

# Behavioral Fault Modeling for Model-based Safety Analysis\*

Anjali Joshi  
University of Minnesota  
Minneapolis, U.S.A.  
ajoshi@cs.umn.edu

Mats P.E. Heimdahl  
University of Minnesota  
Minneapolis, U.S.A.  
heimdahl@cs.umn.edu

## Abstract

*Recent work in the area of Model-based Safety Analysis has demonstrated key advantages of this methodology over traditional approaches, for example, the capability of automatic generation of safety artifacts. Since safety analysis requires knowledge of the component faults and failure modes, one also needs to formalize and incorporate the system fault behavior into the nominal system model. Fault behaviors typically tend to be quite varied and complex, and incorporating them directly into the nominal system model can clutter it severely. This manual process is error-prone and also makes model evolution difficult. These issues can be resolved by separating the fault behavior from the nominal system model in the form of a “fault model”, and providing a mechanism for automatically combining the two for analysis. Towards implementing this approach we identify key requirements for a flexible behavioral fault modeling notation. We formalize it as a domain-specific language based on Lustre, a textual synchronous dataflow language. The fault modeling extensions are designed to be amenable for automatic composition into the nominal system model.*

## 1 Introduction

Model-based safety analysis (MBSA) [8, 9, 3, 2, 15], where the safety analysis is based on a central formal model of the system, has been proposed to address some of the issues arising due to the manual, informal, and error prone nature of the traditional safety analysis process. Some of the advantages of this approach include automation of parts of the safety analysis process (e.g., auto-generation of fault trees [4, 10, 17, 18]), providing consistent analyses, and most importantly, tightly integrating the systems and safety engineering processes around a central system model.

Since safety analysis is performed in the context of the entire system, MBSA requires modeling of the physical (hardware and mechanical) components in addition to the digital components that are typically modeled as part of model-based development. The focus of the system safety analysis is on analyzing the safety requirements in presence of component faults. Thus, the MBSA approaches need to support some notion of modeling of fault behaviors in addition to the nominal (non-failure) system behaviors. Fault behaviors, however, typically tend to be quite varied and complex even for simple system components. Consider, for example, a simple mechanical valve whose nominal behavior is to regulate the outgoing pressure based on a position command. Even with such a simple nominal behavior the fault behaviors of this valve can be quite varied. They can include fault behaviors such as the valve getting stuck open, closed, or at some partial opening position. Numerous other failure modes may also be present depending on the manufacturer’s failure specification of the actual physical valve. More involved but realistic fault behaviors, such as error propagations and conditional fault activations, add even more complexity to the overall system fault behavior.

One approach to modeling these fault behaviors is to specify them using the system modeling notation itself, and incorporate them directly into the nominal system model. Unfortunately, directly adding such complex fault behaviors into the system model tend to severely clutter the model with failure information. This added complexity typically obscures the actual non-failure system functionality making model creation, development, inspection, and maintenance difficult. In the absence of tool-support, the incorporation of the fault behaviors is performed manually, leading to error-prone extension of the model with fault behavior.

To address these issues, we believe it is crucial to have the ability to separate the fault behavior from the nominal system model in the form of a “fault model”, and at the same time provide a mechanism for combining the two to perform meaningful safety analysis. In addition, having a notation that is specifically targeted for fault modeling will promote ease of specification of complex fault behaviors,

---

\*This project was partially funded by the NASA Langley Research Center under contract NCC1-01001 of the Aviation Safety Program and NASA Ames Research Center Cooperative Agreement NNA06CB21A.

such as error propagations and fault hierarchies, allowing the engineer to create simple but realistic models for precise safety analysis. In this paper, we identify the key requirements for flexible behavioral fault modeling. We propose a prototype implementation of these requirements as domain-specific fault modeling extensions to the synchronous dataflow language Lustre [7]. This notation, *LustreFM*, also enables automatic composition of the fault model into the nominal system model that can then be analyzed for safety. These language extensions can be easily mapped to other synchronous dataflow languages with minor modifications.

The rest of the paper is organized as follows. In Section 2, we motivate the problem and identify the requirements on flexible fault modeling. In Section 3, we enumerate the key behavioral fault modeling requirements based on which we propose the domain-specific *LustreFM* notation in Section 4. Section 5 provides a brief discussion of the ongoing work. Section 6 discusses some of the related work in the area of fault modeling, which is followed by a conclusion.

## 2 Behavioral Fault Modeling Illustration in Simulink

We motivate the behavioral fault modeling requirements with the help of an aircraft Wheel Brake System example. We model this example using Simulink [12], a graphical synchronous dataflow language commonly used for modeling digital control systems in the safety-critical systems domain. By modeling both the nominal component as well as the fault behaviors in Simulink, we expose some of the issues with using existing system modeling notations for fault modeling.

**Wheel Brake System:** The Wheel Brake System (WBS), as described in ARP 4761 [20], provides braking pressure to the aircraft wheels. We chose this example primarily because the ARP 4761 document is used as the main reference for safety assessment by the majority of the safety engineers in the avionics community. The WBS consists of a digital controller, the Braking System Control Unit (BSCU), and the hydraulic pipe assembly that carries the braking pressure to the wheels. Based on the safety requirement that loss of all wheel braking shall be less probable than  $5 \cdot 10^{-7}$  per flight, a design decision was made that each wheel has a brake assembly operated by two independent sets of hydraulic pistons. One set is operated from the *Green* pump and is used in the *normal* braking mode. The *alternate* braking system is on standby and is selected automatically when the normal system fails. Switch-over between the hydraulic pistons and the different pumps is automatic under various failure conditions, or can be manually selected.

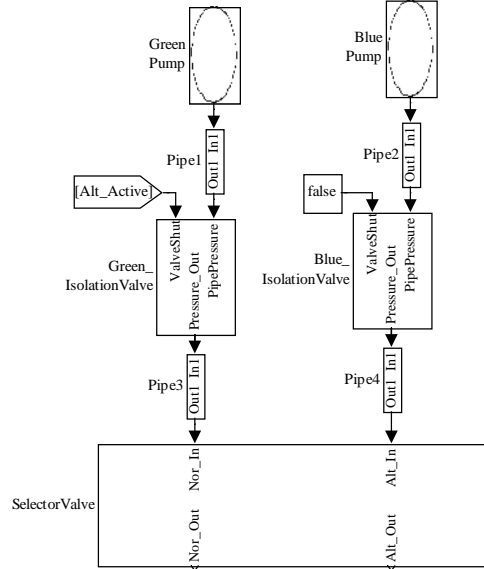


Figure 1. Part of the Wheel Brake System

We use only a small part of the WBS consisting of the hydraulic pumps, the isolation valves (used to isolate one of the hydraulic lines in case of a failure), and the connecting pipes (as shown in Figure 1) to illustrate realistic behavioral fault modeling in Simulink.

### 2.1 Modeling Internal Faults

Internal faults<sup>1</sup> are intrinsic to a component and originate from within the component boundary. From the behavioral modeling point of view, the internal fault behavior lies encoded in the component itself, representing a dormant fault. Activation of this fault leads to a component error or failure<sup>2</sup>. Internal fault activation occurs *independent* of other component failures, and can be modeled using an independent system input or a triggering condition.

As an example, consider the nominal behavior of a simple isolation valve as shown in Figure 2. There are two instances of this valve, *Green\_IsolationValve* and *Blue\_IsolationValve* in the WBS. It has two inputs; *ValveShut* is a boolean input that controls whether the isolation valve is open (*False*) or shut (*True*), and *PipePressure* that captures the pressure on the incoming pipe connected to the valve. The output *Pressure\_Out* models the regulated pressure that goes out on the outgoing pipe. The nominal behavior of the valve can then be captured with a simple Simulink *Switch* block – if the middle boolean input (*ValveShut*) is true, then

<sup>1</sup>For terminology details we refer the reader to [1].

<sup>2</sup>This is always true in the case of synchronous dataflow models, as each component executes every time step irrespective of whether its output gets used elsewhere or not, which may or may not correspond to reality.

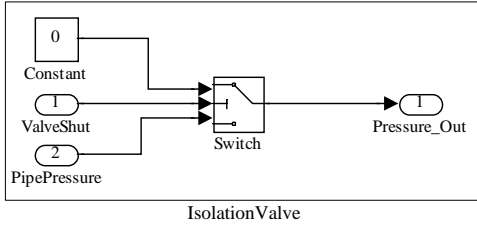


Figure 2. Nominal Isolation Valve in Simulink

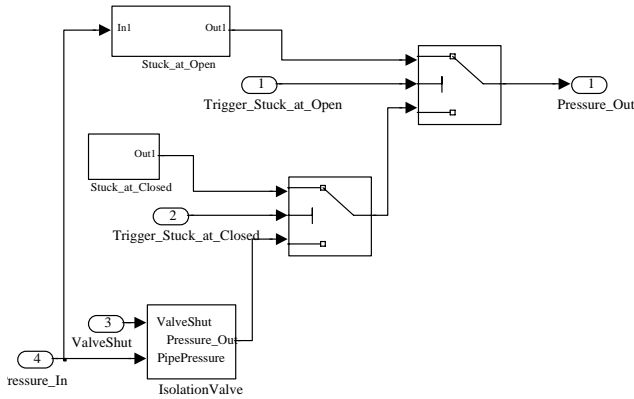


Figure 3. Extended Isolation Valve

the top input of the switch block (a constant value of zero) is forwarded to the output, otherwise the bottom input (incoming `PipePressure`) gets forwarded.

We can now define two internal faults on this valve; `Stuck_at_Open` (the outgoing pressure is the same as the incoming pressure) and `Stuck_at_Closed` (the outgoing pressure is zero). These faults are captured by simple Simulink blocks by the same name. We can now extend the nominal behavior of the isolation valve with these two fault behaviors. We need two independent fault activation triggers for activating these two independent faults, call them `Trigger_Stuck_at_Closed` and `Trigger_Stuck_at_Open`.

Another basic consideration for fault modeling is the duration of the fault. One can define a *permanent* fault by latching the trigger inputs once activated. In case of *transient* faults, we want the activation triggers to be non-deterministic, in which case, the triggers are plain inputs.

Since we are defining two fault behaviors that affect the same output of the component, there is potential for a *conflict* between them when both faults are activated at the same time. We need to define *priorities* to break such a conflict. In this case, we define `Stuck_at_Open` with a higher priority than `Stuck_at_Closed` for the isolation valve. We define an extended component that now includes the nominal component behavior with the added fault behavior taking into account the priorities as shown in Figure 3.

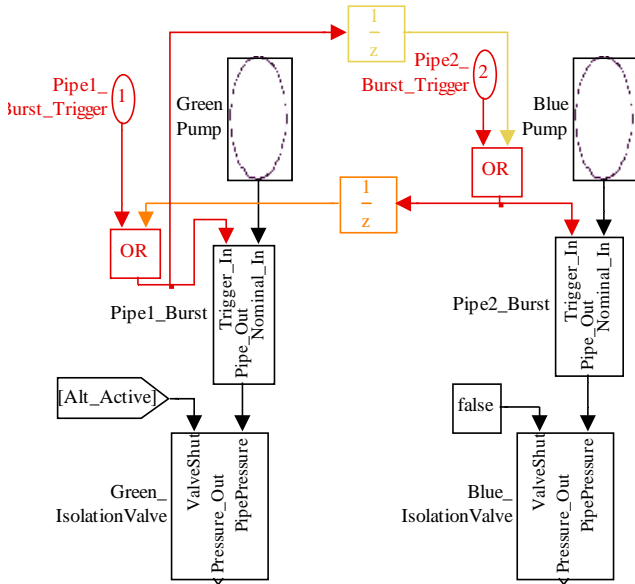
**Basic Modeling Issues:** Since extending the nominal component behaviors with the fault behaviors also extends the component interfaces, the additional signals have to be wired correctly. In Simulink, if the extended component lies deeply embedded in the system, the newly added fault activation trigger inputs have to be routed all the way up to the topmost system level. If we decide to add new fault behaviors, or redefine the priorities, we have to manually identify the affected components and make the required modifications. Also, the priority decisions lie implicit in the way the fault behaviors are composed, without being explicitly stated.

## 2.2 Modeling External Faults

The faults that get activated by interaction or interference due to error propagation from outside the component boundary are considered as External Faults. For example, a power supply failure that propagates an error to all the digital components powered by that supply becomes an external fault for those digital components. An external fault typically necessitates the prior presence of a *vulnerability* (i.e., an internal fault) that enables an external error propagation to harm the component. From the behavioral modeling point of view, we model a component vulnerability in the same manner as an the internal fault behavior. The external faults differ from the internal faults in their activation—their fault activation triggers are dependent on the error propagating components as opposed to being triggered alone (as with internal faults).

Consider the pipe assembly in Figure 1. We model the pipes as having the nominal behavior of simply forwarding the input pressure to the output. For simplicity, we only define fault behaviors on the pipe components and not the valves. We model two types of fault behaviors for pipes; `Pipe_Burst` (the pipe is severed and the resulting pressure at the output is zero), and `Pipe_Leak` (the resulting pressure at the output is slightly lower than the incoming pressure). These behaviors model both internal faults and external fault vulnerabilities. We now consider two types of error propagations that can occur in our example.

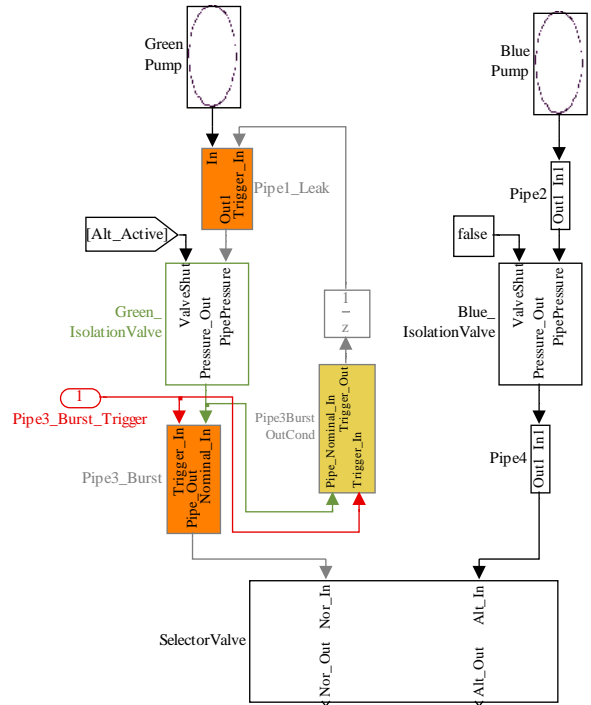
**Error Propagation in Unconnected Components:** In our example, the *normal* (powered by Green pump) and the *alternate* (powered by Blue pump) sets of hydraulic pipes are independent; i.e., there is no dataflow between them. In our Simulink model (Figure 1), we have only modeled the *logical* architecture of this pipe assembly, and it does not encode any information about the actual *physical* architecture. If in the physical layout the two redundant sets of pipes are routed very close to each other, a failure of one of the pipes (say, a pipe burst) can affect the pipe that is routed close to it. Thus, though `Pipe1` and `Pipe2` do not



**Figure 4. External Fault: Propagation from an Unconnected Component**

seem to affect each other logically, there is indeed physical interference between them and a rupture of one pipe can lead to the rupture of the other pipe. To capture this dependency in the fault behaviors, we add explicit error propagation paths between these two pipes, (Figure 4). Note that the explicit error propagations that are captured by routing the trigger of one of the pipes to the other causes a *cyclic dependency* between the two components (called a semantic loop in Simulink). Synchronous dataflow languages, including Simulink, do not allow such a cyclic dependency, and it can be broken by adding a unit delay ( $\frac{1}{z}$  block in Simulink).

**Error Propagation in Reverse Dataflow Direction:** In our pipe assembly example, a failure of some pipe will affect all the connected pipes in the assembly, provided the connecting valve is not closed. Due to the data-flow in the underlying architecture, the effect of the failure of a pipe will get propagated downstream; the error propagation downstream is implicitly captured in the model as it is in the same direction as the dataflow. Thus, Pipe\_Burst of Pipe1, will lead to no pressure in Pipe3. Observe, however, that in our simple nominal pipe model, there is no mechanism to propagate errors upstream. Though a Pipe\_Burst failure of Pipe3 will lead to a loss of pressure in Pipe1 when the connecting isolation valve is not closed, there is no such data dependency in the nominal model. Such an error needs to be propagated explicitly as an external fault in Pipe1 in the direction opposite to that of the dataflow (see Figure 5). We have replaced



**Figure 5. External Fault: Back Propagation**

the Pipe3 component with an extended Pipe3\_Burst component, that embeds the fault behavior Pipe\_Burst, which can be activated by an independent trigger input, Pipe3\_Burst\_Trigger.

Note that the occurrence of a backward error propagation from Pipe3 to Pipe1 depends on satisfaction of a *condition: the isolation valve is not closed*. In Figure 5, the condition is captured in a separate Simulink block Pipe3BurstOutCond. This block checks if the Pressure\_Out value of the isolation valve is greater than zero (i.e., the valve is open), and if so, sets its boolean error propagating output, Trigger\_Out, to true. Here again, we need to add a delay block to break the cyclic dependency created due to the backward error propagation. Note also that the target fault behavior for Pipe1 is not Pipe\_Burst but Pipe\_Leak to model the fall and not complete loss of pressure in Pipe1.

**Error Propagation Modeling Issues:** In reality, there are a number of possible error propagation paths between various system components. One has to model all these error propagations, in addition to the component internal fault behaviors, the activation and error propagation conditions, etc., which will dramatically increase the complexity and clutter in the extended model. Note that adding such explicit error propagations also blurs the distinction between the actual system architecture and the additional fault be-

havior in the extended model.

## 2.3 Issues with Direct Incorporation of Fault Behaviors in System Models

As can be seen from the examples above, there are issues and shortcomings with using the existing modeling notations, such as Simulink, for fault modeling and incorporating the fault behaviors directly into the nominal system model.

**Model Complexity and Clutter:** As one can observe from these simple examples, the system fault behavior can get quite complex. Directly composing the fault behaviors in the system model tends to severely clutter and complicate the system model. The added connections for modeling error propagations between different components hinder the visibility of the underlying system architecture.

**Manual Extension:** Manually extending the nominal behavior with the fault behavior is error-prone, cumbersome and leads to model evolution issues. As the system evolves, when changes to the nominal or the fault behavior are required, one has to make these changes in a cluttered model requiring a redo of a lot of modeling effort, which is highly undesirable.

**Lack of Language Support for Fault Modeling:** Since there is no good inherent support for fault modeling in the synchronous dataflow language domain, one has to perform the fault modeling activity with a system modeling mindset. The modeler has to deal with issues that arise when fault modeling, for example, wiring of new signals, manually routing the explicit error propagation paths when modeling external faults, composing multiple fault behaviors with the right priorities taken into account, latching the fault activation triggers inputs based on the duration, etc.

## 3 Behavioral Fault Modeling

Most of the issues highlighted in the previous section can be resolved if we specify a separate *fault model* using a domain-specific notation that is more suitable for representing flexible fault modeling ideas, for example, the issues of fault behavior specifications and their associations to nominal components, error propagations, permanent and transient faults, conflict resolution, etc. identified in the previous section. While the model-based development that model-based safety analysis extends uses the nominal system model for driving a variety of activities, such as code-generation, test-case generation, etc., the fault model is used solely for the purpose of safety analysis. Keeping the fault and system models separate helps the separate evolution and usage of the two models.

We can address the manual extension issue by making the notation amenable to composition using an *automatic*

*composition tool*, which will take care of all composition and wiring issues and generate an extended model with the required fault behavior inserted in the right places. This will now eliminate the error-prone manual extension required otherwise.

Based on our experiences in model-based development and safety analysis we have defined a set of requirements on such a *Behavioral Fault Modeling Language*.

### Component Fault Behavior

The notation must enable the engineer to specify component fault behaviors for both internal faults and vulnerabilities to external faults.

#### Associations to the Nominal Component

(1) *Explicit Associations:* Since the system fault model is defined separate from the nominal model, the notation must enable specifying explicit associations between the relevant fault behaviors and the nominal components.

(2) *Multiple Associations:* Since a component can fail in more ways than one, the notation must enable associations of more than one fault behavior to a particular component.

(3) *Conflict Resolution:* A *conflict* may occur between the multiple fault behaviors (multiple internal fault behaviors or vulnerabilities) associated with a single component. These conflicts must to be resolved by defining some form of priorities or user-defined strategies.

(4) *Nominal Component Types:* For flexible associations, a notion of component *types* must be supported. The user can specify component types to group together nominal components that have similar nominal or fault behaviors for the purpose of easy associations.

### Fault Activation

(1) *Trigger and Persistence/Duration:* The language shall support the trigger and persistence specification for both internal and external faults. It shall also support the specification of conditional fault activation, where the trigger and the persistence will be controlled by the condition.

(2) *Error Propagation Rules:* For identifying and activating the external faults, in addition to specifying the vulnerability behaviors, the notation shall also support specification of error propagation rules – i.e., mappings between the vulnerabilities and the corresponding error propagating behaviors.

### Fault Model Hierarchies

For more flexibility, the engineer must be able to successively specialize fault behavior definitions as the design of the system and fault model progresses. For example, one can define a generic *stuck\_at* failure mode for different types of valves, which can later be specialized for different valves as *stuck\_at\_Open*, *stuck\_at\_Closed*, etc.

## 4 Fault Modeling Extensions to Lustre: LustreFM

For performing the fault modeling language extensions, we have chosen a textual synchronous dataflow language, Lustre [7]. Lustre shares the same underlying semantic model as languages such as Simulink and SCADE (it is the underlying textual notation to the graphical SCADE tool). We preferred a textual language to a graphical notation as the graphical notations add more complexity to the extension definitions and composition without providing any significant conceptual advantages. Another reason for choosing Lustre over other textual notations is that we have a well established translator framework which uses Lustre as an intermediate language to translate from graphical languages, such as Simulink, into input languages of formal analysis tools, such as NuSMV [13], and PVS [16]. Thus, the system may be originally defined in SCADE or Simulink, which will then be transformed into an intermediate *Lustre nominal system* model.

**Lustre:** Lustre [7] is a synchronous dataflow language, where the behavior of a Lustre program is derived from a set of *equations* that assign *variables*. Lustre assignments to variables are always *functional*, meaning that there is no possible nondeterminism in the assigned variables. Each variable is a function of time: any variable or expression denotes a flow, i.e., an infinite sequence of values of its type. A Lustre program is effectively an infinite loop, and each variable or expression takes the  $k^{th}$  value of its sequence at the  $k^{th}$  step in the loop. Boolean, arithmetic, comparison, conditional operators are implicitly extended to operate pointwise on flows. Lustre also supports *clocks*, which allow different portions of a specification to run at different rates. Lustre definitions can be recursive, but the language requires that a variable can only depend a past values of itself. A Lustre program may not contain syntactically cyclic definitions. Lustre programs are organized into *nodes*, which package equations into modules that can be reused within a program. A node consists of an interface specification (the input and output parameters) and a body, which declares local variables and defines the assignment equations for local and output variables. We can specify the nominal behavior of the isolation valve (modeled as a Simulink subsystem instantiated twice in Figure 2) as a Lustre node:

```
node IsoValve (ValveShut: bool ; PipePressure: int)
  return (Pressure_Out: int);
let
  Pressure_Out = if (ValveShut) then 0
                 else (PipePressure);
tel;
```

### 4.1 Component Fault Behavior

Fault behaviors can be typically captured in terms of a regular Lustre nodes without any language extensions. To illustrate, let us revisit the definitions of the two simple failure modes we considered earlier, `Stuck_at_Open` and `Pipe_Burst`, in Lustre.

```
node Stuck_at_Open (Nominal_In : int)
  return Stuck_Out : int;
let
  Stuck_Out = Nominal_In;
tel;

node Pipe_Burst ()
  return Burst_out : int;
let
  Burst_Out = 0;
tel;
```

Note that the `Stuck_at_Open` fault behavior is in the form of a *wrapper* to the nominal component behavior as it uses the input to affect the output and bypasses the nominal behavior. The `Pipe_Burst` behavior on the other hand, directly affects the nominal output (in effect, after the generation of the output using the nominal behavior). Same holds in case of an *inverted* failure mode, where it inverts the nominal component boolean output.

Modeling certain fault behaviors might necessitate adding new inputs to the system that originally did not exist. Consider the example of having to model a non-deterministic fault behavior, that outputs a non-deterministic integer. Since non-determinism is not supported in Lustre, we can model this by taking a non-deterministic input. To capture this additional input, we make an extension to Lustre, new, as shown below.

```
node nondeterministic (new nondeter_In: int)
  return nondeter_Out : int ;
let
  nondeter_Out = nondeter_In;
tel;
```

### 4.2 Association with the Nominal Components

**Nominal Component Types:** Consider our running example as shown in Figure 1. Note that there are two instances of the `IsoValve` implementation, viz., `Green_IsolationValve` and `Blue_IsolationValve` in the model, but the information that these are implemented exactly the same is lost in the Lustre model. We enable specification of explicit nominal component types to group together components that have the same nominal implementation, and now the component types in addition to the specific component instances can be used for association. Part of the LustreFM grammar is given in Figure 6.

```
cType IsoValve = Green_IsolationValve,
          Blue_IsolationValve;
```

**Internal Fault Association:** For all internal faults, we must explicitly specify associations to the affected nominal components. In addition to identifying the affected nominal component/type, the association also includes (1) the affecting fault behavior(s), and (2) the fault activation information.

The fault behavior definitions *affect* nominal component variables (typically outputs), and may also *use* the nominal component variables (inputs, locals, or outputs); e.g., in the `Stuck_at_Open` fault behavior, the `Nominal_In` input corresponds to a particular input of the nominal component(s) that this fault behavior uses, and `Stuck_out` output corresponds to the output of the nominal component(s) that it affects. For the fault behavior association to be automatically composable, the correspondence between the fault behavior variables and the nominal variables must be explicitly specified.

Internal fault activation can be carried out based on either (1) an explicit, independent *boolean system-level input*, which captures the activation and persistence of the fault, or (2) a *triggering condition*, which when satisfied activates the fault and keeps it active for the duration it holds. In case of explicit boolean input trigger, the persistence can be defined to be either *permanent* (once triggered, permanently active) or *transient* (triggered for an unspecified duration, and can become dormant and active arbitrarily). An example of a triggering condition for internal faults is when a *value* failure occurs, where the component fails if the incoming value is not in a specified range. In this case, the fault activation of the corresponding internal fault is controlled solely by the condition that captures the violation of this range. Thus, an internal fault activation can be specified in terms of either the fault duration in case of the explicit input, or by identifying the triggering condition.

We can now capture an example association in the following manner-

```
fm_assoc StuckOpen:IsoValve = permanent {
  Pressure_Out = Stuck_at_Open(PipePressure);}
```

The above `fm_assoc` association definition has a name (`StuckOpen`), and refers to an affected nominal component/type (`IsoValve`). On the right-hand side is the body of the association that includes the duration (`permanent`) of the associated fault(s), and a list of Lustre style equations that correspond to node call expressions. The LHS of the equation is the *affected* output(s) of the nominal component (`Pressure_Out`), and the RHS of the equation represents a Lustre node call expression, with the node being a fault behavior node (`Stuck_at_Open`), and the parameters being the *used* nominal component variables (`PipePressure`) or constants.

**Vulnerability Behavior Association:** The association of the fault behavior corresponding to the vulnerability can be

```
cType ::= cType Nom_CompTypeId='Nom_InstId {'Nom_InstId}*';
Nom_CompId ::= Nom_CompTypeId | Nom_InstId
FMCompName ::= FM_AssocName='Nom_CompId
```

```
Duration ::= transient | permanent
ValueCond ::= valueCondition FM_NodeId
FMAssoc ::= fm_assoc FMCompName = [Duration | ValueCond]
          {'Lustre_NodeCallEqList; '}
```

```
EPRule ::= ep_Rule FM_AssocName=' FMCompName
          [FM_NodeId]'→' FMCompName';
| ep_Rule FM_AssocName=' FMCompName'↔' FMCompName';
```

**Figure 6. Fragment of LustreFM Grammar**

defined similar to the internal fault association described in the previous paragraph. However, the activation information is not explicitly specified in this case as it depends on the error propagating sources. It is implicitly derived based on the error propagation rules (Section 4.4), which identify the (conditional or unconditional) activation triggers for the vulnerability specified.

**Priority Definition for Conflict Resolution:** We have an explicit priority definition ( $>$  or  $=$ ) for resolving conflicts between fault behaviors.

### 4.3 Conditional Fault Activation

Conditions can be specified in the following cases - (1) a conditional internal fault activation, or (2) a conditional error propagation for external fault activation, or (3) an explicit boolean condition that combines more than one explicit triggers for any internal fault behavior or vulnerability. If there are more than one activation triggers for any fault specification and in absence of an explicit fault activation condition, the implicit condition is an OR of all the incoming triggers affecting the particular fault behavior.

**Condition as a Lustre Node:** This follows the rules defined in Section 4.1. This type of condition can be used for both conditional activation of internal faults, and conditional error propagation. Note that, for conditional error propagation, the condition should always implicitly consider the activation of the fault behavior in the error originating component for evaluation. For example, the condition that captures the valve open condition (`PipeBurst_Cond`) will be evaluated only when the `Pipe_Burst` fault behavior of `Pipe3` component has been activated.

```
node PipeBurst_Cond (Pressure_In: int)
  return Trigger_Out: bool;
let
  Trigger_Out = (Pressure_In > 0);
tel;
```

The additional trigger dependence will be added by the composition tool, and does not need to be captured by the Lustre node explicitly.

**Condition as a Logical Operator/Expression:** We also support the specification of a boolean logical operator (e.g., OR) or a logical expression to combine multiple incoming triggers that all meant to activate the same internal fault/vulnerability. We skip the details for lack of space.

#### 4.4 Error Propagation Rules

To capture the external faults, we must support the specification of the possible error propagation paths in the system. For this current prototype implementation, we only support explicit definitions of the error propagation paths between arbitrary component instances (this will thus include all types of error propagations including backward and unconnected). To explicitly define error propagation paths, we need to consider the following: (1) the propagation originating component and its corresponding fault behavior, (2) the propagation target component and the corresponding internal vulnerability that gets affected by the propagation, and (3) an optional condition on the outgoing error propagation. Consider the example of an error propagation rule corresponding to the backward error propagation example we considered earlier-

```
ep_Rule Pipe31Prop = PipeBurst:Pipe3
    PipeBurst_Cond(Green_IsolationValve.Pressure_Out)
    -> PipeLeak:Pipe1;
```

The left-hand side of the arrow denotes the error propagating side that also has a propagating condition defined, and the right-hand side specifies the target component and the corresponding vulnerability definition.

#### 4.5 Fault Model Example

Assume we have some basic fault behaviors defined as shown in Section 4.1. In the fault model example (Figure 7), we then start by defining some basic component types. We add the explicit internal fault behavior associations using the construct `fm_assoc`. The vulnerabilities (used as targets in the error propagation rules, `ep_Rule`) are also specified using `fm_assoc`, but with no duration or activation condition defined. An internal fault behavior can be also used as a vulnerability, in which case only the behavior is considered and the duration is dropped (exception is the use of a conditional activation association). In the case where a vulnerability association is made to a component type (`PipeBurst:Pipe`) and also a internal fault association with the same name is made to an component instance belonging to the same type (`PipeBurst:Pipe3`),

the component type association is overridden by the specific component instance association.

For the unconnected error propagations between `Pipe1` and `Pipe2` can be captured as shown in `Pipe12Prop`. The double sided arrow means that the error propagation paths exist in both directions for the components with the corresponding fault behaviors defined. `priorities` are specified to resolve potential conflicts.

## 5 Discussion and Ongoing Work

With the simple fault modeling extensions that we have defined in LustreFM, one can now model the system fault model separate from the nominal system model. Though we have performed these extensions for Lustre, we believe they can be extended with minor modifications to other synchronous dataflow notations, such as Simulink, that the engineers are more familiar with.

We envision the practical system fault model definition process to use libraries of commonly used domain-specific fault behaviors that that can then be specialized (using fault hierarchies) for creating the system-specific fault models. The reusable libraries should also include the domain-specific error propagation rules that would capture the implicit domain-specific constraints on the potential error propagation paths. In addition to having domain-specific component types, for the specification of the generic error propagation rules, one would also need explicit specification of additional constraints on the *direction* of propagation, and allowable intermediate component types. Our ongoing work includes generalization of error propagation rule definitions, and definition of fault hierarchies.

We are currently building an automatic composition tool that will take the nominal system model in Lustre and the fault model in LustreFM as inputs and output a complete extended Lustre model with the fault behaviors added. In addition to the syntactic information, the composition tool must take into account semantic information such as, underlying architecture and data dependency information. This fault modeling and composition technique is in the spirit of aspect-oriented programming [11], where the fault models can be viewed as system aspects that can be statically composed using an aspect weaver. Some of the composition issues are simplified in our domain due to the simpler language semantics.

As a side note, the complexity of fault behavior typically can be reduced by adding more complexity in the nominal behaviors, and vice-versa. Since most of the mechanical component models are typically used only for the safety analysis that analyzes their fault behaviors, we chose to add most of the details in the system fault model and specify only as much detail as needed in the nominal model to perform basic simulations and analysis of the nominal system



```

cType Pipe = Pipe1, Pipe2, Pipe3, Pipe4;
cType IsoValve = Green_IsolationValve, Blue_IsolationValve;

--Internal Faults
fm_assoc StuckOpen:IsoValve = permanent { Pressure_Out = Stuck_at_Open(PipePressure); }
fm_assoc StuckClosed:IsoValve = permanent { Pressure_Out = Stuck_at_Closed(); }
fm_assoc PipeBurst:Pipe3 = permanent { Out1 = Pipe_Burst(); }

--Vulnerabilities
fm_assoc PipeLeak:Pipe = { Out1 = Pipe_Leak(); }
fm_assoc PipeBurst:Pipe = { Out1 = Pipe_Burst(); }

ep_Rule Pipe31Prop = PipeBurst:Pipe3 PipeBurst_Cond(Green_IsolationValve.Pressure_Out) -> PipeLeak:Pipe1;
ep_Rule Pipe42Prop = PipeBurst:Pipe4 PipeBurst_Cond(Blue_IsolationValve.Pressure_Out) -> PipeLeak:Pipe2;
ep_Rule Pipe12Prop = PipeBurst:Pipe1 <-> PipeBurst:Pipe2;
ep_Rule Pipe34Prop = PipeBurst:Pipe3 <-> PipeBurst:Pipe4;

priorities { Stuck_at_Open > Stuck_at_Closed; Pipe_Burst > Pipe_Leak; }

```

**Figure 7. Example Fault Model in LustreFM**

behavior. Thus, our fault modeling support lets one keep the nominal models simple, especially those that are added particularly for the safety analysis, but at the same time uses any details already specified in the nominal model.

## 6 Related Work

There currently exist a number of notations to specify fault models [22, 5, 6, 4, 19, 2]. Here, we discuss some of the most closely related notations and tools.

In the context of the ESACS/ISAAC [14] methodology, FSAP/NuSMV-SA [4] provides an automatic fault tree generation tool based on NuSMV models. The primary focus of their work is on automating safety analysis, and the tool allows specification of only simple base-level component failure modes and their automatic injection in the nominal system model. They do not support any of the more flexible fault behaviors, such as error propagations or multiple failure modes. We believe our fault modeling and composition technique can complement the back-end automated analysis work in ISAAC.

Other notations, such as the AADL Error Annex [22] and the Failure Propagation and Transformation Notation (FPTN) [5], support flexible error modeling including explicit specification of error (failure in FPTN) transformation and propagation. In the AADL Error Annex there are predefined error propagation rules that define potential error propagation paths between various *types* of components and connections; e.g., A processor can propagate error to the process that is hosted on that processor. These error propagations can only occur in the direction along the dataflow in the architecture and cannot occur where the components are not connected to each other either through direct connections (port or access) or through explicit bindings. The main advantage of the AADL annex, is that it enables specification of error annotations on the original AADL [21] architecture model which provides inherent support for bindings

between the physical and logical architectural components, which is a critical consideration for system safety analysis. The AADL Error Annex is quite a comprehensive notation for architectural models, and our earlier experience with this notation [10] has influenced some of our behavioral modeling extensions. FPTN is a simpler notation as compared to the AADL Error Annex. It can explicitly specify and transform failures between three categories - value failure, commission, and omission.

The important distinction of our approach with respect to the AADL and FPTN notations is that we constructively specify the fault behaviors and hence can analyze how the fault behaviors interact with each other, and also with the underlying nominal component behaviors. Also, the fault behaviors that can be captured in our approach are more flexible, e.g., modeling error propagations in the reverse direction to the dataflow and between unconnected components. Since we can control the activation of the faults through system inputs, our technique can enable simulations of different fault scenarios and how the system responds to such faults. Based on the feedback from a simple simulation demonstration illustrating our MBSA approach to a practicing safety engineer, we realized that the safety engineers find simulation appealing for getting quick insights into the system's response and fault tolerance.

One of the main drawbacks of our current approach arises from the limitations of the behavioral notations when it comes to capturing the architecture of the system. For example, the physical-logical component bindings that can be performed elegantly in architecture description languages, such as AADL, are not supported in these notations. In the future, we plan to integrate the higher-level architectural and lower-level behavioral notations to be able to derive the benefits of both classes of modeling notations for fault modeling and safety analysis.

## 7 Conclusion

To make model-based safety approach based on behavioral models feasible, there is a need for providing language support for specifying simple yet realistic fault models and also providing tool support for automatically composing the fault models into the nominal system models for analysis. This paper identifies the key requirements for flexible behavioral fault modeling for model-based safety analysis. Based on these requirements, fault modeling-specific extensions to Lustre amenable to automatic composition are proposed. Though these language extensions are defined for Lustre, they could be applied to other synchronous dataflow languages with minimal changes. We are in the process of implementing a composition tool to automatically extend the nominal model based on the fault model, following which our approach can be rigorously evaluated.

## 8 Acknowledgement

The authors wish to thank Dr. Michael Whalen (Rockwell Collins Inc.) for his feedback and insightful comments. The authors also wish to thank Michael Peterson (Rockwell Collins Inc.) for his feedback and safety engineering perspective during the early stages of our MBSA approach.

## References

- [1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable and Secure Computing*, 1(1):11–33, 2004.
- [2] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In *Proceedings of the 4th European Dependable Computing Conference*, pages 19 – 31. Springer-Verlag, 2002.
- [3] M. Bozzano and etal. Esacs: an integrated methodology for design and safety analysis of complex systems. In *Proceedings of ESREL 2003*, pages 237–245. Balkema Publishers, June 15-18 2003.
- [4] M. Bozzano and A. Villaflorita. Improving system reliability via model checking: the fsap / nusmv-sa safety analysis platform. In *In Proceedings of SAFECOMP 2003*, pages 49–62, Edinburgh, 2003. Springer.
- [5] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.
- [6] H. Giese, M. Tichy, and D. Schilling. Compositional hazard analysis of uml component and deployment models. In *Computer Safety, Reliability, and Security*, volume 3219 of LNCS, pages 166–179. Springer, 2004.
- [7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [8] A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of LNCS, pages 122–135. Springer-Verlag, Sept 2005.
- [9] A. Joshi, S. Miller, M. Whalen, and M. P. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proceedings of 24th DASC*, Nov 2005.
- [10] A. Joshi, S. Vestal, and P. Binns. Automatic generation of static fault trees from aadl models. In *DSN Workshop on Architecting Dependable Systems (WADS)*, Edinburgh, 2007.
- [11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in LNCS, pages 220–242. Springer-Verlag, 1997.
- [12] Mathworks Inc. Simulink product web site. Via the world-wide-web: <http://www.mathworks.com/products/simulink>.
- [13] The NusMV Toolset, 2005. Available at <http://nusmv.irst.itc.it/>.
- [14] O.Akerlund, P.Bieber, E.Boede, M.Bozzano, M.Bretschneider, C.Castel, A.Cavallo, M.Cifaldi, J.Gauthier, A.Griffault, O.Lisagor, A.Luedtke, S.Metge, C.Papadopoulos, T.Peikenkamp, L.Sagaspe, C.Seguín, H.Trivedi, and L.Valacca. Isaac, a framework for integrated safety analysis of functional, geometrical and human aspects. In *In Proceedings of ERTS*, Toulouse, France, 2006.
- [15] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif. Combining formal methods and safety analysis - the formosa approach. In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of LNCS, pages 474–493, 2004.
- [16] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993.
- [17] G. Pai and J. Dugan. Automatic synthesis of dynamic fault trees from uml system models. In *13th International Symposium on Software Reliability Engineering (ISSRE)*, pages 243–254, Annapolis, USA, 2002.
- [18] Y. Papadopoulos and M. Maruhn. Model-based synthesis of fault trees from matlab-simulink models. In *The International Conference on Dependable Systems and Networks (DSN'01)*, July 01 - 04 2001.
- [19] Y. Papadopoulos and J. A. McDermid. Hierarchically performed hazard origin and propagation studies. In *In Proceedings of the 18th International Conference, SAFECOMP'99*, volume LNCS 1698. Springer-Verlag, 1999.
- [20] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.
- [21] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.
- [22] SAE-AS5506/1. *Architecture Analysis and Design Language Annex Volume 1*. SAE, June 2006.