:	EXECUTE Statements D8..D9 with MSG.DESTINATION = LAUNCHER_N AND <conditions> MSG.ACTION = LAUNCH ONLY WHEN LAUNCHER[LAUNCHER_N].STATE = READY <program AND MSG.ORIGINATOR = SSM_FCS assertions> AND MSG.CHANNEL = CHANNEL[SSM_FCS] AND MSG.RTC < MAX_DELAY AND LAUNCHER[LAUNCHER_N].MISSILE = SSM

Step 5.1 Part 2 The safety-critical logic check for the software safety requirement shown in Table 4 results in the BFG of Figure 16. It appears that there are two paths which can logically lead to the execution of statements represented by comments D8..D9. As shown in Table 5, each path has a different set of assertions. The Path 1 assertions do not match the <program assertions> in the Type I software safety requirement (Table 4), while there is a match between the Path 2 assertions and the <program assertions>. Closer examination reveals that, because the <program assertions> are not verified, the program will allow the SAM_FCS(or even a launcher), masquerading as the SSM_FCS, to send a LAUNCH command to LAUNCHER_N. If the DPMS works correctly, this should not happen. Nonetheless, safety is not ensured by naively relying on the propriety of external components.

Table 5. Paths Leading to Statements D8..D9.

Path 1:	D7 - A16 - A6 - C15 - C12 - <LAUNCH CMD from SAM_FCS>
Path 1 Assertions:	MSG.ORIGINATOR = SAM_FCS AND MSG.CHANNEL = CHANNEL[SAM_FCS] AND MSG.DESTINATION = LAUNCHER_N AND MSG.ACTION = LAUNCH AND LAUNCHER[LAUNCHER_N].STATE = READY

Table 5. Paths Leading to Statements D8..D9 (Continued)

Path 2: D7 - B17 - B6 - C16 - C12 - <LAUNCH CMD from SSM_FCS>

Path 1 Assertions: MSG.ORIGINATOR = SSM_FCS AND
MSG.CHANNEL = CHANNEL[SSM_FCS] AND
MSG.DESTINATION = LAUNCHER_N AND
MSG.ACTION = LAUNCH AND
LAUNCHER[LAUNCHER_N].STATE = READY

Event S2 Since S2 is a Type II software fault event, **Step 5.2** is taken.

Step 5.2 Part I Part I of Step 5.2 requires that component and software safety requirements be defined for event S2; these are shown below in Table 6.

Table 6. Safety Requirements for Event S2

Software Fault Event:	LCS SOFTWARE FAILS TO ORDER DATA_XFER MESSAGE TO LAUNCHER_N
Type: II (SEND_IF)	
Component Safety Requirement:	SEND DATA_XFER MESSAGE TO LAUNCHER_N WITHIN ALLOWABLE MAX DELAY IF DATA_XFER MESSAGE RECEIVED FROM SSM_FCS AND LAUNCHER[LAUNCHER_N].MISSILE = SSM AND LAUNCHER[LAUNCHER_N].STATE E {SAFE,PGM_LOADED,INER_REF_EST,TGT_DATA_LOADED}
Software Safety Requirement:	**EXECUTE** Statements D8..D9 with MSG.DESTINATION = LAUNCHER_N AND <conditions> MSG.ACTION = DATA_XFER IF MSG.ORIGINATOR = SSM_FCS <program assertions>

Step 5.2 Part 2 The safety-critical logic check for the software safety requirement shown in Table 6 results in the FFG of Figure 17. It appears that a DATA_XFER message will be expeditiously handled by the LCS software. Not all SEND_IF software safety requirements involve the receipt of an external message, however. Examples of other initiating events are losses of communication (time-outs) and failed consistency checks.

Event S3. Since S3 is a Type III software fault event, **Step 5.3** is taken.

Step 5.3 Part I of Step 5.3 requires that component and software safety requirements be defined for event S3; these are shown below in Table 7.

Table 7. Safety Requirements for Event S3

Software Fault Event:	LCS SOFTWARE SENDS STATUS_RESPONSE MESSAGE INCORRECTLY INDICATING 'READY'
Type: III (DATA_REPORTING)	
Component Safety Requirement:	REPORT STATUS OF LAUNCHER_N TO SSM_FCS AS 'READY' **ONLY IF** LAST STATUS_RESPONSE MESSAGE RECEIVED FROM LAUNCHER_N INDICATES MSG.ORIGINATOR = LAUNCHER_N AND MSG.ACTION = STATUS_RESPONSE AND MSG.DATA[1] = 'READY' AND MSG.CHANNEL = LAUNCHER_N's CHANNEL
Software Safety Requirement:	**EXECUTE** Statements C19 **ONLY IF** MSG.ORIGINATOR E {LAUNCHER_1, LAUNCHER_2, LAUNCHER_3, LAUNCHER_4} AND MSG.ACTION = STATUS_RESPONSE AND MSG.CHANNEL = CHANNEL_MSL[MSG.ORIGINATOR]

This example is not the most appropriate to illustrate the importance of properly updating a safety-critical data item because the data item is not directly used in status reporting. The status reported by the LCS is actually the unaltered status received from the responding launcher. Nonetheless, statement C19 is representative of a program update of a safety-critical data item. In another system, the determination of a 'READY' status may involve several independent data items whose values become known at different times. In such a situation, the relevance of Type III software safety requirements becomes more clear. It is important to realize that the component safety requirement does not require that the status be updated, but that, if and when it is updated, it is done correctly.

Step 5.3 Part 2 The safety-critical logic check for the software safety requirement shown in Table 7 results in the BFG of Figure 18. By comparing the <program assertions> in the software safety requirement with the assertions evident in the BFG of Figure 18, we can see that the program does not verify that the channel was appropriate. This needs to be done to ensure that LAUNCHER_N's STATUS_RESPONSE did, indeed, come from LAUNCHER_N.

Step 6

This step requires that, based on the results of Step 5, the externally defined safety-critical data items be identified and their consistency relations be determined. Although all consistency relations will not be developed, examples are provided in Table 8, below:

Table 8. Examples of Consistency Relations for Externally Defined Safety-Critical Data Items

Externally Defined Safety-Critical Data Item	Consistency Relation
MSG.ORIGINATOR	CHANNEL_MSL[MSG.ORIGINATOR] = MSG.CHANNEL
MSG.DESTINATION	MSG.DESTINATION E {SAM_FCS, SSM_FCS} ==> ¬(MSG.ACTION = STATUS_REQUEST)
MSG.ACTION	MSG.ACTION = DATA_XFER ==> MSG.ORIGINATOR = SSM_FCS
MSG.DATA	MSG.ACTION = DATA_XFER ==> SIZE(MSG.DATA) > 0
MSG.CHANNEL	MSG.CHANNEL > 0 AND MSG.CHANNEL < 8
MSG.TIME	MSG.TIME > RTC + MAX_DELAY

It is apparent, after examining the LCS software, that the verification of these consistency relations is not implemented as consistency checks. Although it will not be done here, several modifications to include consistency checks are obvious and will result in 'safer' software.

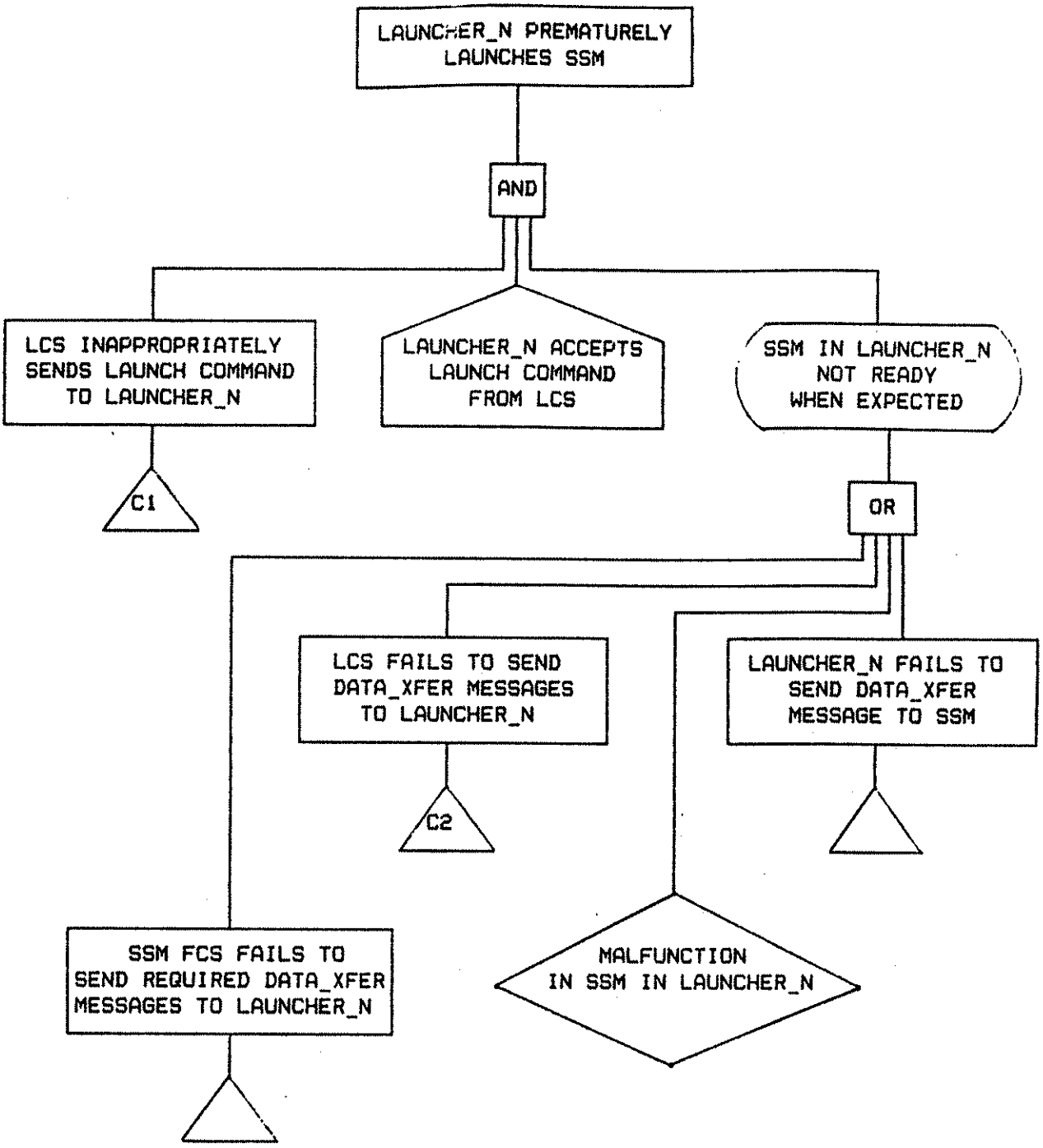

Figure 12. System Fault Tree: Launcher_N Prematurely Launches SSM.

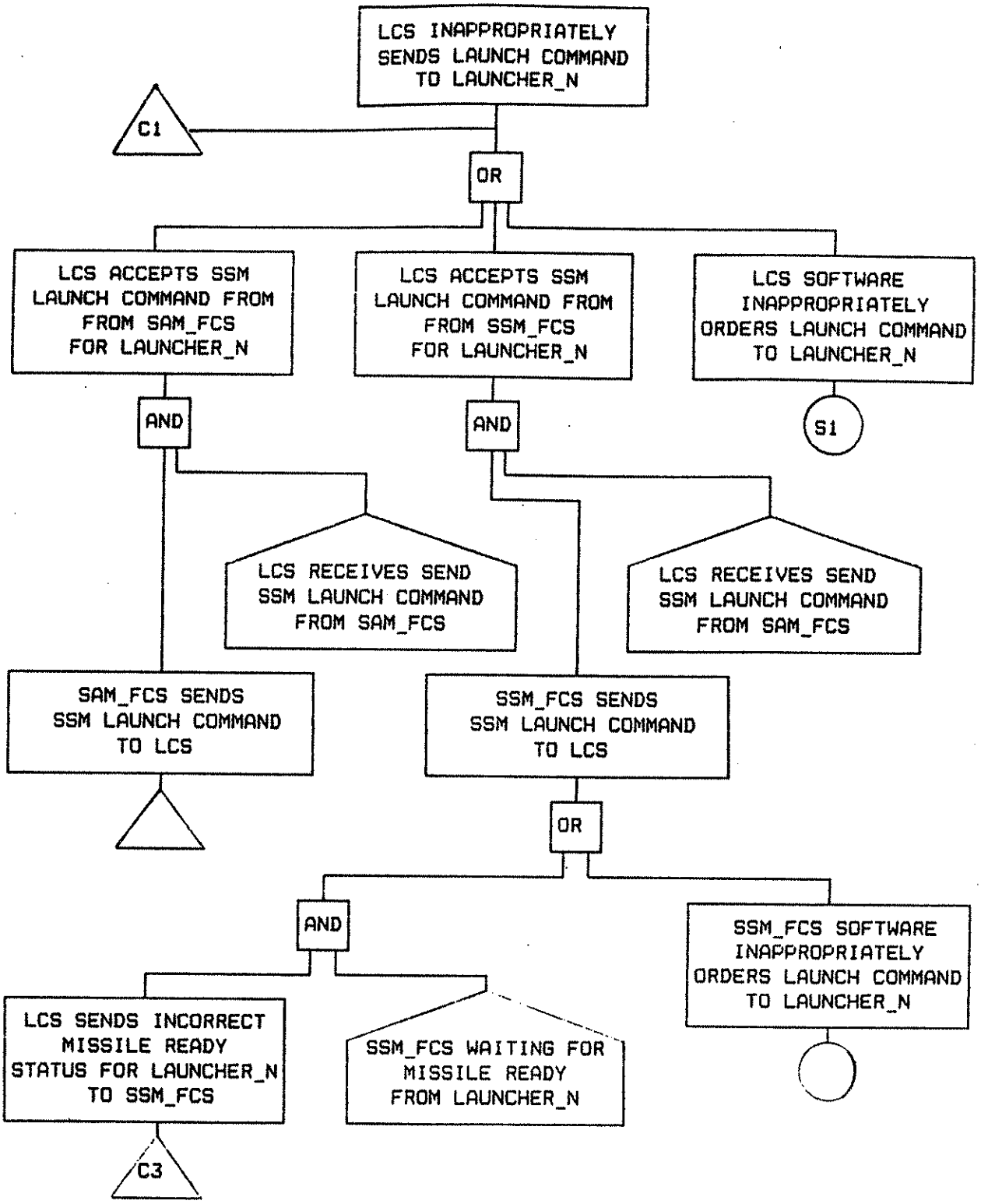

Figure 13. Component Fault Tree for 'LCS INAPPROPRIATELY SENDS LAUNCH COMMAND TO LAUNCHER_N'.

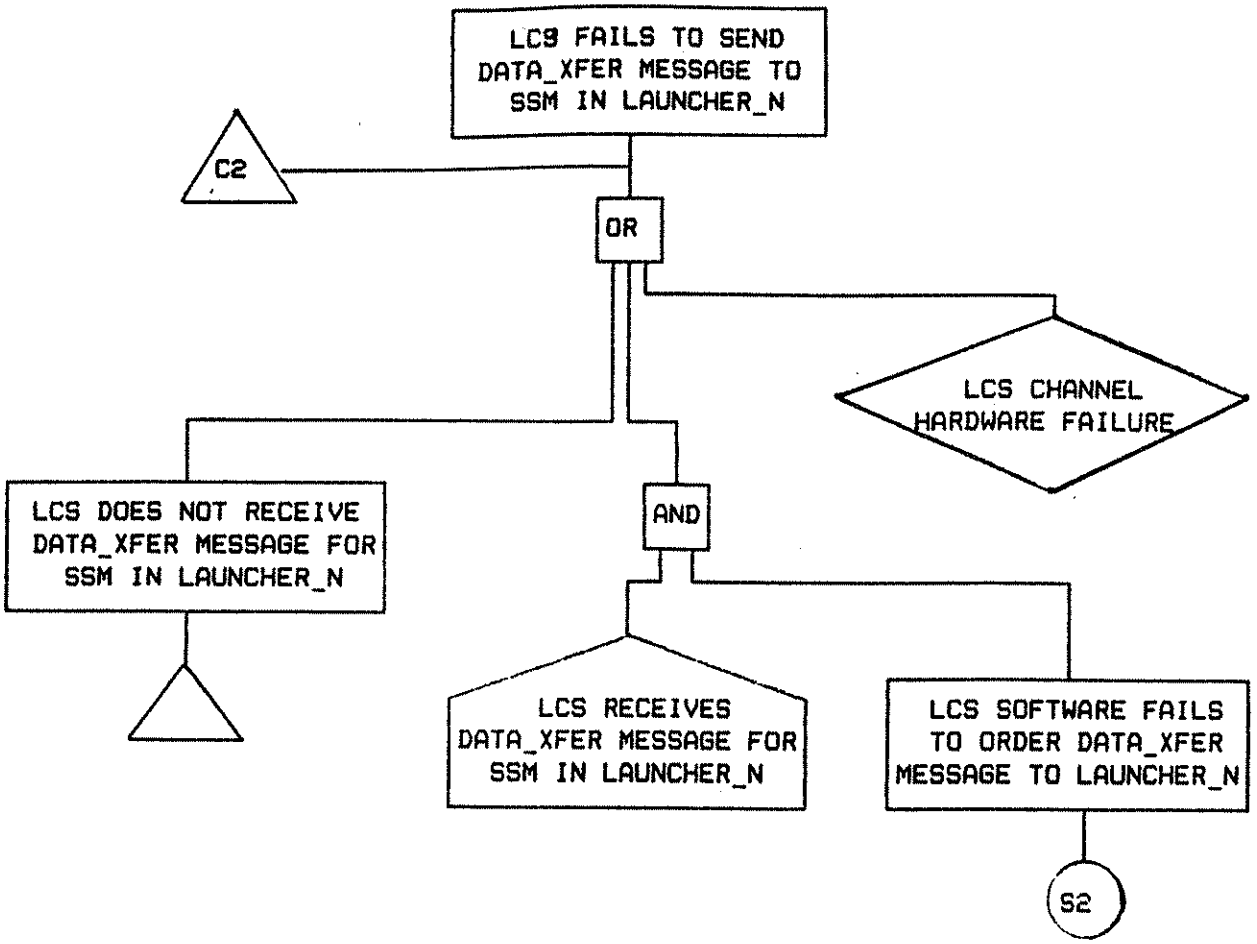

Figure 14. Component Fault Tree for 'LCS Fails to Send DATA_XFER Message to SSM in LAUNCHER_N.

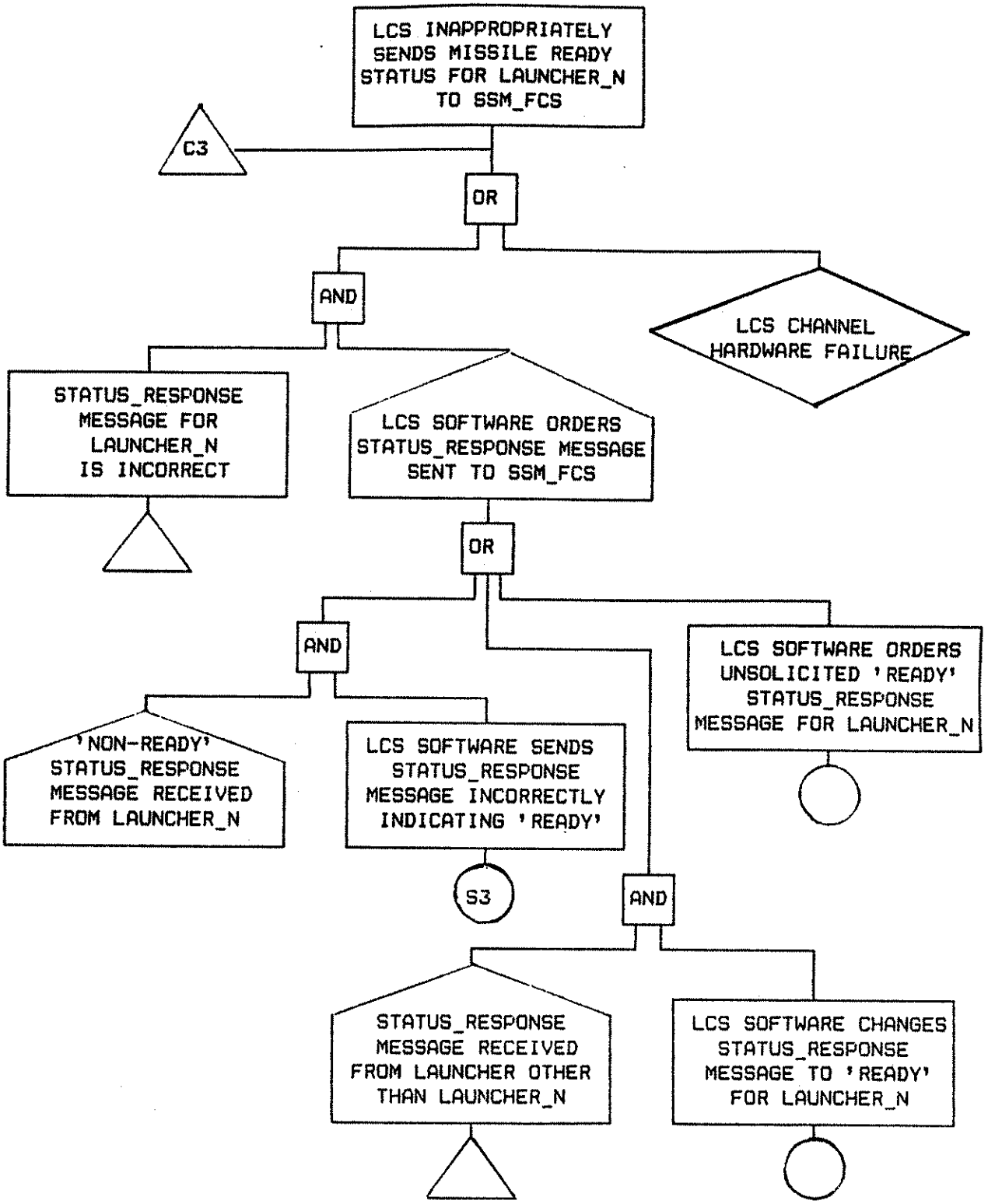

Figure 15. Component Fault Tree for 'LCS Inappropriately Sends Missile READY Status for LAUNCHER_N to SSM FCS.

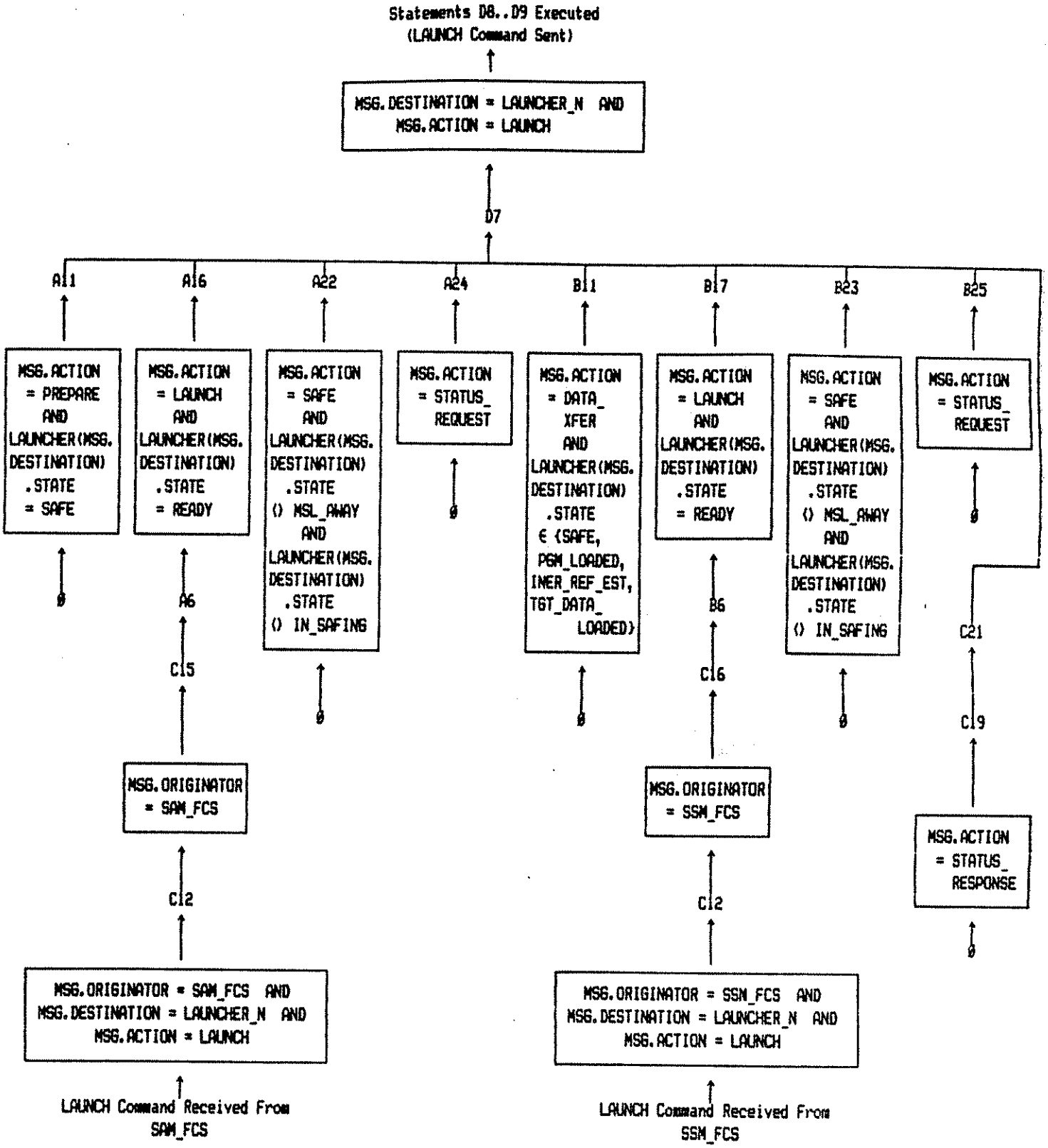

Figure 16. BFG for the safety-critical event, 'LCS sends LAUNCH command to LAUNCHER_N', corresponding to software fault event S1.

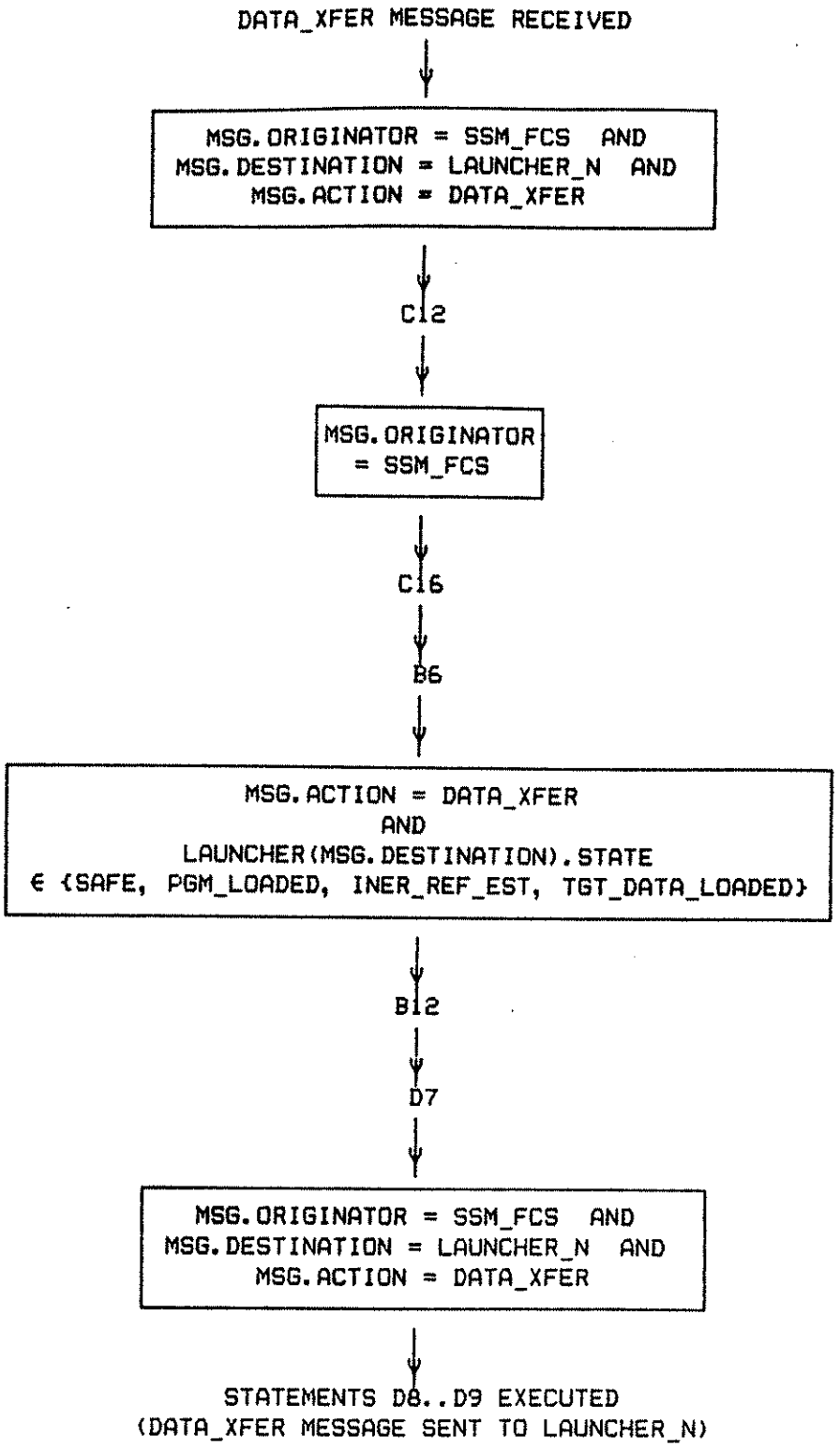

Figure 17. FFG for the safety-critical event, 'DATA_XFER message sent to LAUNCHER_N', corresponding to software fault event S2.

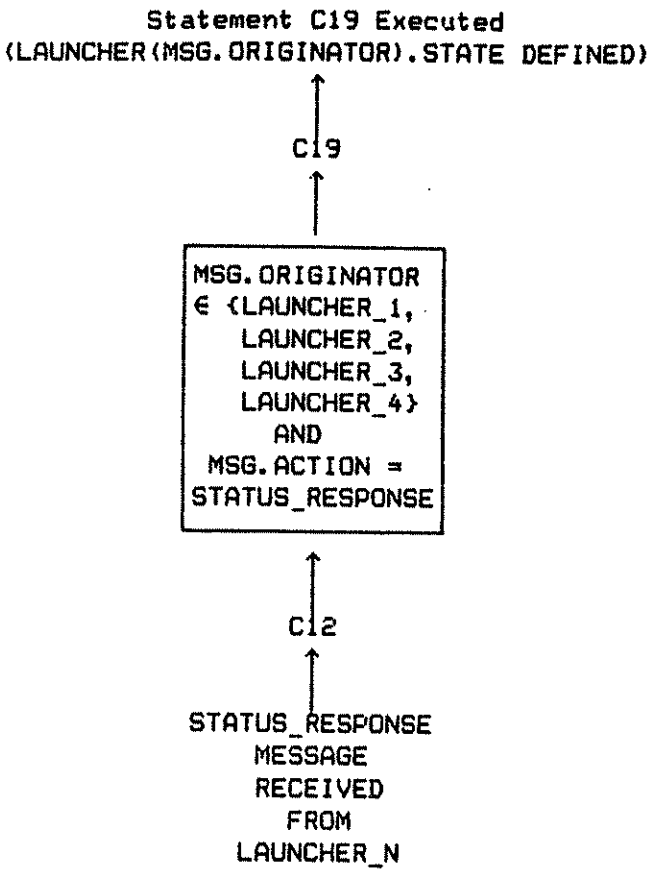

Figure 18. BFG for the event 'State of LAUNCHER_N defined', corresponding to the software fault event S3.

6. DISCUSSION

Ensuring distributed system safety certainly requires more than the application of a single technique. In order to provide for an adequate level of safety in the systems which it develops, the Department of Defense has established standard requirements for system safety programs. As defined in [2], Military Standard: System Safety Program Requirements(MIL-STD-882B), any DoD system safety program involves three major activities:

1. Define the safety program required for the system;
2. Carry out the tasks specified by the system's safety program; and
3. Feed back the results of the system safety tasks to the system's engineering change process.

The first activity above involves choosing specific safety tasks and tailoring the system safety program to the system being developed. The second activity encompasses the on-going execution of the tasks within the system safety program. The third activity couples the system safety program with the rest of the system development process by providing a mechanism by which safety-related changes can be introduced.

To be effective, the system safety program must be developed in consonance with the system life-cycle. Appendix B of [2] describes the relationship between the various system safety tasks and the phases of a system life-cycle. The safety tasks described in [2] fall into three groups: Management(100-series), System(200-series), and Software(300-series). The scheduling of tasks from these groups is system-dependent and is incorporated in the system safety program plan.

The steps in conducting CMFA may be spread across several of the tasks described in [2]. Although we have, as yet, no definitive experience in using the CMFA approach with a MIL-STD-882B system safety program, an initial matching of CMFA steps to MIL-STD-882B tasks is provided in Table 9.

Table 9. Correspondence Between CMFA Steps and MIL-STD-882B Tasks

CMFA Step	MIL-STD-882B Task
Steps 1,2,3	Preliminary Hazard List (Task 201) Preliminary Hazard Analysis (Task 202) System Hazard Analysis (Task 204)
Step 4	Subsystem Hazard Analysis (Task 203) System Hazard Analysis (Task 204) Software Requirements Hazard Analysis (Task 301)
Step 5,6	Top-Level Design Hazard Analysis (Task 302) Detailed Design Hazard Analysis (Task 303) Code-Level Software Hazard Analysis (Task 304)

Additionally, the component safety requirements developed during CMFA step 5 may provide a systematic basis for Software Safety Testing (Task 305 in [2]).

7. CONCLUSIONS

The CMFA approach, by itself, is not sufficient for ensuring the safety of a critical distributed system, but would be a valuable addition to a total system safety program. Several other techniques have been examined, including software fault tree analysis [4,7], concurrent program synchronization analysis [8], and Petri net modeling [6]. Although each of these techniques may provide meaningful results, only CMFA deals explicitly with a distributed system, message passing and three distinct categories of software fault events. The method proposed in this paper is intended to provide an analyst with a straightforward procedure for analyzing the safety of distributed system software. Whether this is true in practice waits to be seen. The CMFA approach will be used on a defense system safety certification task and feedback from analysts should be forthcoming.

8. REFERENCES

[1] NAVORD, OD 44942, Chapter 7, Hazard Analysis Techniques. U.S. Navy, Government Printing Office, Washington, DC.

[2] MIL-STD-882B (Notice 1), Military Standard: System Safety Program Requirements, 30 March 1984, Department of Defense, Washington, DC.

[3] Borning, A., "Computer Systems Reliability and Nuclear War," COMM ACM, Vol 30, 2(Feb 1987), 112-131.

[4] Leveson, N.B., and Harvey, P.R., "Analyzing Software Safety," IEEE TSE SE-9, 5(Sept 1983), 569-579.

[5] Leveson, N.B., "Software Safety in Computer-Controlled Systems," Computer, Feb 1984, 48-55.

[6] Leveson, N.B., and Stolzy, J.L., "Safety Analysis Using Petri Nets," IEEE TSE SE-13, 3(March 1987), 386-397.

[7] Leveson, N.G., "Software Safety: Why, What and How," ACM Computing Surveys, 18, 2(June 1986), 125-163.

[8] Taylor, R.N., "A General Purpose Algorithm for Analyzing Concurrent Programs," COMM ACM, 26, 5(May 1983), 362-376.

[9] Reference Manual for the ADA Programming Language, Washington, DC: U.S. Dep. Defense, 1983.