Redundancy and Fault Trees

Aaron Greenhouse

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM22-0842

General Problem

- Generate a fault tree from AADL with EMV2 annex clauses
  - Some prior work on this—None of it considers operational modes
    - (Modes ruin everything)
  - Want to explicitly consider
    - Operational modes/System reconfiguration
    - Limitations of AADL and EMV2 semantics
      - Missing semantics—What is under specified in the standards?
      - Missing semantics—What is not expressible currently?
  - Is a fault tree the right way to understand the behavior?
    - Starting with it because they are well know/understood
      - Well known techniques to analyze
    - But do they properly/naturally/easily represent what we want?

Branches of Research

1. Representing and Describing Redundancy

2. Generating Fault Trees

Redundancy in Systems

- Examples of systems that reconfigure generally use redundancy
  - Reconfiguration occurs when a component fails
  - New configuration “works around” failure
  - Eventually reconfiguration is not possible: System fails
- Looked at a number of examples from the DFT literature
  - Hypothetical Computer System (HECS)—multiple redundant subsystems
  - Dual redundant system
- Became clear that architecture models do not contain enough information to understand redundancy
  - Forget about building a fault tree, how do we even build the model?

Expressing Redundancy

- There is a need to express designer intent about redundancy within the model.
  - Some prior work on this [Baklouti]
- Need to know, e.g.,
  - Which set of components are intended to be redundant?
  - What redundancy pattern do they use?
    - K of N?
      - What is k?
    - Spares?
      - Which components are spares?
      - Hot/cold/warm?
      - What is the ordering of spares? Pooled, etc.?

Capturing Redundancy Intent

- Can be captured using AADL property associations
  - [Baklouti] uses a SysML/UML profile
- Advantages of capturing the redundancy intent
  - Enables better fault tree generation (e.g., spare gates)
    - Only thing that [Baklouti] proposes to use the profile for
  - Forms a *design pattern*:
    - AADL and EMV2 can be partially generated from the intent
      - Component modes and mode transitions
      - Error states and state transitions
      - Error detection clauses
    - Conversely, pattern can *enforced* as the model *evolves*
      - Check that specific modes and transitions exist
      - Check that specific states and state transitions exist
      - Etc.

```
system Bus_Subsystem
  features
    bus1: provides bus access B.i;
    bus2: provides bus access B.i;
  end Bus_Subsystem;

system implementation Bus_Subsystem.i
  subcomponents
    b1: bus B.i;
    b2: bus B.i;
  connections
    pb1: bus access b1 <-> bus1;
    pb2: bus access b2 <-> bus2;
  properties
    Redundancy_Properties_1::Redundancies => ([
      -- Straightforward duplication; Need at least 1 operating out of a total of
2
      pattern => Active;
      k => 1; -- Should fail when (2-1)+1 = 2 buses have failed
      n => 2;
      components => ([ref=>reference(b1);], [ref=>reference(b2);]);
    ]);
end Bus_Subsystem.i;
```

K of N redundancy — 1 of 2 needed

Early idea of what needs to be captured.

Have very basic ideas of implementation patterns.

```
system implementation Processor_Subsystem.i
  subcomponents
    a1: processor P_Normal.i;
    a2: processor P_Normal.i;
    --COLD SPARE: activated when a1 or a2 fa
    a_spare: processor P_ColdSpare.i;
    monitor: device Processor_Monitor.i;
  connections
    -- . . .
  properties
    Redundancy_Properties_1::Redundancies => (
      [pattern => Standby;
       spare => Cold;
       components => ([ref=>reference(a1); rank=>1;], [ref=>reference(a_spare); rank=>2;]);
       switches => ([ref=>reference(monitor); src=>reference(a1); dst=>reference(a_spare);]);
      ],
      [pattern => Standby;
       spare => Cold;
       components => ([ref=>reference(a2); rank=>1;], [ref=>reference(a_spare); rank=>2;]);
       switches => ([ref=>reference(monitor); src=>reference(a2); dst=>reference(a_spare);]);
      ]);
end Processor_Subsystem.i;
```

- Component **a_spare** is a spare for both **a1** and **a2**
- Component **monitor** determines when **a1** fails and switches to **a_spare**
- Component **monitor** determines when **a2** fails and switches to **a_spare**

Fault Trees

- How to derive a fault tree from AADL and EMV2
  - In a modular way: Respecting component boundaries
  - While considering modes and error states

- Currently inspired by Component Fault Trees [Kaiser, et al.]
  - Can be generated per component description (AADL component classifer)
  - Without needing to know specifics of subcomponent behaviors
  - Complete set of component fault trees can be assembled into a fault tree following an instantiation approach

Layers of Components

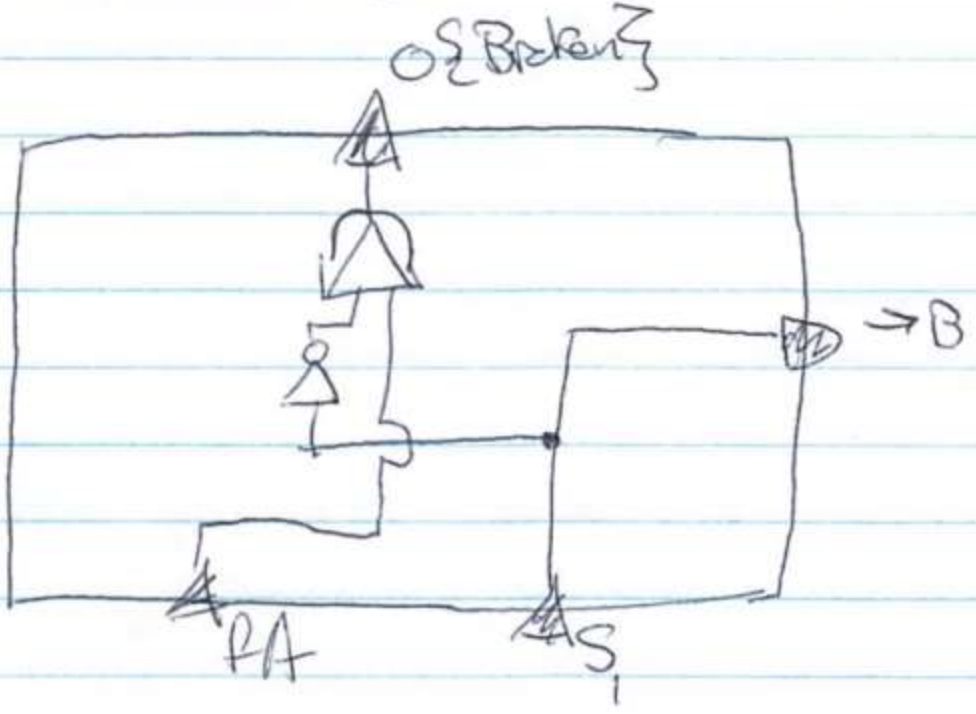
- Component Fault Tree for an AADL Classifier is build from layers:
  - Component Fault Tree
    - Modal Component Fault Trees
      - Stateful Component Fault Trees
        - Subcomponent Component Fault Trees

  - Model is considered projected into specific modes and behavior states

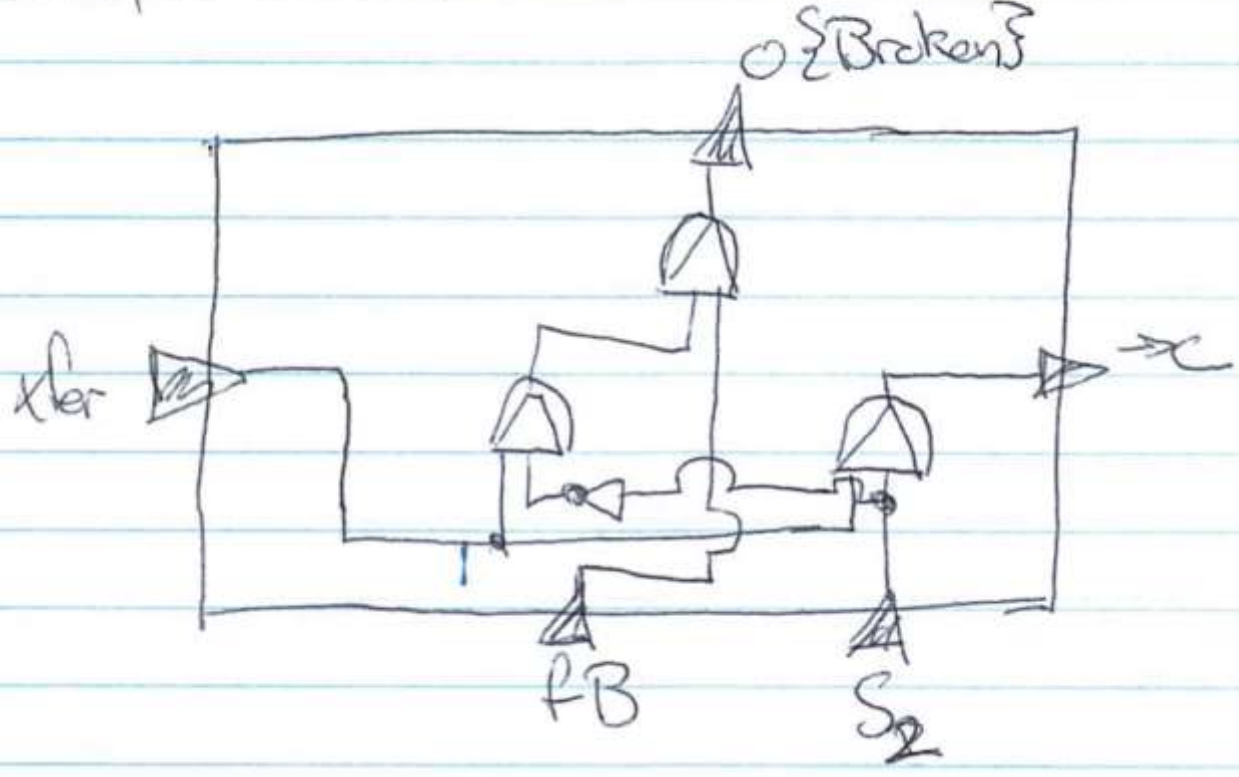
```
package ErrLib
Public
  annex EMV2 {**
    error types
      I_Broke: type;
    end types;

    error behavior Simple
    events
      Switch: error event;
      FailInA: error event;
      FailInB: error event;
      FailInC: error event;
    states
      A: initial state;
      B: state;
      C: state;
    transitions
      t1: A -[Switch]-> B;
      t2: B -[Switch]-> C;
    end behavior;
  **};
end ErrLib;
```

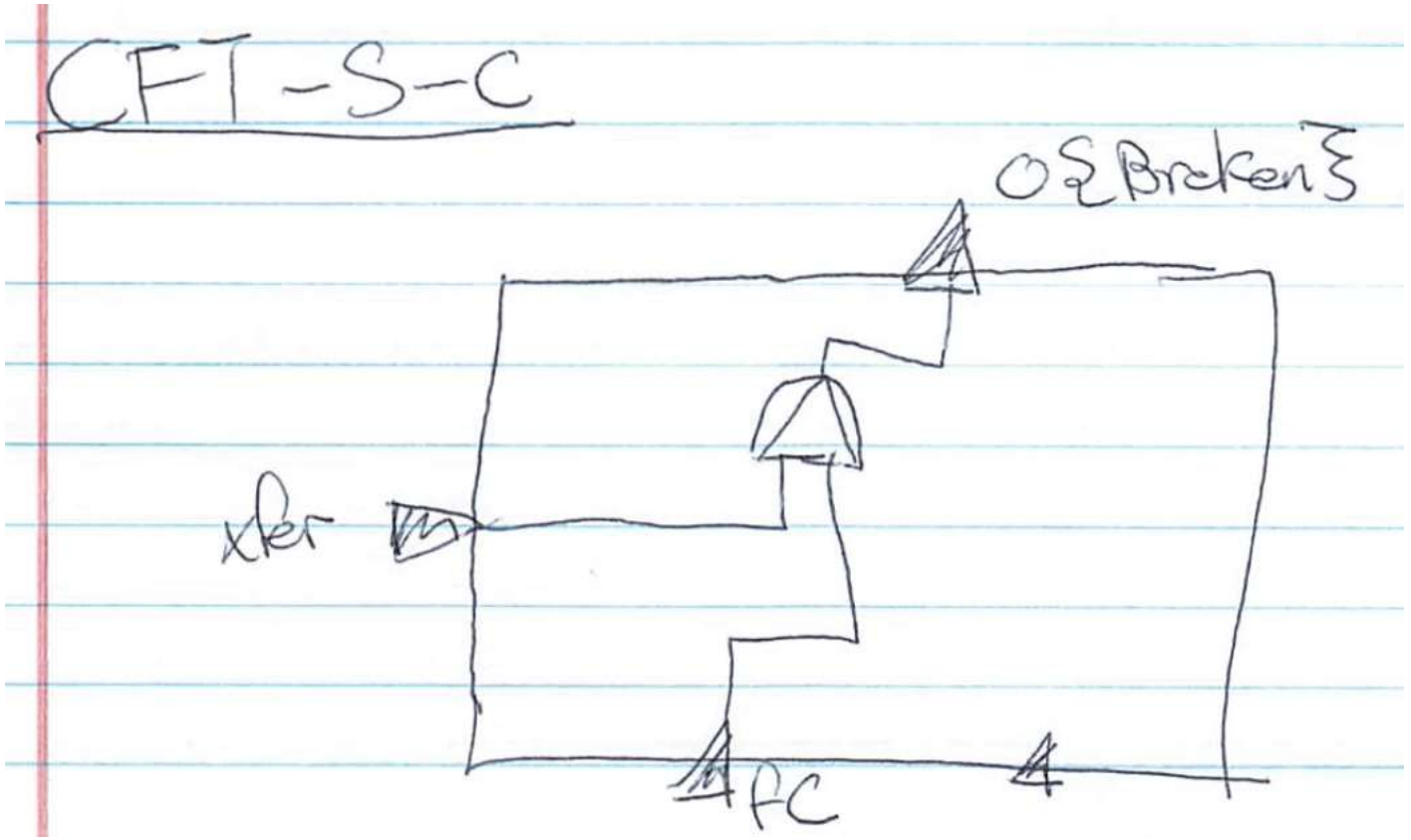
```
package Test
public
  system S
    features
      output: out event port;

    annex EMV2 {**
      use behavior ErrLib::Simple;
      error propagations
        output: out
propagation{ErrLib::I_Broke};
      end propagations;
      component error behavior
      propagations
        pA: A -[FailInA]->
output{ErrLib::I_Broke};
        pB: B -[FailInB]->
output{ErrLib::I_Broke};
        pC: C -[FailInC]->
output{ErrLib::I_Broke};
      end component;
    **};
end S;
```

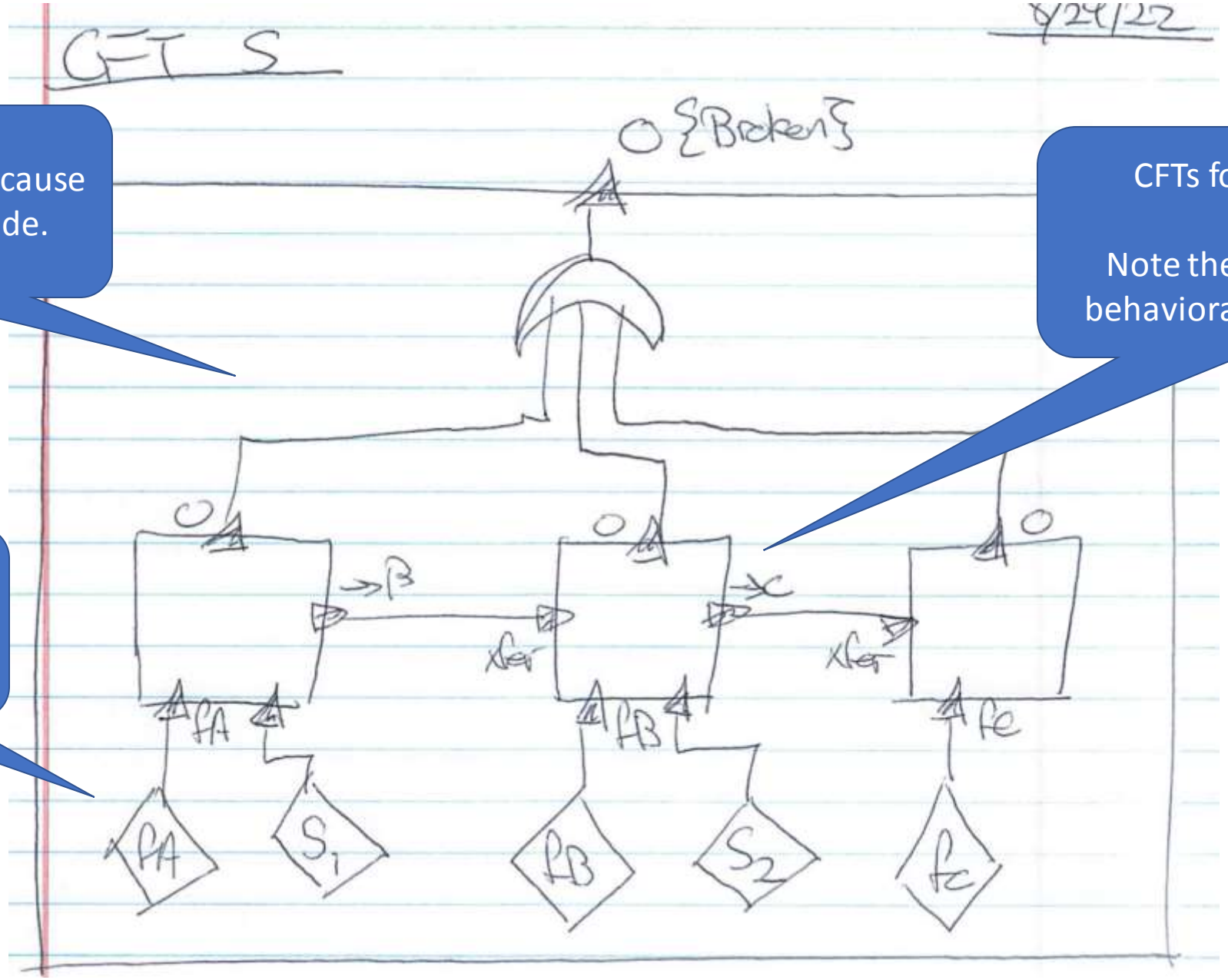
CFT - S-A



CFT - S-B







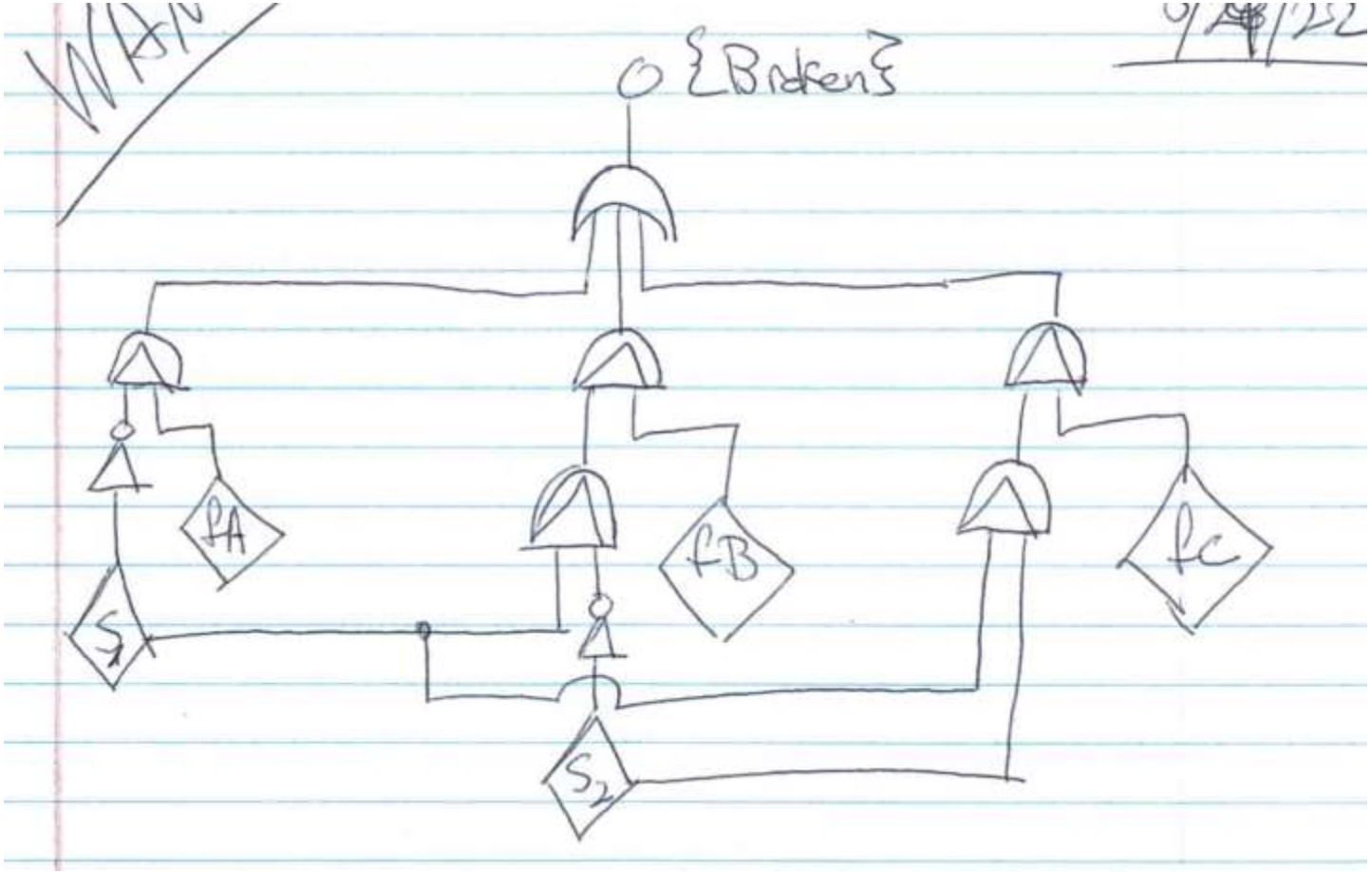
Missing modal layer because there is only one mode.

CFTs for each state.

Note they *replicate* the behavioral state machine.

Error events as FT basic events.

Final Fault Tree



Other Representations

- Resulting fault tree encodes the state machines for
  - Modes
  - Error states

- Perhaps the fault tree is the wrong abstraction to be using?
  - Trying to make it do too much?
  - Are there abstractions that better capture the relationships but still allow for analysis?