Quantitative Fault Propagation Analysis for Networked Cyber-Physical Systems

Linda Briesemeister,* Grit Denker,* Daniel Elenius,* Ian Mason,*
Srivatsan Varadarajan,† Devesh Bhatt,† Brendan Hall,† Gabor Madl,† and Wilfried Steiner‡
*SRI International, Menlo Park, CA, USA
†Honeywell Labs, Golden Valley, MN, USA
‡TTTech Computertechnik AG, Vienna, Austria
firstname.lastname@[sri|honeywell|tttech].com

***Abstract*—This paper presents an approach to analyzing a model of networked cyber-physical systems for fault propagation. We present an implementation of a probabilistic logic model, which allows for reasoning via symbolic evaluation as well as numeric evaluation to perform a quantitative fault analysis. Our models are built from a few building blocks, which can be instantiated as standard or high integrity; communication paths can be made redundant, and finally, whole subsystem blocks can be replicated. We assume an underlying networking infrastructure of TTEthernet, which allows traffic of time-triggered, rate-constrained, or best-effort modes with different safety features. We apply our approach to a case study of a brake-by-wire system that contains communication flows with different traffic modes according to their criticality.**

I. Introduction

Safety-critical networked systems such as the avionics in an airplane or an automotive X-by-wire system typically have to be fault tolerant, i.e., these systems have to remain operational even in the presence of failures. Achieving fault tolerance requires safety analysis that is best performed beginning with the earliest design stages. Part of such design analysis is the so-called fault propagation analysis of the system model. In safety-critical systems, we are interested in characterization faults and how faults manifest as failures that affect overall reliability goals of the system.

In this paper, we present a fault analysis tool for networked cyber-physical systems (CPSs). We present networked CPSs as a graph where nodes represent physical components, such as CPU, communication controller, and physical links, and edges represent the direct information flow between nodes. We assume that faults occurring in a particular node may propagate along the connections in the graph representing the networked cyber-physical systems. For example, a faulty communication controller may send an arbitrary number of faulty messages. Without having protection mechanisms in place, the failure of the communication controller could lead to monopolizing the shared network infrastructure, thereby making it unusable for other non-faulty communication controllers.

Various redundancy schemes and safety techniques can be used as a protection mechanism to prevent or mitigate failure propagation and, thus, improve system availability and integrity. However, the decision about where to locate what protection mechanism along the graph of the networked CPS currently relies heavily on expert knowledge. The proposed tool allows designers to efficiently analyze various network choices and evaluate the effect of protection mechanisms on fault propagation, and therefore rely less on expert knowledge.

We provide a formal Maude [4] model that uses basic building blocks (Host, End System, Switch) to model networked CPS topologies. Given an underlying networking infrastructure of TTEthernet (TTE) [15], which is an extension to standard Ethernet, we can select traffic modes with varying safety properties. Other safety techniques are selecting components with standard- or high-integrity features and adding path and system redundancy. We then combine this network and protection mechanism model with fault propagation rules that describe in a probabilistic logic how faults occurring in a component, or at an input of a component, transform into a fault at the output. Our implementation allows for both symbolic and quantitative fault propagation analysis. We use the example of a brake-by-wire (BBW) system with multiple communication flows to illustrate our approach.

II. Related Work

Our work is mostly inspired by the functional and component failure analysis of the Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [14] method. HiP-HOPS uses these analyses

Approved for Public Release, Distribution Unlimited.

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

in conjunction with the system model to automatically construct fault trees. Our symbolic analysis yields probabilistic logic formulas, which could be interpreted as fault trees. In contrast to our work, HiP-HOPS supports mature tool integration and hierarchical models.

An inductive method to study fault propagation is the Failure Propagation and Transformation Analysis (FPTA) [6]. A failure logic for each component allows for automated analysis of the system. Here, the failure logic depends only on the inputs, which is appropriate for modeling software failures, but cannot explicitly use failure states of the component itself. In [5], the authors present an approach to quantify the failure propagation analysis using the PRISM probabilistic model checker.

So-called State/Event-Fault Trees (SEFTs) [8] integrate finite state models with fault trees, in particular Markov Chains and Statecharts. For quantitative probabilistic analysis, SEFTs are translated component-wise into Dynamic Stochastic Petri Nets (DSPNs) and then merged into one flat net for analysis using tools such as TimeNet.

III. Networked System Architecture

For this study, we assume an underlying communication network using TTEthernet (TTE) technology [16]. TTE is an extension to standard Ethernet for usage in networked cyber-physical systems such as avionics or automotive applications. In standard Ethernet, messages are communicated according to a best-effort paradigm, which means that no bounds on a message's transmission latency can be given. Under high load, buffers in network switches can overflow and messages are lost entirely. In noncritical systems, higher-layer protocols such as TCP/IP often compensate for message loss using re-transmission strategies. However, networked cyber-physical systems typically have stringent end-to-end timing requirements that do not allow the temporal penalty of repeated transmission attempts. Furthermore, standard networks cannot guarantee that any successive transmissions will actually be successful.

TTE achieves timely transmissions with known upper bounds on latency and jitter by providing rate-constrained (RC) and time-triggered (TT) communication paradigms in addition to standard best-effort (BE) traffic. Rate-constrained traffic is well-known from an avionics Ethernet variant ARINC 664P7-1 [1]. It is based on an *a priori* agreement of the network components on the number, size, and maximum frequency of messages. This knowledge is sufficient to calculate the required network resources, i.e., the switch buffers, and it can be guaranteed that no message will be lost due to buffer overflows. However, different network components may source their messages at about the same point in time or may even source several messages back to back. This uncoordinated transmission pattern quickly leads to high transmission latencies.

The time-triggered communication paradigm takes the level of determinism to the extreme. Here, all communication participants are equipped with local clocks that are brought in close agreement to establish a synchronized global time. An Ethernet frame may now be dispatched at an *a priori* specified point in the global time, which makes the frame transfer time-triggered. The sum of all those specified points in time for dispatch, relay, and potentially also reception is collectively referred to as the "communication schedule" for time-triggered communication. When correctly aligned, the communication schedule ensures that any two time-triggered Ethernet frames will never compete for transmission resources (e.g., communication links, switch buffers) and their transmission latencies can be kept minimal and almost constant.

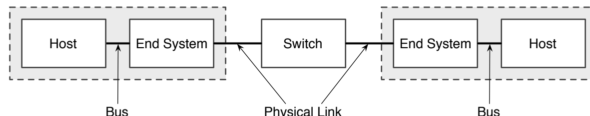


Fig. 1. Simple chain of network components

To model networked CPSs, we identified a few generic components from which we build networks. Fig. 1 shows an example of these components that are connected via a communication link. We distinguish the host (H) component (CPU, memory, I/O and so on) from an end system (ES), which denotes the network interface card, here the TTE controller. H and ES components are often located on the same physical platform and are usually connected via an on-chip bus. To connect different ESs in the network, one uses at least one switch (SW) in between. For our models, these are TTE SWs, which allow the different traffic modes, but a legacy system could also be built with standard Ethernet SWs. A bus connects Hs and ESs, and a physical link connects SWs with other SWs or ESs.

A. Dependability Means

Aside from choosing an appropriate traffic mode that TTE supports, we consider the following three broad fault tolerance strategies that improve the *integrity* and *availability* properties of a network architecture and thereby also enhance the safety of the network. Integrity is the absence of improper system alteration; it is the ability of a system to detect faults in its own operation and to reach a fail-safe state or safe output states in the event of failure and inability to recover. Availability

is a measure that the system functions correctly in the presence or absence of both transient and permanent failures of the different network components. We discuss the three strategies next.

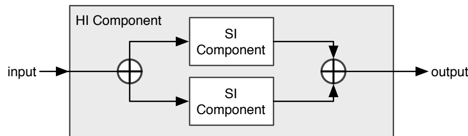


Fig. 2. High-integrity self-checking pair

In our model, all Hs, ESs, and SWs can be instantiated as **standard-** or **high-integrity** components. As shown in Fig. 2, a High-Integrity (HI) component consists of two replicated Standard-Integrity (SI) components functioning as a single unit for increased integrity. The input messages received at each SI component pair are exchanged and compared to ensure that each of them receive identical inputs and as a result triggers identical internal message processing states in each component. The output messages are also compared and only identical messages whereby both components agree are in effect transmitted. The expectation is that by providing each SI component with identical inputs, using the same internal processing logic, states and clocks, the pair's outputs will match exactly under fault-free conditions. When operators $\bigoplus$ during input exchange or output comparison do not agree, then the combined high-integrity component fails silently so as to not influence downstream components. In our formal model, this behavior translates into converting an arbitrary fault into an omission failure. Notice that this mechanism uses replication of components solely for integrity at the cost of availability.

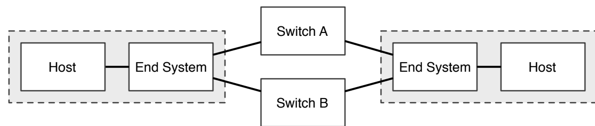


Fig. 3. Path redundancy in simple chain

Including **path redundancy** increases the availability of a message by transmitting an identical copy over disjoint but redundant paths from the source to the destination. Fig. 3 shows the simple chain of network components enhanced with two switches A and B that allow for disjoint paths between the ESs. The receiving ES now picks the first arriving message from the sender, thereby protecting against message loss on one of the paths due to permanent or transient failures of the SWs or links on that path. Note that this mechanism does not protect against failures at source or destination components.

Finally, **system redundancy** in conjunction with the availability and integrity constructs introduced above, can be used to improve the overall safety and reliability of the network, if safety properties like replicate group determinism or interactive consistency are strictly adhered to [2], [7], [9], [10]. Literature defines architectures such as dual-dual active standby, triple or N-modular redundancy, and triplex-triplex [3], [17]. Additional replication coordination mechanisms are passive, semi-active, or active replication; voting and multistage N or Triple Modular Redundancy (N(T)MR), congruency exchange.

B. Faults

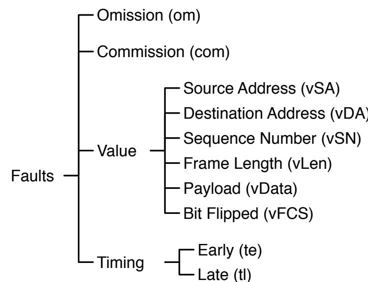


Fig. 4. Hierarchy of faults

We depict a hierarchy of faults in Fig. 4. This hierarchy is meant as a reference to typical faults used in literature. In our study, we are concerned only with the leaf nodes, which comprise the set of possible faults in our network-centric application with specific modes for the TTE communication technology used such as vSN for incorrect sequence numbers in rate-constrained traffic.

Faults listed in Fig. 4 can be introduced in specific components, or can be detected and stopped or transformed to a different fault at some components and thus propagated downstream. A number of fault detection and protection mechanisms are builtin to the different network components and depend on the type of traffic mode: Time-Triggered (TT), Rate-Constrained (RC) or Best-Effort (BE). We model the fault propagation probabilistically, taking into account the component failure rates and the efficacy of the protection mechanisms. We highlight some of the fault detection and protection mechanisms in Table I. The TTE specification [15] contains more details.

IV. A CASE STUDY: BRAKE-BY-WIRE

Papadopoulos et al. [13] describe an initial brake-by-wire model that we are extending to include communication of signals from the wheel brakes to the brake

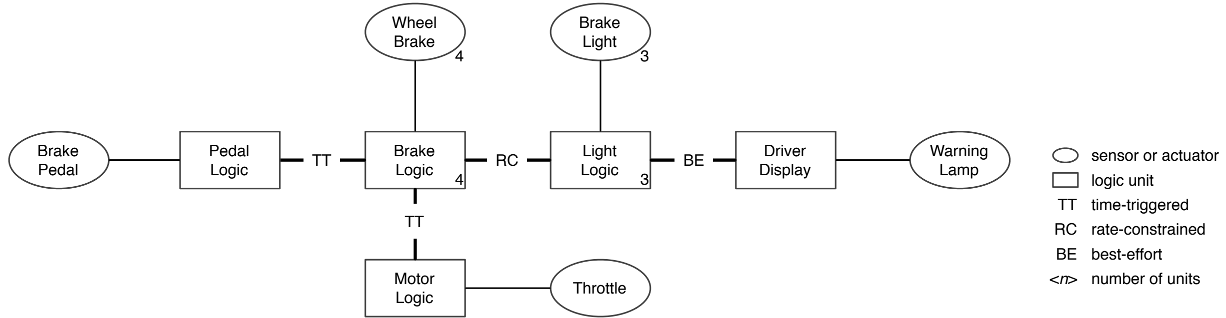


Fig. 5. High-level overview of brake-by-wire system design

TABLE I
FAULT DETECTION AND PROTECTION MECHANISMS

Mechanism	Traffic Type	Components
FCS/CRC addition	TT, RC, BE	ES, SW
VL ID & destination address	TT, RC, BE	ES, SW
Message length and type	TT, RC, BE	ES, SW
Payload length	TT, RC	ES, SW
Scheduled TT dispatch	TT	ES, SW
FIFO ordering	RC, BE	ES, SW
Traffic shaping	RC	ES
Timing window	TT	SW
BAG policy	RC	SW
Port BAG enforcement	BE	SW
Age check	RC	SW
Redundancy management	TT, RC	ES
Integrity checking	TT, RC	ES
Input exchange	TT, RC, BE	High Integrity
Output cross comparison	TT, RC, BE	High Integrity

TABLE II
TRAFFIC FLOWS IN BRAKE-BY-WIRE SYSTEM

VL	Sender	Receiver(s)	Type
1	Pedal Logic	Brake Logic 1, 2, 3, and 4	TT
2	Brake Logic 1	Motor Logic	TT
3	Brake Logic 2	Motor Logic	TT
4	Brake Logic 3	Motor Logic	TT
5	Brake Logic 4	Motor Logic	TT
6	Brake Logic 1	Light Logic 1, 2, and 3	RC
7	Brake Logic 2	Light Logic 1, 2, and 3	RC
8	Brake Logic 3	Light Logic 1, 2, and 3	RC
9	Brake Logic 4	Light Logic 1, 2, and 3	RC
10	Light Logic 1	Driver Display	BE
11	Light Logic 2	Driver Display	BE
12	Light Logic 3	Driver Display	BE

lights as well as feedback to the driver from the light sensors about whether any of the bulbs need to be replaced. We also include a safety-critical communication of brake signals to the motor control logic in order to prevent opening the throttle while braking.

Fig. 5 shows a high-level overview of our brake-by-wire system design. Each sensor and actuator has a corresponding logic unit, which interfaces with the TTE communication infrastructure. Corresponding to the criticality of the signal, the TTE communication links are labeled as TT (high criticality), RC (medium criticality), and BE (low criticality). The highly critical paths should also be protected by redundancy. The following rules govern the expected behavior of the system.

- If brake pedal is engaged, brake at each wheel.
- If wheel brake is engaged, illuminate brake lights.
- If wheel brake is engaged, close throttle at motor.
- If brake light does not work, show warning.

From the general system specification, we model the system using the components and mechanisms introduced above. For each message flow, we assign a so-called *Virtual Link* (VL) between the source and the destination(s). During design refinement, each VL is mapped onto actual channels in the network architecture. Table II contains all 12 virtual links in our model of the brake-by-wire system.

V. MAUDE IMPLEMENTATION

A. Equational Logic and Maude

Equational logic (EL) [12] is the subset of first-order logic with $=$ as the only predicate symbol, and equations as the only formulas (i.e., there are no logical connectives). Despite being a very small subset of first-order logic, equational logic can be used to define any computable function. Furthermore, EL can be used as a programming language, by treating equations as left-to-right rewrite rules (i.e., ignoring the symmetry rule).

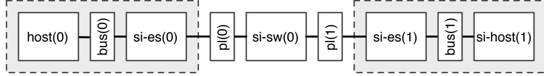
Maude is a multiparadigm executable specification language encompassing EL. The Maude interpreter provides efficient prototyping of quite complex test cases as well as built-in search and model checking capabilities.

For the fault analysis tool, we use (conditional) equations to specify fault propagation rules. Terms for the equations are built from operators and variables. Equational axioms are introduced with the keyword `eq` (or `ceq` for conditional equations) followed by the two

terms being declared equal separated by the equality sign =.

B. Probabilistic Fault Analysis Using Maude

We have developed a network fault analysis in Maude that allows us to specify network topologies and traffic flows, and analyze the fault introduction and propagation in the network in a probabilistic way. The most important data structures in this framework are *network configurations*, consisting of a *network* and one or more *dataflows*. A simple network configuration specified in this framework is shown in Fig. 6.




```
op net : -> NetworkConfiguration .
eq net =
  (< si-host(0) | empty | out(0) >,
   < conn(0) | si-host(0) : out(0) to bus(0) : in(0) >,
   < bus(0) | in(0) | out(0) >,
   < conn(1) | bus(0) : out(0) to si-es(0) : in(0) >,
   < si-es(0) | in(0) | out(0),out(1) >,
   < conn(2) | si-es(0) : out(0) to pl(0) : in(0) >,
   < pl(0) | in(0) | out(0) >,
   < conn(3) | pl(0) : out(0) to si-sw(0) : in(0) >,
   < si-sw(0) | in(0) | out(0) >,
   < conn(4) | si-sw(0) : out(0) to pl(1) : in(0) >,
   < pl(1) | in(0) | out(0) >,
   < conn(5) | pl(1) : out(0) to si-es(1) : in(0) >,
   < si-es(1) | in(0),in(1) | out(0) >,
   < conn(6) | si-es(1) : out(0) to bus(1) : in(0) >,
   < bus(1) | in(0) | out(0) >,
   < conn(7) | bus(1) : out(0) to si-host(1) : in(0) >,
   < si-host(1) | in(0) | out(0) >,
   < conn(8) | si-host(1) : out(0) to null : in(0) >) ||
   df(0 | tt |
     faults(conn(0) | unknown ),
     faults(conn(1) | unknown ),
     faults(conn(2) | unknown ),
     faults(conn(3) | unknown ),
     faults(conn(4) | unknown ),
     faults(conn(5) | unknown ),
     faults(conn(6) | unknown ),
     faults(conn(7) | unknown ),
     faults(conn(8) | unknown )) .
```

Fig. 6. Example of a network configuration specified in Maude

The network part consists of a number of *network components*, each surrounded by angle brackets. The network specified here is an all-SI version of the network shown in Fig. 1. We have two SI hosts, two buses, two SI end systems, two physical links (PL), and an SI switch. Each component has ingoing and outgoing *ports*, connected by *connections*. Numbers are used as identifiers to distinguish different components of the same type. The bottom part of the network configuration is a dataflow (df). The dataflow specifies its traffic type (in this case `tt` for time triggered). It also lists all the connections that are part of the dataflow, with a *fault annotation* for each one. Before fault analysis, each fault annotation is `unknown`. The first objective of the fault analysis is to replace all the `unknown`s with boolean formulas representing the conditions under which the fault occurs.

Each network component can introduce and/or propagate faults of the different types shown in Fig. 4. The introduction/propagation behavior of each component is specified using one or more equations in the Maude specification. For example, the rule for transmitting SI host is shown in Fig. 7.

```
ceq
  (< SIHost | INS | Out1,OUTS >,
   < Conn1 | SIHost : Out1 to C : In1 >, Net) ||
   (df(DF | TType | faults(Conn1 | unknown), CS),DFS)
  =
   (< SIHost | INS | Out1,OUTS >,
    < Conn1 | SIHost : Out1 to C : In1 >, Net) ||
   (df(DF | TType |
      faults(Conn1 | om : OMout, com : COMout,
        vSA : VSAout, vDA : VDAout, vSN : VSNout,
        vLen : VLENout, vData : VDATAout,
        vFCS : VFCSout, te : TEout, tl : TLout ),CS),
        DFS)
if
  outgoingBusConn(< Conn1 | SIHost : Out1 to C : In1 >,
    Net, CS) /\
  HFail := pr(si-host-out-fail,SIHost : Out1) /\
  OMout := HFail /\
  COMout := HFail /\
  VSAout := if TType == be then HFail else false fi /\
  VDAout := HFail /\
  VSNout := false /\
  VLENout := if TType == be then HFail else false fi /\
  VDATAout := HFail /\
  VFCSout := false /\
  TEout := HFail /\
  TLout := HFail .
```

Fig. 7. Fault introduction rule for SI hosts

This is a conditional equation, consisting of a left side before the = sign, a right side after the = sign, and a condition after the `if`. Note that all the capitalized parts are variables. The rule will execute when any part of the network matches the left side, and the condition is true. Thus, this particular rule applies to SI hosts with an outgoing connection that is part of a dataflow. The result of executing the rule is to replace the matched part of the network with the right side of the equation. The only difference between the left and right sides of the equation is that in the right side, the `unknown` fault annotation has been replaced by a number of actual fault annotations for the different fault types (`om`, `com`, etc.). The variables that represent the boolean formulas (`OMout`, `COMout`, etc.) are defined in the condition part of the equation. The condition first creates a probabilistic variable (using the `pr` operator) called `HFail` of type `si-host-out-fail`. This variable represents a failure associated with the outgoing port of the host.[1] This variable is then used in the variable assignments that follow in a straightforward way. Most of the faults simply happen if and only if the host failure happens.

[1]It is also possible to create more fine-grained failure types, rather than one big failure type for the whole component.

Because a transmitting host is the originator of any data it sends, unlike all other components, it cannot propagate faults, only introduce them. For a more typical case, Fig. 8 shows part of the condition of the rule for SI end systems. Here, we have both propagation and introduction of faults, and separate possible failures on the incoming and outgoing ports, respectively.

```
ESInFail := pr(si-es-in-fail,SIES : In2) /\
ESOutFail := pr(si-es-out-fail,SIES : Out2) /\
OMout := OMin or (COMin or  VDAin or TEin or TLin) and
  not ESInFail or ESInFail or  ESOutFail /\
COMout := ESOutFail /\
VSAout := ESOutFail /\
VDAout := (VDAin and ESInFail) or ESOutFail /\
VSNout := if TType == rc then ESOutFail else false fi /\
VLENout := ESOutFail /\
VDATAout := VDATAin or ESInFail or ESOutFail /\
VFCSout := ESOutFail /\
TEout := ESOutFail /\
TLout := ESOutFail
```

Fig. 8. Part of fault rule for SI end systems

Note that the incoming failures (`OMin`, `COMin`, etc) are themselves (possibly complex) formulas. Thus, the execution of the equations builds up these formulas in a combinatorial way. We are primarily interested in the fault annotation on the last connection of the dataflow, i.e., the combined effect of all of the network components that data has to go through. As an example, Fig. 9 shows the fault formula for `com` faults on connection 8 of our example network, after doing the fault analysis.

```
pr(si-es-out-fail, si-es(1) : out(0)) or
pr(si-es-in-fail, si-es(1) : in(0)) and
  (pr(si-sw-out-fail, si-sw(0) : out(0)) or
    pr(si-es-out-fail, si-es(0) : out(0)) and
  pr(si-sw-in-fail, si-sw(0) : in(0)))
```

Fig. 9. A fault formula generated by the Maude fault analysis

Again, the formula is a boolean combination of probabilistic variables that represent the occurence of failures in various components through which the data is transmitted. In general, these formulas can become quite large. The next step of fault analysis is to evaluate the formula. Given some actual numbers for each type of failure (`si-es-out-fail`, `si-sw-in-fail`, etc.), we want to know the probability for the whole compound formula. There are both exact and approximate ways of doing this. In the following, we present our first implementation of both types of methods.

To calculate the probability of one of our formulas, we use the following rules of probabilistic logic.[2]

$Pr(\texttt{not}\ P) = 1.0 - Pr(P)$
$Pr(\texttt{true}) = 1.0$
$Pr(\texttt{false}) = 0.0$
$Pr(P\ \texttt{and}\ Q) = Pr(P) * Pr(Q)$
    if P and Q are independent
$Pr(P\ \texttt{or}\ Q) = Pr(P) + Pr(Q)$
    if P and Q are disjoint

[2] http://en.wikipedia.org/wiki/Probability_space

The issue here that prevents a straightforward calculation is the side conditions for the and- and or- rules. The two subformulas are typically *not* independent or disjoint due to variables appearing in both. There are different approaches to computing with probabilistic logic [11].

One approach is to convert the whole formula into a form where all conjunctions are independent and all disjunctions are disjoint. It turns out that formulas in full disjunctive normal form (full DNF) are of the type described. A formula is in DNF if and only if it is a disjunction of one or more conjunctions of one or more literals (a literal is either a propositional variable or a negated variable). For example, $(P\ \texttt{and}\ Q)\ \texttt{or}\ (P\ \texttt{and}\ R)$ and $P\ \texttt{or}\ (Q\ \texttt{and}\ \texttt{not}\ R)$ are both in DNF. A formula is in *full* DNF if and only if it is in DNF and if each of its variables appears exactly once in every clause. Any formula can be converted to full DNF. For example, the first formula above becomes

$(P\ \texttt{and}\ Q\ \texttt{and}\ R)\ \texttt{or}\ (P\ \texttt{and}\ Q\ \texttt{and}\ \texttt{not}\ R)\ \texttt{or}$
$(P\ \texttt{and}\ \texttt{not}\ Q\ \texttt{and}\ R)$

Note that each formula has a unique full DNF form, but not a unique DNF form. The full DNF form can be interpreted as the entries in a truth table that make a formula true. For example, for the formula above (with 1 for true and 0 for false) the truth table is shown in Table III.

TABLE III
FULL DNF FORMULA AS A TRUTH TABLE

P	0	0	0	0	1	1	1	1
Q	0	0	1	1	0	0	1	1
R	0	1	0	1	0	1	0	1
(P and Q) or (P and R)	0	0	0	0	0	1	1	1

Each column describes one complete variable assignment, or "possible world." Each column differs in at least one position. Hence, each column is disjoint from all the others. They describe different states of affairs that cannot be true at the same time. Similarly, each row for the atomic variables is independent of the others, since we already stated that we assume that the atomic variables are independent.

Looking at the table, we see that the entire formula (last row) is true exactly in the situations described by the rightmost three columns. In other words, when P is true, Q is false, and R is true, OR when P is true, Q is true, and R is false, OR when all three are true. As can be seen from this verbiage, each column

can be interpreted as a conjunctive formula, and the combination of several columns can be interpreted as a disjunction. Each column contains all the variables. Hence, what we get from the table is a full DNF formula, the probability of which we can easily compute using the rules above. Our first implementation used this approach, within the Maude framework.

The problem with the "full DNF" method described above is its computational complexity. The conversion to full DNF (or even regular DNF) form is exponential in the number of variables contained in the formula. Thus, we have been able to use this method only for small examples, and as a reference implementation with which to compare other approaches. Our second approach uses Binary Decision Diagrams (BDDs) as a representation instead of converting the formulas to full DNF form. There are several highly efficient BDD packages. These can very quickly return all variable assignments that satisfy a given formula/BDD. Given those variable assignments, we can calculate our probabilities the same way as with the full DNF formulas. Our implementation uses Maude to generate the fault formulas, and then uses the JavaBDD[3] library to analyze the results in Java. In our tests, computation time with BDDs was negligible, even where the original implementation crashed or stalled.

It is still possible to hit upon limitations with the BDD approach, as its theoretical worst-case behavior is still NP-hard. Another approach to calculating the probabilities is to use an approximate, sampling-based method. The idea is to generate a number of samples, where each sample randomly determines the values (true or false) of all the probabilistic variables, according to their probabilities. The formula is then evaluated with all the variables replaced with true or false. The whole formula thus becomes true or false for each sample. With enough samples, the distribution of true/false for the entire formula will approach the correct result that would be calculated by an exact method such as the one described above. We have implemented a sampling-based method using a combination of Java code and Maude. The implementation is capable of evaluating several hundred thousand samples per second. However, our failure probabilities are sometimes on the order of 10^{-9}. In order to "catch" these tiny probabilities, we need a huge number of samples and hours of computation time. Thus, this method is still not fast enough for practical purposes. However, we are exploring alternative, weighted sampling methods that may be applicable, which would make the computation much faster.

[3]http://javabdd.sourceforge.net/

C. Fault Analysis Results for BBW

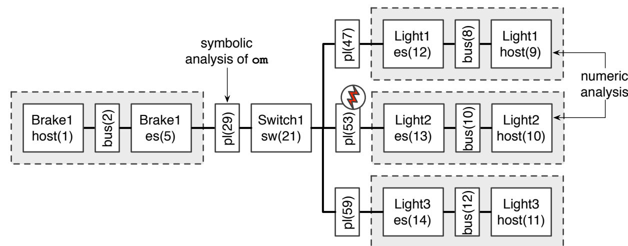


Fig. 10. Example VL 6 (rate-constrained, no path redundancy) with points of symbolic and numeric analysis and fault introduction

Let us reconsider the BBW application introduced in Section IV. Fig. 10 depicts the subgraph of VL 6 as an example. In our implementation, we model buses and physical links as part of the network, so as to define failures on such links. In VL 6, a signal from one wheel brake is sent to all three brake lights to illuminate. The numbers in parenthesis denote identifiers in the Maude code.

In Fig. 11 we show the symbolic analysis for omission faults for the connection from the Brake1 end system to the physical link toward Switch1. The symbolic analysis for connections further downstream becomes too large to print here.

```
pr(bus-fail, bus(2)) or
pr(si-host-out-fail, si-host(1) : out(0)) or
pr(si-es-in-fail, si-es(5) : in(0)) or
pr(si-es-out-fail, si-es(5) : out(1)) or
not pr(si-es-in-fail, si-es(5) : in(0)) and
 (pr(bus-fail, bus(2)) or
  pr(si-host-out-fail, si-host(1) : out(0)))
```

Fig. 11. Results of VL 6 symbolic analysis (for connection from Brake1 to physical link with Switch1)

TABLE IV
EXAMPLE FAILURE PROBABILITIES FOR BBW

Fault Type	Probability*)
FCS check	10^{-9}
physical link	10^{-9}
bus	10^{-9}
SI host input	10^{-6}
SI host output	10^{-6}
SI end system input	10^{-6}
SI end system output	10^{-6}
SI switch input	10^{-6}
SI switch output	10^{-6}

*) These are not representative numbers for real network components.

When analyzing the quantitative fault propagation of a VL, we first supply the failure probabilities for each component type as a valuation set. Table IV shows the failure probabilities used for the example computation

below. Fig. 12 contains the result of analyzing VL 6 at the receiving hosts Light1 (without any introduced fault) and at Light2 (with a physical link failure introduced between Switch1 and Light2).

```
om : 7.003789723448e-5,     om : 1.0,
com : 1.00001999971e-5,     com : 1.00001999971e-5,
vSA : 1.001000000009e-5,    vSA : 1.001999979992e-5,
vDA : 1.001010010009e-5,    vDA : 2.001989959991e-5,
vSN : 3.002969910071e-5,    vSN : 1.0,
vLen : 1.001010010009e-5,   vLen : 2.001989959991e-5,
vData : 1.001010010211e-5,  vData : 2.001989959991e-5,
vFCS : 1.0010000000099e-5,  vFCS : 1.001999979992e-5,
te : 1.000000000002e-10,    te : 1.000000000002e-10,
tl : 1.999990000002e-10     tl : 1.999990000002e-10
```

Fig. 12. Results of VL 6 analysis (without and with introduced fault) using scientific notation (i.e., 1e-9 $= 10^{-9}$)

One can see clearly that a faulty physical link at the input of Light2 causes an omission error and also a violation of the sequence numbers that are expected during a rate-constrained communication. If the traffic mode were to be set to time-triggered, the sequence numbers no longer apply. However, the omission error would still occur. If the system-level requirements call for better reliability of the message delivery from the brakes to the lights, the system design engineer could now test an architecture with path redundancy or other enhancements.

VI. Conclusion and Future Direction

The Maude-based Fault Analysis tool provides a means to evaluate the effectiveness of fault protection mechanisms in various network architectures. We have implemented the tool and applied it to various sample networks. A special-purpose editor automatically generates input to the Maude Fault Analysis tool. Because of space limitations, we have not featured the editor in this paper. However, the network editor and the fault analysis tool will be made publicly available.[4]

The evaluation on a simple brake-by-wire network with high- and standard integrity components, and path redundancy but no system redundancy, illustrates the complexity of fault formulas. The analysis tool successfully performs symbolic and quantitative fault analysis.

Future extensions concern extending the underlying network model and fault propagation rules for system redundancy schemes and analyzing the symbolic fault formulas to determine main contributors to faults, so as to point the designer where additional protection would have the most payoff. Another direction includes modeling fault propagation with temporal aspects, possibly extracting dynamic fault trees for better comparison with other approaches. An additional temporal aspect to consider is how failure rates change over time (e.g., as a "bathtub curve"). Our current tool assumes constant failure rates.

[4]http://promise.csl.sri.com

References

[1] ARINC Report 664P7-1. *Aircraft Data Network, Part 7: Avionics Full Duplex Switched Ethernet (AFDX) Network*, Sept. 2009.

[2] R. W. Butler. A primer on architectural level fault tolerance. Technical Report TM-2008-215108, NASA, Feb 2008.

[3] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *FTCS*, pages 28–37, 1992.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] X. Ge, R. Paige, and J. McDermid. Probabilistic failure propagation and transformation analysis. In *Computer Safety, Reliability, and Security (SAFECOMP)*, volume 5775 of *Lecture Notes in Computer Science*, pages 215–228. 2009.

[6] X. Ge, R. Paige, and J. McDermid. Analysing system failure behaviours with PRISM. In *International Conference on Secure Software Integration and Reliability Improvement Companion*, pages 130–136, June 2010.

[7] R. C. Hammett and P. S. Babcock. Achieving 10^{-9} dependability with drive-by-wire systems. In *SAE 2003 World Congress and Exhibition, March 2003,Session: Safety Critical Systems*, Detroit, March 2003.

[8] B. Kaiser, C. Gramlich, and M. Förster. State/event fault trees – a safety analysis model for software-controlled systems. *Reliability Engineering & System Safety*, 92(11):1521–1537, 2007.

[9] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The maft architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37:398–405, 1988.

[10] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[11] N. J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87, 1986.

[12] M. J. O'Donnell. Equational logic as a programming language. In R. Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, page 255. Springer, 1985.

[13] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering & System Safety*, 71(3):229–247, Mar. 2001.

[14] Y. Papadopoulos and J. A. McDermid. Hierarchically performed hazard origin and propagation studies. In *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 688–688. 1999.

[15] W. Steiner. *TTEthernet Specification*. TTA Group, 2008. Available at http://www.ttagroup.org.

[16] W. Steiner, G. Bauer, B. Hall, and M. Paulitsch. TTEthernet: Time-Triggered Ethernet. In R. Obermaisser, editor, *Time-Triggered Communication*. CRC Press, Aug 2011.

[17] Y. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *IEEE Aerospace Applications Conference*, pages 293–307. IEEE, 1996.