

META II: PROBABILISTIC, COMPOSITIONAL, MULTI-DIMENSION MODEL-BASED VERIFICATION (PROMISE)

Grit Denker, Linda Briesemeister, Daniel Elenius, Shalini Ghosh, Ian Mason, Ashish Tiwari
SRI International

Devesh Bhatt, Haftay Hailu, Gabor Madl, Siamak Nikbin, Srivatsan Varadarajan
Honeywell Aerospace

Guenther Bauer, Wilfried Steiner
TTTech Computertechnik AG

Xenofon Koutsoukos, Tihamer Levendovszky
Vanderbilt University

OCTOBER 2011
Final Report



TABLE OF CONTENTS

Section	Page
LIST OF FIGURES	iii
LIST OF TABLES	vi
1. INTRODUCTION	1
2. SUMMARY	2
3. COMPOSITIONAL VERIFICATION OF HYBRID DYNAMICAL SYSTEMS	3
3.1 Introduction	3
3.2 Methods, Assumptions, and Procedures	3
3.3 Results and Discussion	4
3.3.1 HybridSAL Relational Abstractor	5
3.3.2 Generating Contracts for Open Systems	6
3.4 Conclusions	6
4. PROBABILISTIC FAILURE ANALYSIS	8
4.1 Introduction	8
4.2 Methods, Assumptions, and Procedures	10
4.2.1 Description of System Architecture	10
4.2.2 Failure Modes and Mitigation Strategies	15
4.3 Results and Discussion	16
4.3.1 Probabilistic Consistency Engine	16
4.3.2 PCE Models and Results	17
4.3.3 Interpretation of PCE Results: Comparison of Plots and Discussion	19
4.3.3 Theoretical Analysis	23
4.4 Conclusion	26
4.5 Recommendations	27
4.5.1 Voting and Hybrid Faults	27
4.5.2 Analysis of Curvatures of Plots	29
5. NETWORK INTEGRATION ANALYSIS FOR FAULT AND TIMING REQUIREMENTS	30
5.1 Introduction	30
5.1.1 High-Level Problem Description	31
5.1.2 System Requirements and Analysis Objectives	31
5.2 Methods, Assumptions, and Procedures	33
5.2.1 Network Architecture Abbreviations and Descriptions	33
5.2.2 Model of Fault Tolerance Constructs for Network Hardware Components	34
5.2.3 Fault Types	39
5.2.4 Probabilistic Fault Analysis: Failure Introduction and Propagation	44
5.2.5 Analysis Tool Chain Overview	47
5.2.6 Brake-by-wire Case Study	49
5.2.7 Equational Logic, Rewriting Logic, and Maude	52
5.3 Results and Discussion	54
5.3.1 Network Specification in Maude	54
5.3.2 BBW Network Specification in Maude	58
5.3.3 Fault Analysis in Maude	59

TABLE OF CONTENTS (CONCLUDED)

5.3.4 Performance Analysis Using Time-Triggered and Rate-Constrained Communication Paradigm	66
5.3.5 Integration of Tools in a Graphical User Interface.....	76
5.4 Conclusions	79
6. INTEGRATING VERIFICATION INTO EARLY DESIGN FLOW.....	80
6.1 Introduction	80
6.2 Methods, Assumptions, and Procedures.....	80
6.2.1 High-Level Integration Concepts.....	80
6.2.2 Integrating HybridSAL with Design Flow	82
6.2.3 Scalable, Multi-Component Static Verification Integrated with Design Flow..	86
6.3 Results and Discussion	90
6.3.1 Integrating HybridSAL with Design Flow	90
6.3.2 Scalable, Multi-Component Static Verification Integrated with Design Flow..	95
6.4 Conclusions	99
7. REFERENCES	101
LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS.....	104
APPENDIX 1.....	107

LIST OF FIGURES

Figure	Page
1. Property-preserving Abstraction and Compositionality are solutions to tackling the complexity of CPSs.....	3
2. Relational Abstraction is Compositional	4
3. HybridSAL Supports both Qualitative and Relational Abstraction.....	5
4. Verification Workflow Using HybridSAL Relational Abtractor	5
5. Aircraft Environmental Control System (ECS) Integrated Throughout the Airplane	8
6. Fourth-generation Aircraft ECS.....	9
7. Fifth-generation Aircraft Power and Thermal Management System (PTMS).....	9
8. ECS Schematic Diagram.....	11
9. Dual Cabin Air Compressor (CAC) Subsystem	11
10. Distributed Control System Architecture.....	12
11. Control Algorithm Architecture.....	12
12. Supervisory Control Chart	13
13. CAC Controller Algorithm Simulink Diagram	15
14. Modeled CAC Architecture	17
15. Component Failures and System Failures and Their Relationships	18
16. Comparison of System Failure Probabilities of Different CAC Architectures.....	20
17. Comparison of System Failure Probability of Different CAC Architectures (zoomed in version of Figure 16, in the x=0-2 range)	21
18. System Failure and Component Failure Probability Plots for two-cac architecture.....	22
19. SPIDER Architecture.....	28
20. System Centric View	31
21. Hierarchy of System Requirements	32
22. Analysis Trade-offs.....	33
23. Multicast	35
24. Path Redundancy	36
25. Standard- and High-integrity Host Tx and Rx	37
26. Standard and High integrity ES Tx	37
27. Standard and High-integrity ES Rx	38
28. Standard and High-integrity SW.....	39
29. Ethernet Frame Format	41
30. Introduction and Resolution of Inconsistency Errors	43
31. Introduction and Resolution of Inconsistency Errors	44
32. Fault Analysis	45
33. Network Architecture Tradeoff Analysis Tool Chain	48
34. Tool Chain Inputs	48
35. High-level Overview of BBW System Design	49
36. Detailed View of BBW System including replication (1/2) with dataflows from pedal to brakes in black and brakes to lights in blue	50
37. Detailed View of BBW System including replication (2/2) with dataflows of brake light problem in green and brakes engaged in black	51

LIST OF FIGURES (CONTINUED)

38. BBW Network Supporting Dataflows from pedal to the brakes, and from the brakes to the lights.....	59
39. BBW Network Topology	67
40. Virtual Link 1 of the BBW Case Study shown in dashed arrows.....	67
41. TT Schedule for VL 1 - VL 5 of the BBW Case Study	69
42. TT Schedule with Five Additional Frames (VL IDs 1-5), BBW VL IDs 1-5 are Translated to VL IDs 6-10	70
43. TT Schedule with 15 Additional Frames (VL IDs 1-15), BBW VL IDs 1-5 are Translated to VL IDs 16-20	70
44. TT Schedule with 35 Additional Frames (VL IDs 1-35), BBW VL IDs 1-5 are Translated to VL IDs 36-40	71
45. TT Schedule with 95 Additional Frames (VL IDs 1-95), BBW VL IDs 1-5 are Translated to VL IDs 96-100	72
46. Plot of the VL IDs (x axis) Versus their End-to-end Transmission Latencies (y axis)	72
47. Plot of the VL IDs (x-axis) Versus their End-to-end Transmission Latencies (y axis) Considering 15 Additional Dummy Frames	73
48. Plot of the VL IDs (x axis) Versus their End-to-end Transmission Latencies (y axis) Considering 35 Additional Dummy Frames	73
49. Plot of the VL IDs (x axis) Versus their End-to-end Transmission Latencies (y axis) Considering 95 Additional Dummy Frames	74
50. Graphical User Interface with Example.....	76
51. Analyzing Dataflow “bbw6”.....	78
52. Graph for Fault Propagation over Dataflow “bbw6”	78
53. Fault Propagation at Connection 74 of “bbw6”	79
54. The High-level Requirements Interface in CyPhy.....	81
55. Vanderbilt Verification Manager.....	82
56. CAC Control Overview	83
57. Flow Control Internals	84
58. Case Study in ESMoL.....	85
59. META Workflow	85
60. Sample Hybrid Automaton	86
61. Relative Costs of Detecting Errors in Various Life Cycle Phases.....	87
62. Multi-component Analysis of Design Properties.....	89
63. CyPhy - HiLiTE Integration Interfaces and Artifacts.....	90
64. Overall architecture of HybridSAL Integration.....	91
65. Sample Simulink/Stateflow Model	92
66. Hybrid Automata Model of the Saturation Block.....	93
67. Hybrid Automata Model of the CAC Case Study	94
68. CyPhy Captures Relationship between Simulink Models	95
69. CyPhy can Generate HiLiTE Inputs to Run Offline, or Directly Invoke HiLiTE.....	96
70. Verification Results are fed back into the design tool	96
71. Computation in a Model that is susceptible to Divide-by-Zero Overflow Defect.....	97
72. Model that Tunes a Parameter Impacting Defects in Other Models.....	97

LIST OF FIGURES (CONCLUDED)

73. FlightModesTest Example Model.....	98
74. ModesSelectionParameters Limits the Input ranges for FlightModesTest.....	99
75. Value of mode will be 3 only if the value of iSignal03 is greater than 2	99

LIST OF TABLES

Table	Page
1. Relational Abstraction of Various Classes of Dynamics can be automatically generated	4
2. Failure Types for Each Component, How Failures are Detected, and What Effect They Have and How They are Mitigated	15
3. Component Failure Probabilities for Different Prior Weights.....	23
4. Legend and Description of Abbreviations Used in Network Design	33
5. Failure Protection Mechanism of Components.....	46
6. Dataflows in BBW system.....	51
7. Truth Table for Example Full Disjunctive Normal Form.....	64

1. INTRODUCTION

The challenge in designing reliable and adaptable cyber-physical systems (CPSs) is rooted in the complexity of systems that combine a range of functions covering closed- and open-loop control modules. Control modules operate in the context of failure-prone sensors and actuators, integrated computing and hardware platforms, networked systems of systems, and physical environments. Verification methods for CPSs must address the broad range of mathematical models for computational and physical entities.

Verification techniques are computation-cost prohibitive if applied uniformly to each detail of a large, complex CPS design. It becomes necessary to decompose the verification space into appropriate abstractions, and trade-off specificity for tractability in principled ways, or approximate with probabilistic approaches in order to verify large-scale designs

The main technological verification barriers to realizing a “*correct-by-construction*” approach to large-scale CPSs are as follows:

1. Probabilistic certification tools and
2. A composition framework to calculate system-level probabilistic certificates from component-level certificates.

DARPA META PROMISE (**PRO**abilistic, Compositional **MultI**-Dimension Model-Ba**SEd** Verification) developed tools and a composition framework to address both challenges. The PROMISE team is led by SRI International and includes Honeywell International Inc., TTTech Computertechnik AG, and Vanderbilt University. PROMISE builds on the team's unique expertise that combines a long history of successful applications of formal verification tools with expertise and technology for aerospace safety design, certification and reliable and cost-effective manufacturing.

The PROMISE verification framework enables verifying CPS-level behavior and safety properties by using the composition of pre-certified components. By capturing the architectural assumptions and patterns in the reasoning framework that provide design-level certificates, both the depth and scalability of the analysis is significantly improved. This allows probabilistic certification of aircraft-level design with minimal real-world testing. It also enables verifying designs composed with heterogeneous architectural elements and networking configurations, including the verification of integrated compositions of software-control algorithms and electromechanical components, and the verification of mixed synchronous and asynchronous designs on the aircraft-level integrated networks.

2. SUMMARY

The main technological verification barriers to realizing a “*correct-by-construction*” approach to large-scale cyber-physical systems (CPSs) are as follows:

1. Probabilistic certification tools and
2. A composition framework to calculate system-level probabilistic certificates from component-level certificates.

DARPA META PROMISE (**PRO**abilistic, Compositional **MultI**-Dimension Model-Ba**SEd** Verification) developed tools and a composition framework to address both challenges. The PROMISE team is led by SRI International and includes Honeywell International Inc., TTTech Computertechnik AG, and Vanderbilt University.

The results concerning a compositional verification framework are presented in Section 3. COMPOSITIONAL VERIFICATION OF HYBRID DYNAMICAL SYSTEMS. Section 4. PROBABILISTIC FAILURE ANALYSIS and Section 5. NETWORK INTEGRATION ANALYSIS FOR FAULT AND TIMING REQUIREMENTS present our probabilistic verification tools. In Section 6. INTEGRATING VERIFICATION INTO EARLY DESIGN FLOW we summarize our results concerning the integration of two of our verification tools into the CyPhy design-flow environment.

3. COMPOSITIONAL VERIFICATION OF HYBRID DYNAMICAL SYSTEMS

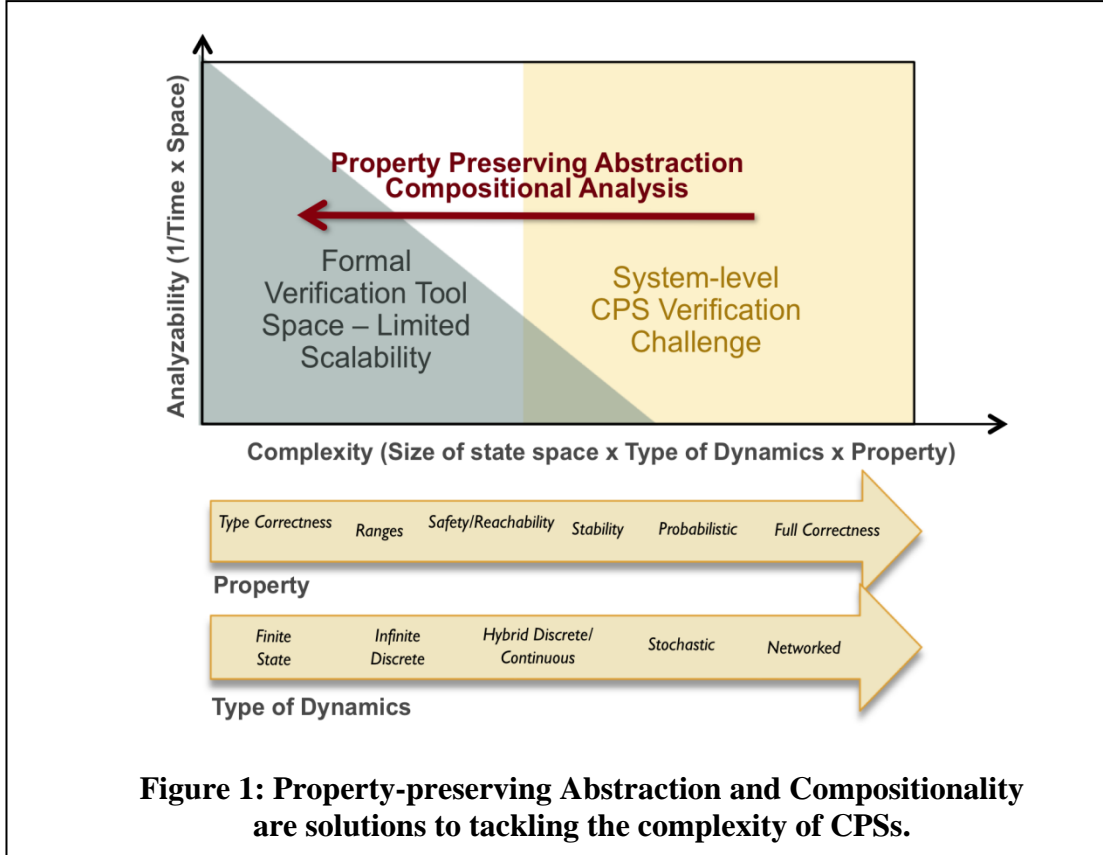
3.1 Introduction

Hybrid dynamical systems are formal models of complex systems that have both discrete and continuous behavior. It is well known that the problem of verifying hybrid systems for properties such as safety and stability is quite hard, both in theory and in practice. No automated, scalable, and compositional tools and techniques exist for formal verification of hybrid systems.

3.2 Methods, Assumptions, and Procedures

We developed the concept of relational abstractions of hybrid systems. A relational abstraction transforms a given hybrid system into a purely discrete transition system by summarizing the effect of the continuous evolution using relations. The state space of system and its discrete transitions are left unchanged. However, the differential equations describing the continuous dynamics (in each mode) are replaced by a relation between the initial values of the variables and final values of the variables. The abstract discrete system is an infinite-state system that can be analyzed using standard techniques for verifying systems such as k -induction and bounded model checking.

Relational abstractions can be constructed compositionally by abstracting each mode separately. Abstraction and compositionality are crucial for achieving scalability of verification (see Figure 1).



3.3 Results and Discussion

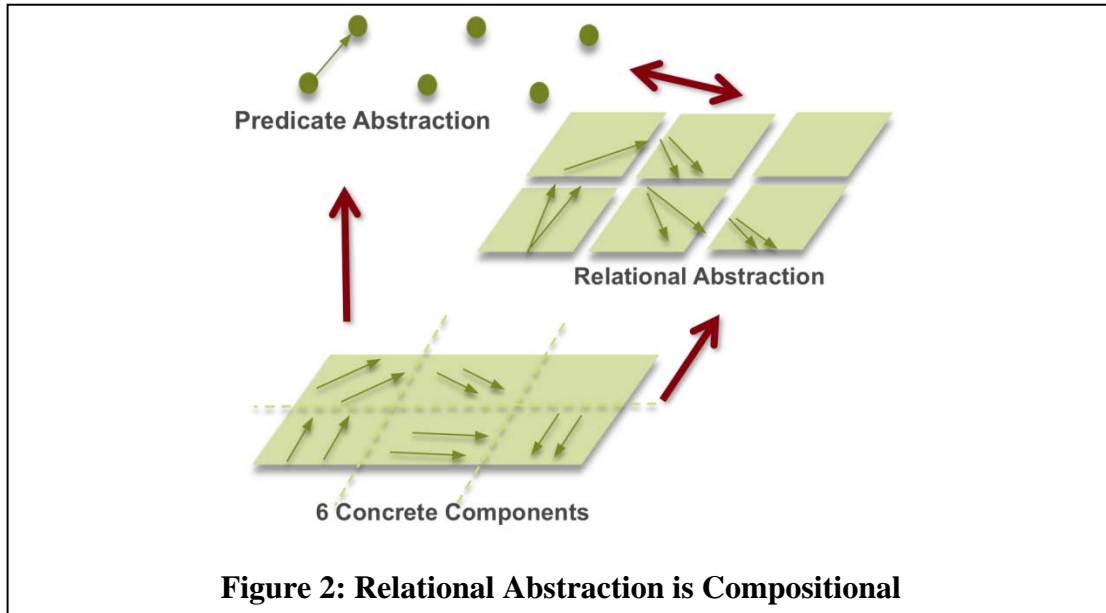
We have developed techniques for constructing high-quality relational abstractions. The details are technical and can be found in papers [1,2].

Table 1 illustrates the process of constructing relational abstraction using some very simple examples.

Table 1. Relational Abstraction of Various Classes of Dynamics can be automatically generated

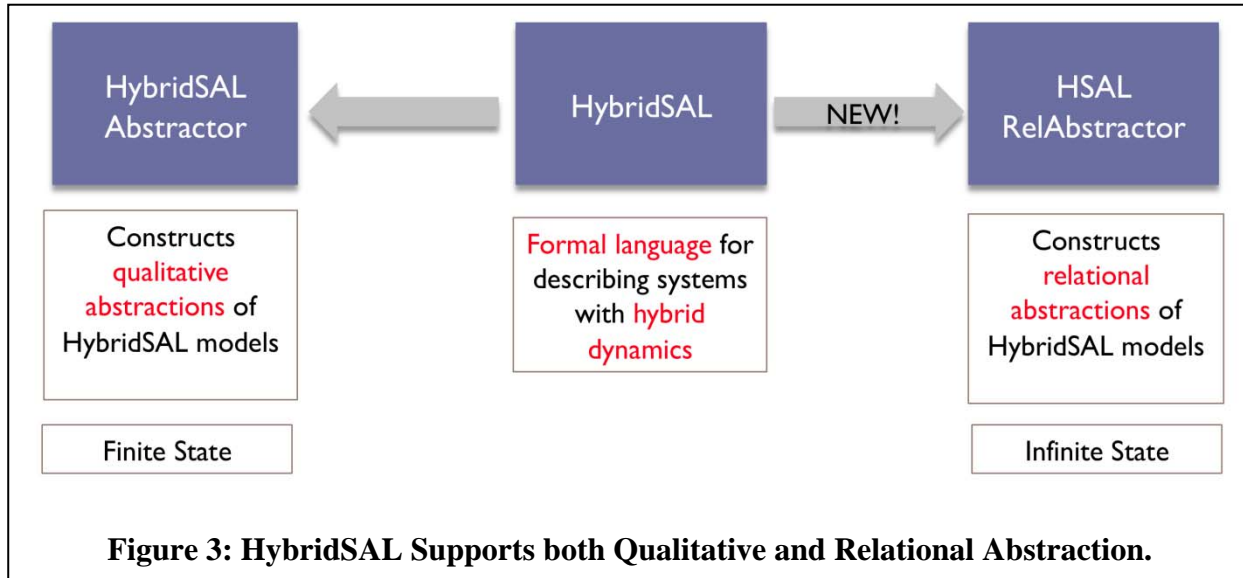
Dynamics	Relational Abstraction
$\dot{x} = 1, \dot{y} = 1$	$x' - x = y' - y$
$\dot{x} = 2, \dot{y} = 3$	$\frac{x' - x}{2} = \frac{y' - y}{3}$
$\dot{\vec{x}} = A\vec{x}$	$0 \leq p' \leq p \vee 0 \geq p' \geq p, p = \vec{c}^T \vec{x},$ \vec{c} Eigenvector of A^T corr. to neg. eigenvalue
$\dot{\vec{x}} = A\vec{x} + \vec{b}$...

Qualitative and predicate abstractions are techniques for abstracting a system that work by simplifying the state space of the system. Specifically, they reduce the state space of the system to a finite set of (qualitative) states defined by certain (qualitative) predicates (see Figure 2).



In contrast, relational abstraction does not simplify the state space, but only simplifies the presentation of the dynamics by replacing hard-to-analyze differential equations by discrete transitions. In principle, predicate and qualitative abstraction can be used on a relational

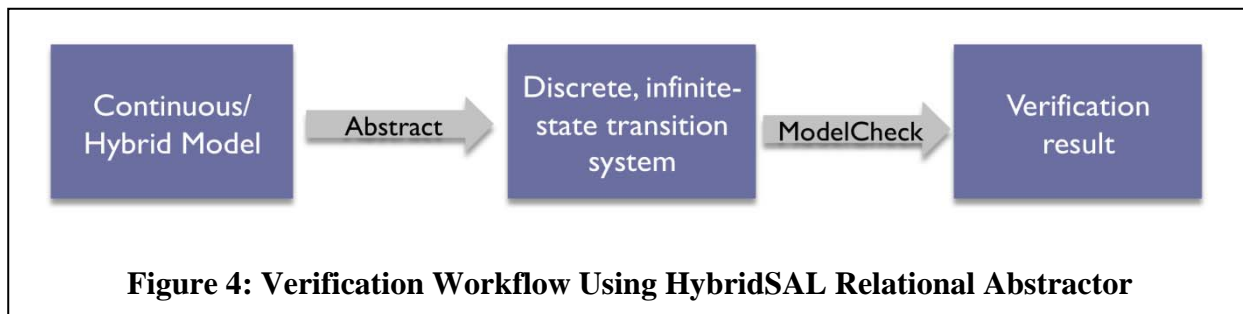
abstraction of a system to further approximate the system, if needed. The original HybridSAL tool implements qualitative abstraction. The new HybridSAL Relational Abtractor implements relational abstraction (see Figure 3).



3.3.1 HybridSAL Relational Abtractor

We developed a tool for analyzing hybrid systems based on constructing relational abstractions. The input to the tool is a specification of a hybrid system in the HybridSAL language. HybridSAL is an extension of SAL (Symbolic Analysis Laboratory), which is a language and a suite of tools for modeling and analyzing discrete state transition systems. HybridSAL extends SAL by allowing specification of continuous dynamics in the form of differential equations.

The verification workflow for using the HybridSAL relational abtractor is shown in Figure 4. The input model of the continuous or hybrid system is in HybridSAL. The relational abtractor tool creates a SAL file that contains the relational abstraction of the input file. Subsequently, the SAL file can be analyzed for safety properties by running a SAL model checking tool, such as SAL infinite bounded checker or SAL k-induction prover.



We also developed an optional front end that generates HybridSAL models automatically from Simulink via the Vanderbilt CyPhy environment.

3.3.2 Generating Contracts for Open Systems

We developed an approach for generating contracts for open systems. This "certificate-based approach" is based on the observation that verification is the same as searching for a certificate of correctness. Hence, in this approach, verification is achieved by directly searching for certificates of correctness (such as inductive invariants and Lyapunov functions) of systems.

The certificate-based approaches are turning out to be particularly effective in proving deep properties of complex systems. Certificate-based methods work by fixing a template for the "certificate of correctness", and casting the verification problem as that of finding an appropriate instantiation of the template. This search is accomplished using numeric or symbolic solvers that reason about arithmetic constraints in the theory of reals.

For example, for verifying stability, the user can provide a template for the Lyapunov function (say a quadratic Lyapunov function) and the solver can find a concrete quadratic function that proves stability of the given system. The most commonly used solvers include those for sums-of-squares (SOS) programming and real quantifier elimination. Numeric approaches, such as SOS programming, have two limitations: first, they may give incorrect answers, and second, they have their limitations when solving Boolean combination of constraints. We use symbolic solvers in the form of real quantifier elimination. We extended the open source symbolic nonlinear solver called Reduce/Redlog and integrated it with another open source solver called Qepcad. The integrated solver is available for download from the Reduce/Redlog website; see also [3]. The technical details of the extension, and further examples of using this extension to apply the certificate-based approach for verification and synthesis of hybrid systems can be found in [4].

The certificate-based approach for verification can also be used to generate contracts, or assume-guarantee pairs for open systems. We illustrated the certificate-based approach for generating contracts by considering a simple PI (proportional integral) controller. We formulated the absolute always eventual region stability problem as the finding of a class of plants that a generic PI controller can always eventually stabilize. We used real quantifier elimination methods to solve this absolute region stability problem and find contracts for PI controller that guarantee region stability. The class of plants found in our solution includes nonlinear and switched plant models. For details, see the papers [4,5].

3.4 Conclusions

Compositional verification of complex systems is a challenging problem. We developed two new approaches to enable compositional analysis of continuous and hybrid dynamical systems: a certificate-based approach and an approach based on computing relational abstractions.

The certificate-based approach directly searches for certificates of correctness using constraint solving. Certificates vary depending on the property. This approach can be used to create assume-guarantee contracts for open components. It requires templates for the form of certificates and form of assumptions. Relational abstraction replaces the differential equations in the system description by sound abstract discrete transitions, thus enabling application of discrete verification tools. The HybridSAL relational abstractor tool automatically computes such

abstractions for linear hybrid systems. For nonlinear systems, both approaches require nonlinear constraint solvers. We improved an existing state-of-the-art symbolic nonlinear solver, but scalability continues to remain a challenge.

4. PROBABILISTIC FAILURE ANALYSIS

4.1 Introduction

In illustrating a new probabilistic failure-analysis capability and tool, we provide examples of verification requirements to exercise the tool. Our use case is from the area of Aircraft Environmental Control Systems (ECS). We describe the electromechanical case study and provide requirements for the verification of control and fault tolerance properties.

Figure 5 illustrates the key subsystems that interface with the ECS. The actual interfaces differ depending on whether the ECS pack utilizes bleed air or not. The main heat loads cooled by the ECS are: cabin, cockpit, avionics and other ancillary loads. The cabin and cockpit are air-cooled and the air temperature is independently controlled; avionics are liquid-cooled. The ECS also cools its components: motors and motor controllers. The motor controllers are liquid cooled, and the motors are air-cooled. There is a single temperature control in the current design with only one temperature sensor that measures the return temperature of the coolant. The single-temperature control provides several advantages, including a simple, lightweight design. The trade-off is there is no independent cooling control for each controller or avionics box.

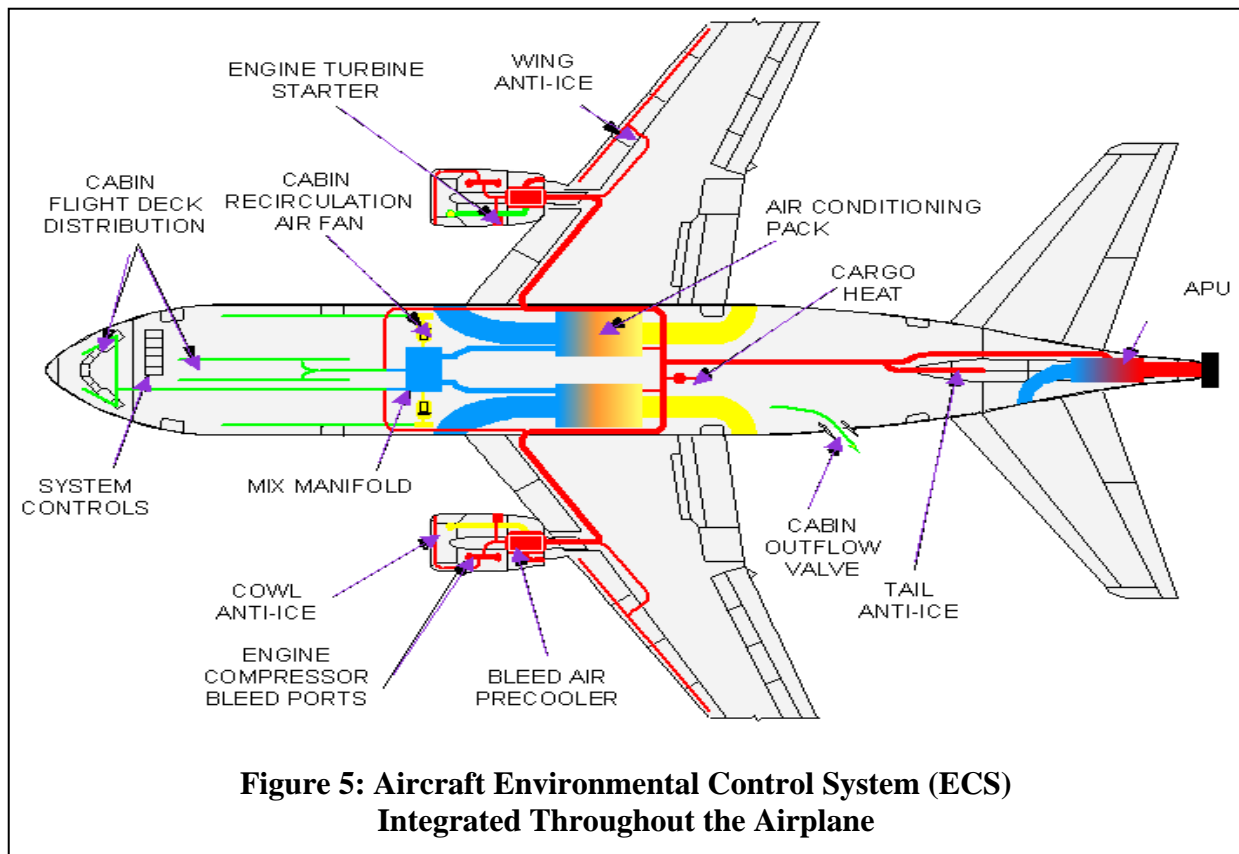


Figure 6 and Figure 7 detail the fourth- and fifth- generation ECSs.

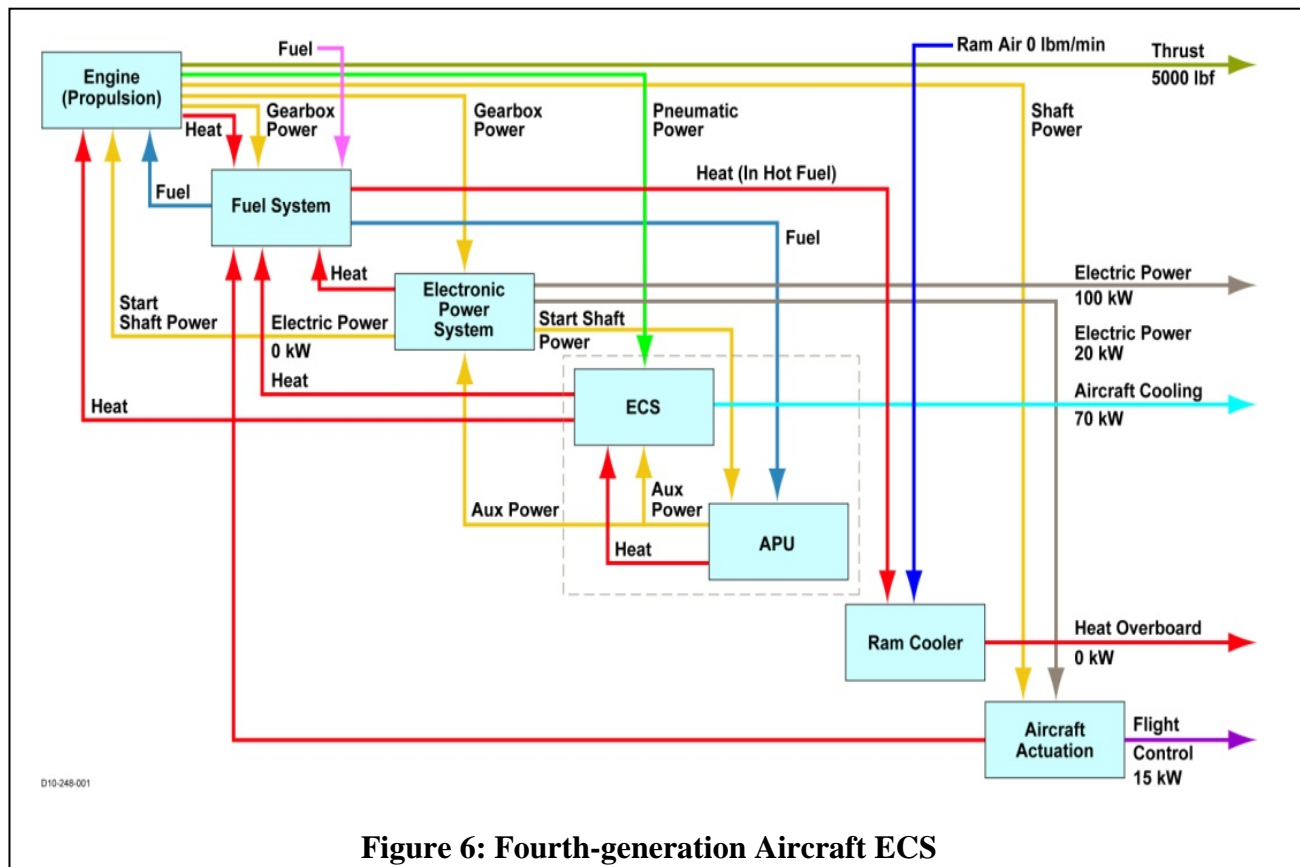


Figure 6: Fourth-generation Aircraft ECS

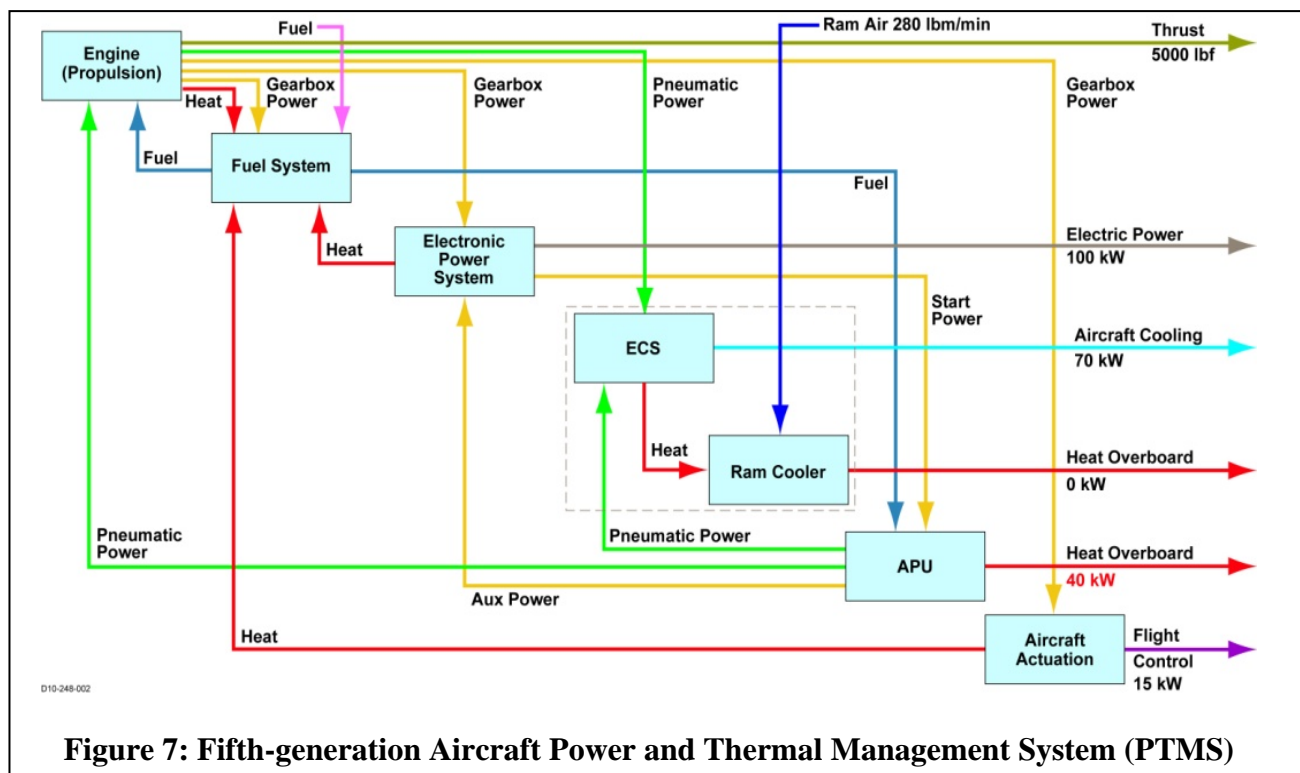


Figure 7: Fifth-generation Aircraft Power and Thermal Management System (PTMS)

The case study will exercise a mix of different types of components and verification aspects, including (i) software, electro-mechanical and pneumatic components, (ii) hybrid control, and (iii) a mix of scenarios of failures and degraded modes of operation.

Component interactions are captured through higher-level operational modes and scenarios, demonstrating the use of compositional verification and multi-dimensional trade-off analysis.

The system description includes controller elements, plant, operational scenarios and potential failure modes.

Figure 8 shows the schematic diagram of the next-generation environment control system that would be a part of the “more electric architecture” concept used in new aircraft. The dual CAC subsystem shown by the dashed circle in the figure is used to deliver fresh air to the cabin/cockpit and provide cabin pressurization. The subsystems outside the dual CAC condition the air before sending it to the cabin/cockpit, and cool the aircraft avionics and ECS motor controllers using a special coolant. The dual CAC subsystem is the main focus of the case study – outlined by the dashed line in Figure 8 – because it possesses all the attributes of a cyber-physical system such as control algorithms, interaction with the physical environment, data bus communication, fault recovery, and degraded performance operation.

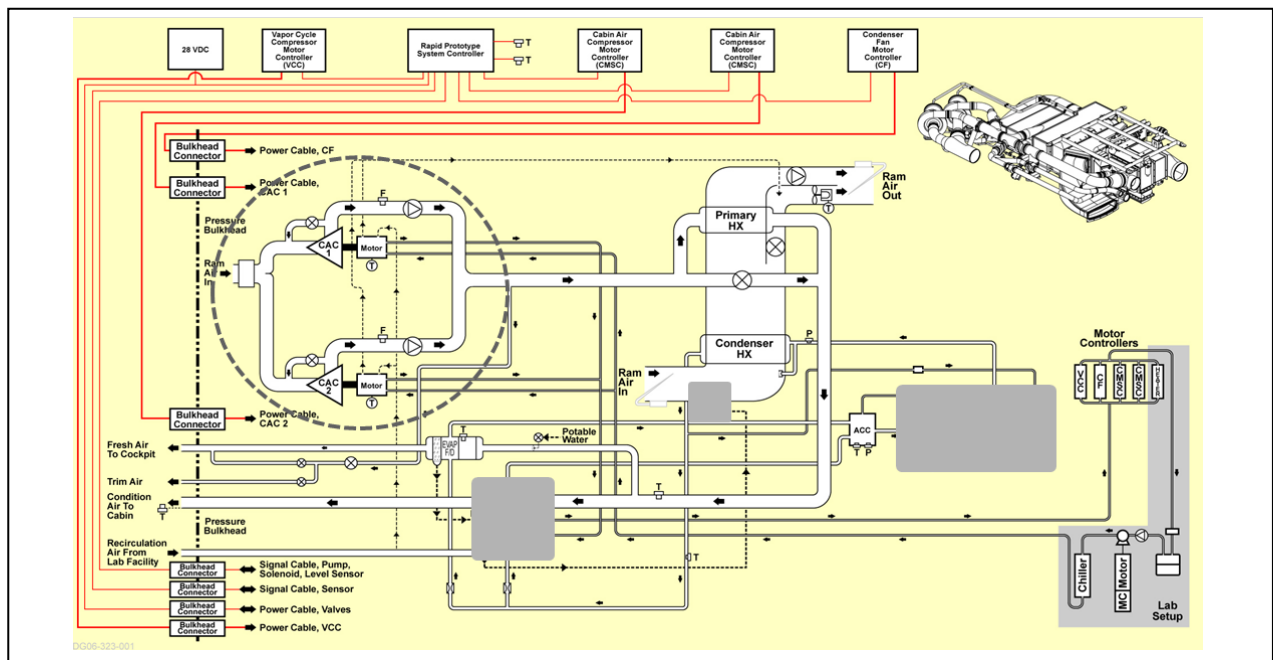


Figure 8: ECS Schematic Diagram

Figure 9 is a schematic diagram of the dual CAC subsystem. The main functions of this subsystem are to provide sufficient fresh air to the crew and passengers and to maintain cabin and cockpit pressurization. Under normal operation, both compressors should work to provide the same airflow rate, though not necessarily at the same speed. The flow control loops of the system controller regulate the airflow rate to a predefined flow set point. The supervisory control of the system controller determines the flow set point depending on whether the dual CAC subsystem exhibits a failure in one of the compressors. In the case of a single CAC failure, the flow demand for the second CAC is increased and the system is run in a degraded mode to provide the minimum air supply.

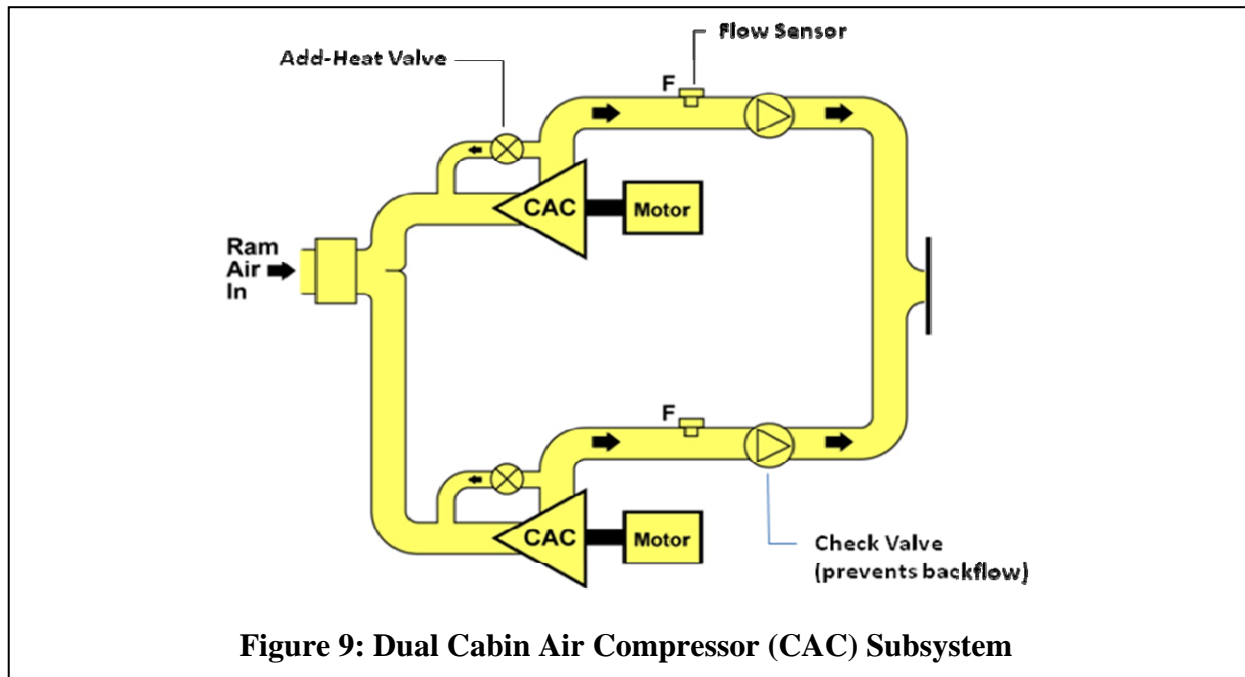
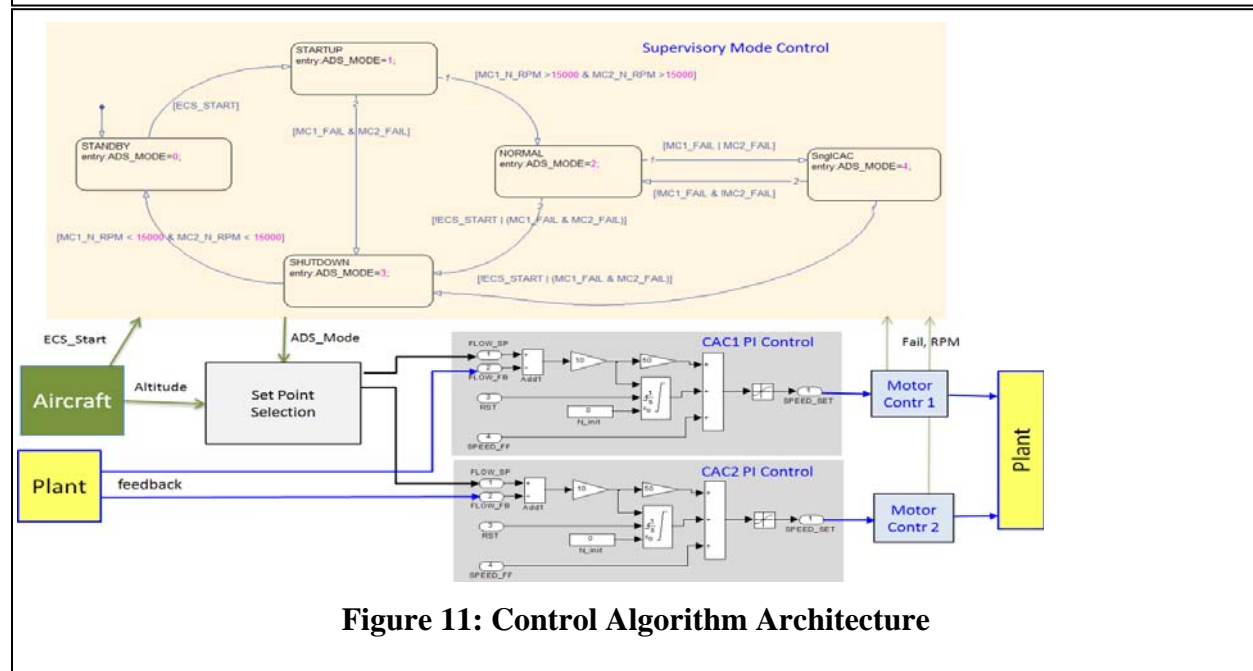
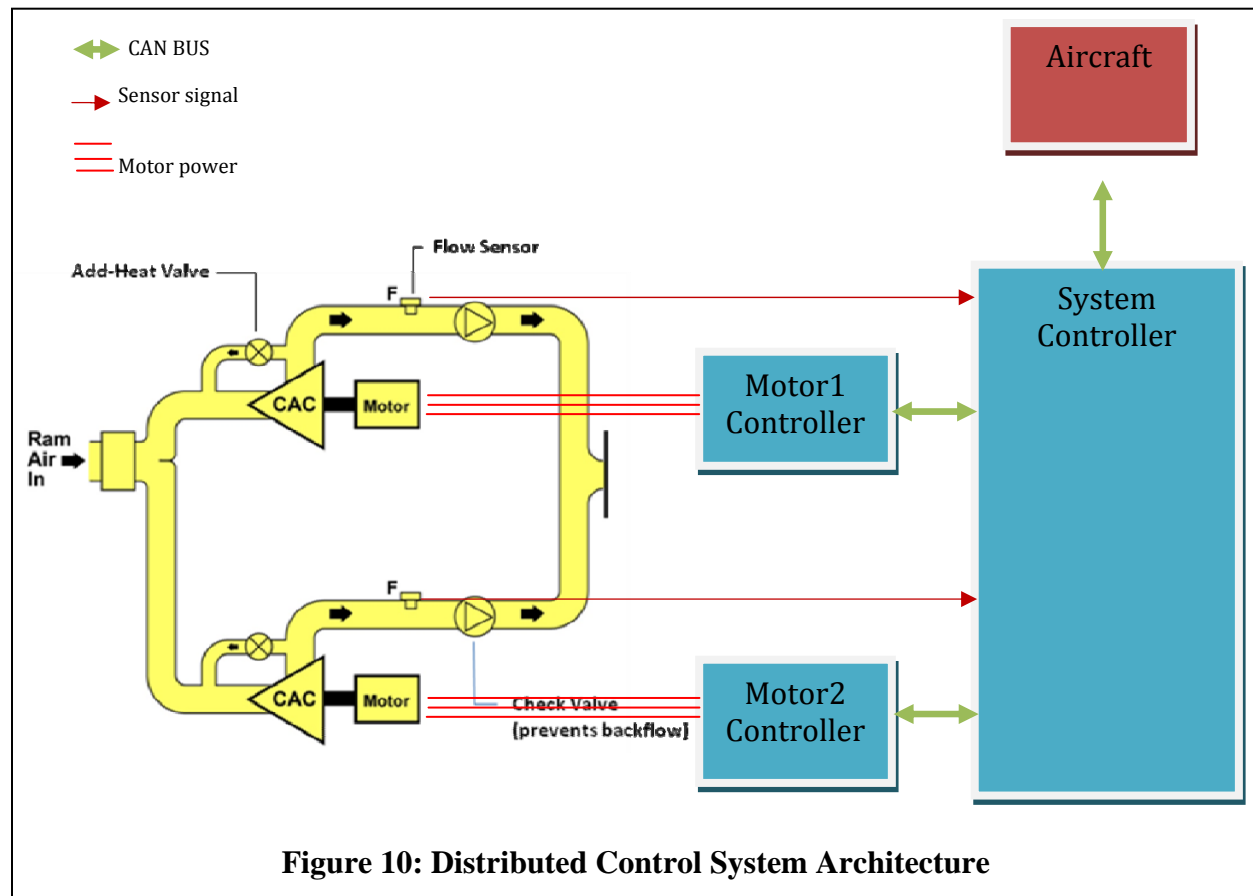


Figure 9: Dual Cabin Air Compressor (CAC) Subsystem

3.2.1.2 System Control Elements and Algorithms

Figure 10 illustrates the distributed control system architecture for the dual CAC. In this architecture, the individual motor controllers communicate to the system controller through a controller area network (CAN) bus. Individual motor controllers are responsible for regulating the speed of their respective motors based on the speed set point received from the system controller. In addition to receiving speed set points, each motor controller receives start/stop signals from the system controller through the same data bus. The individual motor controllers send back many signals to the system controller, including actual speed, power, motor temperature, controller temperature, and all fault conditions as determined by each motor controller. The system controller monitors these signals to determine the overall state of the system. The system controller is responsible for the overall performance of the system, while the motor controllers are for the individual motors.

Figure 11 illustrates the control algorithm architecture and flow of signals. The system controller consists of supervisory mode control, set point selection, and CAC 1 & 2 PI control modules.



The system controller manages the system operation using different control software components. These are classified as supervisory controls and primary control loops. In Figure 11, the system controller consists of supervisory mode control, set point selection, and CAC 1 & 2 PI control modules.

The supervisory controls component is designed utilizing a discrete event (DE) or finite state machine (FSM)-based approach. The main control loops are based on continuous process monitoring (sampled at the appropriate rate) of physical signals, such as airflow rate in this case. The primary control loops are designed according to the inherent physical relationship between the sensors and actuators.

The **supervisor control** specifies the operation modes of the system. For the dual CAC subsystem, the main modes are **STANDBY**, **STARTUP**, **NORMAL**, **SingleCAC**, and **SHUTDOWN**. The **STANDBY** mode is the default mode and is entered upon receipt of electrical power to the controller. These modes are modeled using Stateflow as shown in Figure 12. The transitional conditions are not shown in the figure.

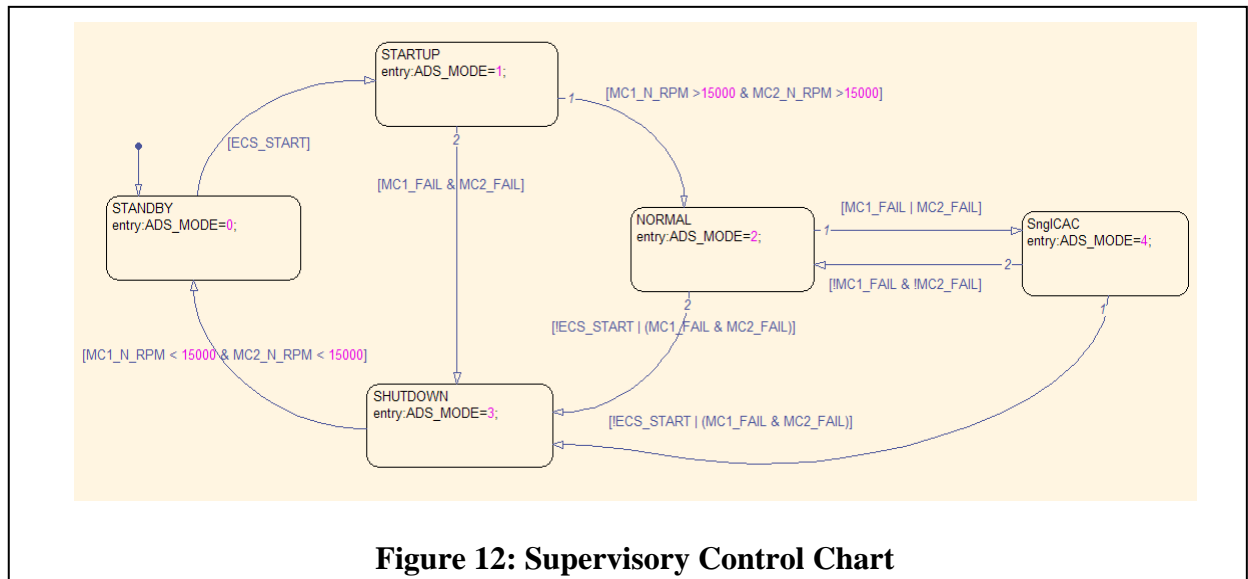


Figure 12: Supervisory Control Chart

- **STANDBY:** entered upon receipt of electrical power to the controller and when the software starts running. This mode is also entered from the **SHUTDOWN** mode upon completion of system shutdown. The mode is exited to the **STARTUP** mode upon receipt of an ECS_Start command from the aircraft. During this mode, all signals are monitored but no commands are issued. The system controller is also communicating with all the other controllers (motor controllers) and the aircraft.
- **STARTUP:** entered from **STANDBY** upon receipt of an ECS_Start command from the aircraft. This mode is exited to the **NORMAL** mode upon successful completion of startup. If the startup is not successful, it exits to **SHUTDOWN** mode. During this mode, the system controller sends a motor enable signal and initial speed to all motor controllers. It also monitors the status signals from the motor controllers.

- ***NORMAL***: entered from *STARTUP* upon successful completion of startup. This mode is exited to either *SingleCAC* mode or *SHUTDOWN* mode. The mode exits to *SingleCAC* when failure of the other CAC is detected. The mode exits to *SHUTDOWN* mode when it receives an ECS_Stop command from the aircraft or when it detects severe fault from any of the components. During this mode, the system controller sends all the speed set points to the motor controllers and also determines the flow set point to the primary control loops. The system is running with all closed loop controls active.
- ***SingleCAC***: entered from *NORMAL* upon detection of signal CAC failure. The mode is exited to either *NORMAL* mode or *SHUTDOWN* mode. The mode exits to *NORMAL* when the fault in the other CAC is removed and it is running. The mode exits to *SHUTDOWN* mode when it receives an ECS_Stop command from the aircraft or when it detects severe fault from any of the components. During this mode, the system controller sends the speed set point to the motor controller and also determines the flow set point to the primary control loop.
- ***SHUTDOWN***: entered from *STARTUP*, *NORMAL*, and *SingleCAC* modes based on their exit rules. The mode is exited to the *STANDBY* mode upon successful completion of shutdown. During this mode, the system controller sends commands to motor controllers to reduce the speed of motors to the minimum limit followed by the disable command.

The **primary control loops** are used to dynamically control the system operation by modulating all available effectors (actuators). For the dual CAC subsystem, the flow control loops dynamically control the airflow rate by comparing the flow set point (coming from the supervisory control) and actual flow as measured by the flow sensors.

The primary control loops for the CAC flow control are implemented as a hybrid open loop and closed loop control with respect to the flow sensor. The hybrid implementation gives faster response with reduced transient. The open loop control is the feed-forward term scheduled as a function of flight altitude. The closed loop is the feedback term that closes about the airflow error. The airflow error is the difference between the desired flow rate and the actual flow rate (as measured by the airflow sensors).

The type of fault that disrupts the closed loop control is failure in a flow sensor. The most critical fault in a flow sensor is an open circuit and short circuit of the electrical terminals of the sensor.

Figure 13 shows a simplified version of the Simulink diagram for CAC controller. The actual speed command to the compressor is based on the air flow error from the desired value (the feedback term) and an ideal command (SPEED_FF) based on the known correlations. Speed Feed Forward (Speed_FF) is used to change the output motor speed in a look-ahead mode (e.g., if we are climbing, set the value higher for higher altitude).

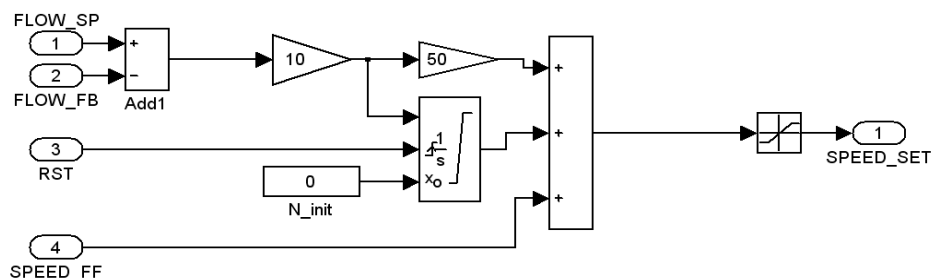


Figure 13: CAC Controller Algorithm Simulink Diagram

The dual CAC subsystem contains two **motor controllers** (one for each compressor) that communicate with the system controller through the CAN bus. Each motor controller modulates the three-phase current from the inverter to control the speed of the motor to the desired value (set point). The speed set point is determined by the system controller and communicated to motor controllers. The motor controllers also monitor the temperature of both the motors and motor controllers and communicate the data to the system controller.

4.2.2 Failure Modes and Mitigation Strategies

Table 2 specifies failure types for each component. For example, the flow sensor fails, giving a zero reading. This results in the CAC generating too much pressure. The supervisory control (SC) detects this situation, determines the speed of the affected CAC to be zero, and uses the other CAC.

Table 2. Failure Types for Each Component, How Failures are Detected, and What Effect They Have and How They are Mitigated

Component	Failure Type	Detection	Effects, Mitigation, Relationships
CAN Bus	Message loss	SC detects	If message loss to one CAC then shut that down. If both CACs unreachable then system fail. CAN Bus failures are temporary => bring back online within flight.
Motor Controller	Overheating (Temp sensor)	MC detects, notifies SC	Cooling system design tradeoff – single cooling control (weighted average of temp) vs. independent controls. Increased load on cooling system => increased probability of other failures
	Contactor failure	MC detects, notifies SC	Motor shutdown, increase load (flow) of other CAC => increased probability of other CAC failure with time
Motor	Overheating	MC detects, notifies SC	Similar to above.
	Stalling	MC detects, notifies SC	Temporary motor shutdown, attempt to bring back on line within flight.
Turbomach./ Compressor	Overheating	MC detects, notifies SC	Similar to above.
Flow Sensor	Reading outside range limits	SC detects	Control design tradeoff – use other flow sensor and apply same speed to both CACs vs. accommodate for changes in flow from the two CACs (also, manufacturing artifacts)

A verification goal derived from this situation is that the SC model will work as designed and will do the control changes in time to prevent damage due to overpressurization and without loss of cabin pressure. A possible design trade-off for this situation is to use triple-redundant flow sensors with validity bit and mid-vale select voting in software. This does not require any mitigation by supervisory control. A variation of this scenario is when the flow sensor is stuck in high or unstable (with high and low values alternating).

4.3 Results and Discussion

4.3.1 Probabilistic Consistency Engine

SRI's Probabilistic Consistency Engine (PCE) is a tool that performs efficient inference with probabilistic first-order rules, using the framework of Markov Logic Networks (MLN).

An MLN [6] is a statistical relational model to formalize first-order logic with probabilities. In an MLN, all random variables are Boolean and all feature functions are Boolean formulas. The formulas in the MLN have weights that are associated with their probabilities – given a knowledge base (conjunction of formulas), the weights on the formulas are used to compute the associated model probability in the KB. One can then compute the marginal probability of a given formula F as the probability aggregate over the models where F evaluates to true. In typical use, an MLN is used to infer the marginal probability distributions for (output) random variables and formulas based on the distribution for the input random variables, over the space of models defined by the formulas and their corresponding probabilities. The MC-SAT inference algorithm computes these marginal probabilities by efficiently averaging the probabilities over a sequence of models.

PCE [7] uses multi-sorted first-order logic for describing a network of formulas and weights to build probabilistic relational models. PCE provides a language for representing MLNs and implements an optimized version of the MC-SAT algorithm of Poon and Domingos for probabilistic inference [8], to compute marginal probabilities of formulas. PCE uses a combination of simulated annealing/SampleSAT and WalkSAT to implement the Markov Chain Monte Carlo (MCMC). The object language of PCE consists of sorts, literals, constants, and formulas. Sorts encode type information, rules are represented as universally quantified formulas in both the observable and hidden predicates along with their associated weights, while facts are represented as grounded instances of literals and formulas. Various probabilistic inference problems can be represented as MLNs in the form of facts and weighted and unweighted rules. The weight of a model is given by the weight of the formulas that hold in the model, and the probability of a formula is the normalized sum of the weights of the models in which the formula holds. PCE outputs the marginal probabilities for the atomic formulas and any query formulas. PCE is order sorted since it also has subsorts.

4.3.2 PCE Models and Results

Figure 14 illustrates the components of the CAC architecture that we are modeling with formal methods.

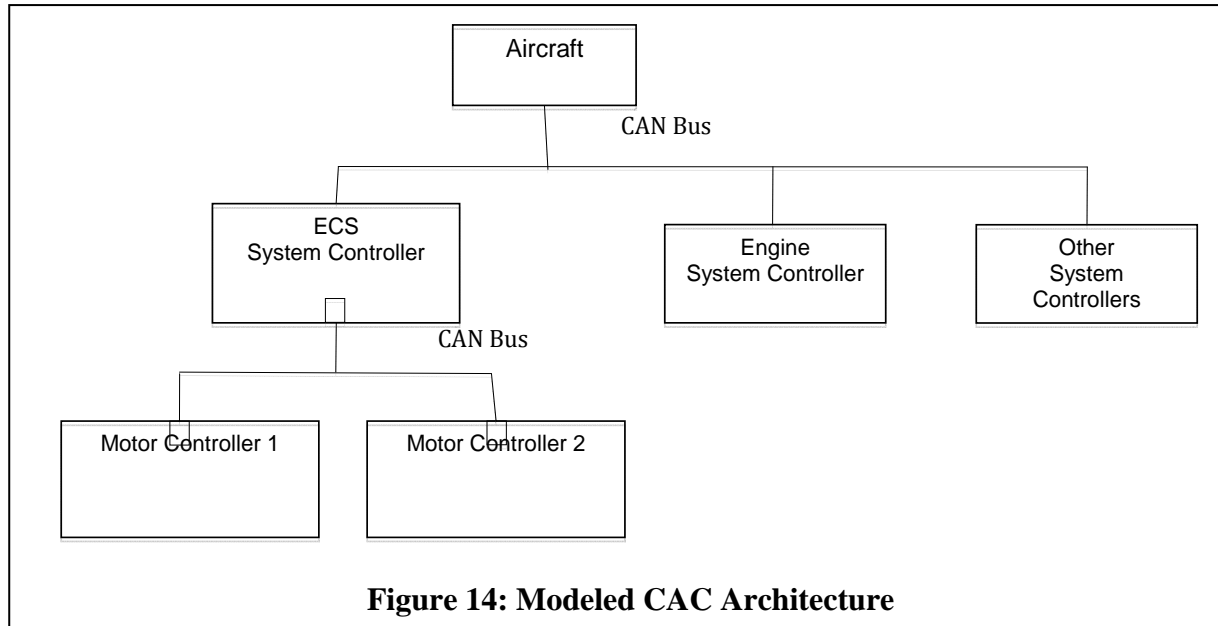
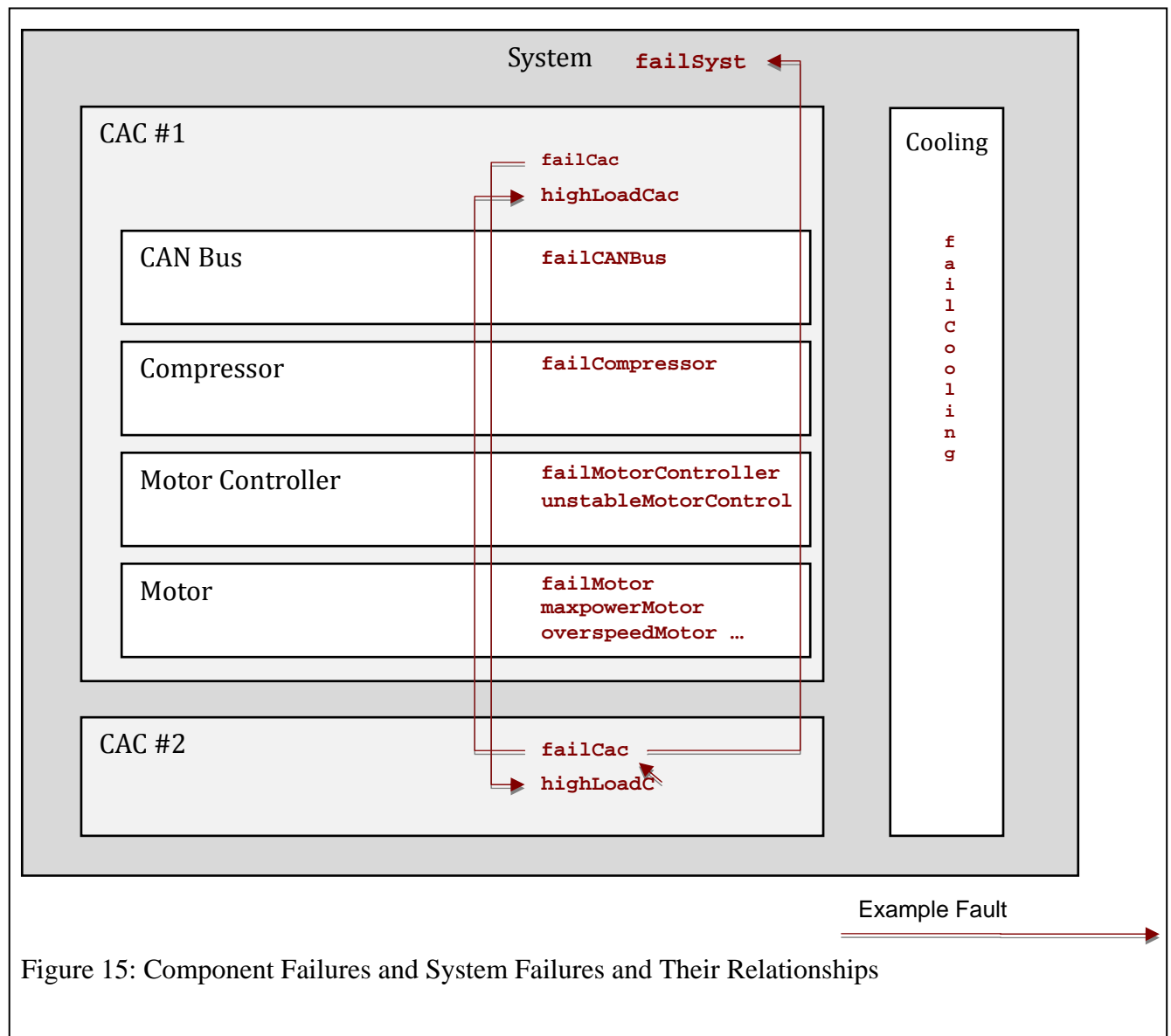


Figure 15 gives example failure modes for each component. For instance, for the Motor Controller component, we model two failure modes: failMotorController and unstableMotorControl. For each CAC in the dual CAC system we model the failure modes failCAC and highLoadCac. The red arrows in the figure illustrate how the failures affect each other. If CAC #1 fails, then CAC #2 is put into highLoadCac mode and vice versa. When both of the CACs fail, the system will fail. We model these relationships in more detail below.



Let us consider the following different architectures for the CAC models:

- A1. One-cac: no redundancy, system has a single CAC
- A2. Two-cac: two cacs connected in parallel, system functions properly if any one CAC is functiona,
- A3. Three-cac: three cacs connected in parallel, system functions properly if any one CAC is functional
- A4. Voting-three-cac: three CACs connected via voting logic, system fails if at least two out of three CACs fail
- A5. Three-cac-highload: three CACs connected in parallel, high load is put on third CAC if two CACs fail, system functions properly if any one cac is functional

The architectures A2, A3, A4, A5 the corresponding Markov-Logic Networks (MLNs) (two-cac1-complete.mln, three-cac1-complete.mln, voting-three-cac-complete.mln, and three-cac-hl-complete.mln) and the corresponding trace outputs on running MCSAT on each MLN (two-cac1-complete.mln.output, three-cac1-complete.mln.output, voting-three-cac-complete.mln.output, and three-cac-hl-complete.mln.output) are provided in the Appendix 1. We also show the dual-cac PCE models – cac-model1-demo.pcein is similar to the model in A2, while cac-model2-demo.pcein is the more fine-grained model of the two-cac system. A4 is similar to the traditional voting model, in which we enforce reliability through voting. Note that A5 is different from that – in A5, we model the fact that if two of the three cacs fail, then the third cac has a high load and fails with a higher probability. That is, in A5, two failed cacs in a three-cac system "hammer" the third cac.

4.3.3 Interpretation of PCE Results: Comparison of Plots and Discussion

The results of running MC-SAT on the MLNs seem reasonable (from the output files). A quick calculation shows that when the priors of the component failures are on the order of 10^{-2} , the system failure probability for the voting three-cac will be ${}^3C_2 \times 10^{-4} + {}^3C_3 \times 10^{-6} \sim 10^{-4}$; this is what we are getting in the results trace. For the regular three-cac model, the system failure probability will be on the order of 10^{-6} if we assume iid. Note that in the results trace, we see that the actual failure probability is somewhat higher than that. One intuitive explanation is that the failures of the CACs are not iid – the failure of one CAC model influences the failure of another via the highLoad rule.

We also did some sanity checking of the three-cac-highload model in A5 – basically instantiating one CAC failure, two CAC failures, and three CAC failures via the MLN facts, and seeing what happens to the system failure probabilities in each case. The output traces are given in the Appendix (three-cac-hl-complete.output.1fail, three-cac-hl-complete.output.2fail, and three-cac-hl-complete.output.3fail). The results are as expected – the system failure probability gets progressively higher as more CACs fail.

A) Plot 1

For each of these architectures, we ran PCE to get the system failure probabilities for different values of the component failure priors. We then generated the plot of the curves for the five different architectures shown in Figure 16. The plot is in negative log-log scale (considering \log_{10}) – a value x on the horizontal axis corresponds to a component prior probability of failure of 10^{-x} , while a value y on the vertical axis corresponds to the system marginal probability of failure of 10^{-y} . The part of the plot under the $x=2$ value on the x -axis is difficult to view in Figure 16, for which we plot a zoomed-in version of Figure 16 with x range 0-2 in Figure 17.

Here are a few interesting observations about the plots in Figure 16:

1. The two-cac and the voting-three-cac model curves cross at around the component prior probability of 10^{-1} , whereas the voting-three-cac and three-cac-highload model curves cross around the component prior probability of 10^{-2} . Something to explore here further is if these crossover points are a function of the weight of the highload rule for the different MLNs (set to 0.1 for now). For example, here is the highload rule for two-cac:

add [c, d] failCac(c) and ~failCac(d) and (c ~= d) => highLoadCac(d) 0.1

We would like to further analyze theoretically why these crossovers happen for different pairs of CAC architectures at the particular points on the plots in Figure 16. Some initial symbolic analysis to that effect is given in this section. But someone using trade-off decisions based on these curves can decide what architecture to choose for certain component prior probabilities or desired system failure probabilities.

2. The negative log-log plots for the architectures other than one-cac are all linear with a slope > 1 in the active region before the curves flatten out in the saturation region. This corresponds to the super-linear power-law relation in the active region:

$$\text{system_failure_prob} = (\text{component_failure_prior_prob})^m, \text{ with } m > 1$$

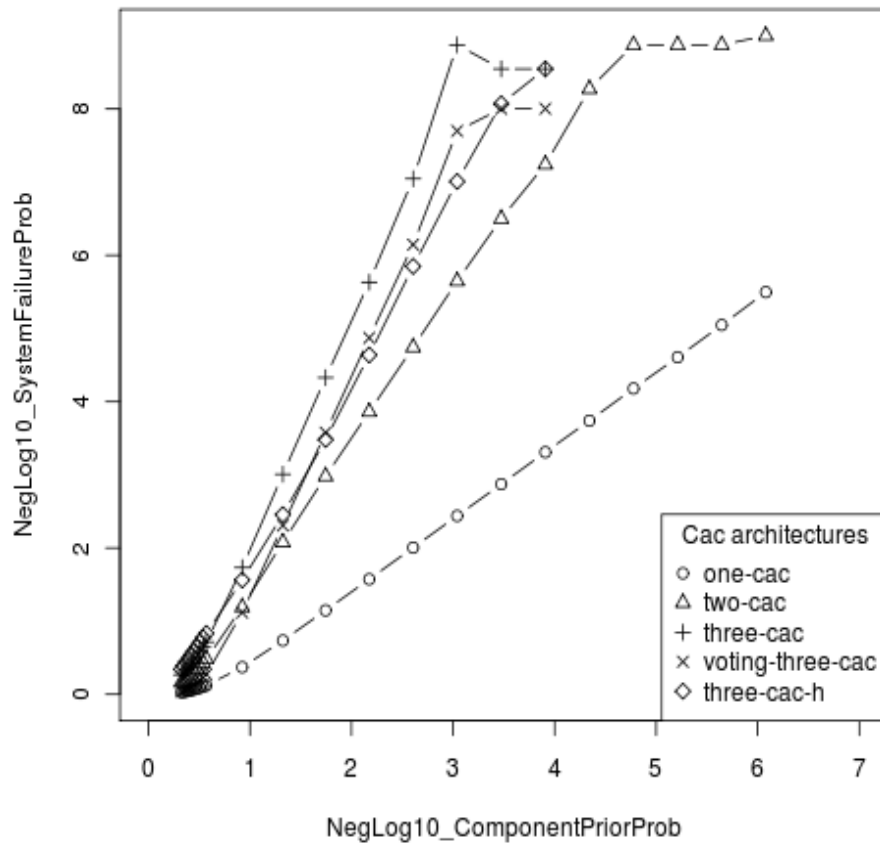


Figure 16: Comparison of System Failure Probabilities of Different CAC Architectures

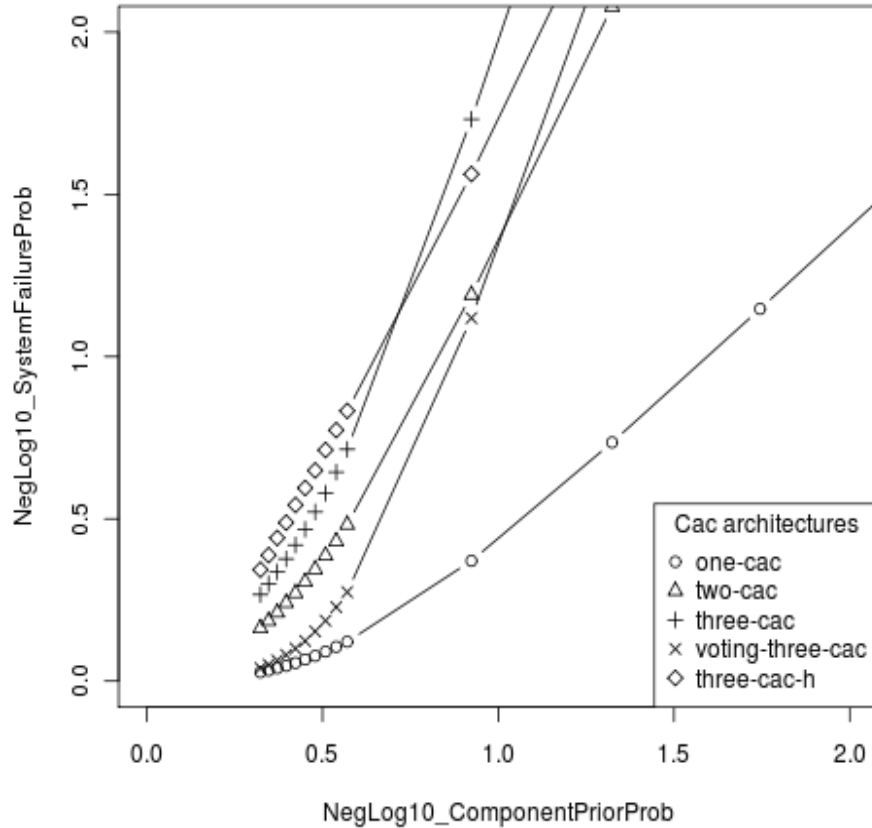


Figure 17: Comparison of System Failure Probability of Different CAC Architectures (zoomed in version of Figure 16, in the x=0-2 range)

For one-cac, the active region has the same form, with $m < 1$. Coming up with a closed-form relation between the system failure probability and the component failure prior probabilities would be an interesting result to try and derive for these architectures. However, we would have to come up with a closed-form relation that would explain the linear relation in the negative log-log scale in the active region as shown in Figure 16, as well as the curvature for those plots in Figure 17.

3. The plots continue linearly for all architectures until reaching a saturation region, where each curve flattens out or dips. These points correspond to very low prior probabilities of failures for the components, and the resulting system marginal probability of failure reveals very low values. In these low-probability saturation regions, the Markov chain in MC-SAT probably does not converge to the stationary distribution. Here are the different numbers of samples for which we ran MC-SAT in different ranges for system probability of failure:
 - 500 million samples when the system failure probability is 10^{-3} or more
 - 1.5 billion samples when the system failure probability is between 10^{-6} and 10^{-3}
 - 5 billion samples when the system failure probability is between 10^{-6} and 10^{-8}

Below the system failure probability of 10^{-8} , the curves reach the saturation zone – we will have to run PCE for more than 5 billion samples in that range. This also indicates that there is work to be done in making PCE and related graphical model MCMC sampling-based inference techniques more efficient for low probability estimates, so that we do not have to run sampling for a very large number of iterations to estimate very low probability events. Each run of MCSAT for the current models takes about 1 hour for 1 billion samples (for the low probability events).

B) Plot 2

Figure 18 shows the failure probabilities of the system as well as the various components of the two-cac model. The y axis plots $-\log_{10}$ of the marginal failure probability for the system and the marginal failure probabilities of the different components. The x axis plots $-\log_{10}$ of the prior probability of failure of the different components.

The data used in plotting is also shown in Table 3, since some of the component-level values overlap in the plots in Figure 18 and may not be clearly discernible.

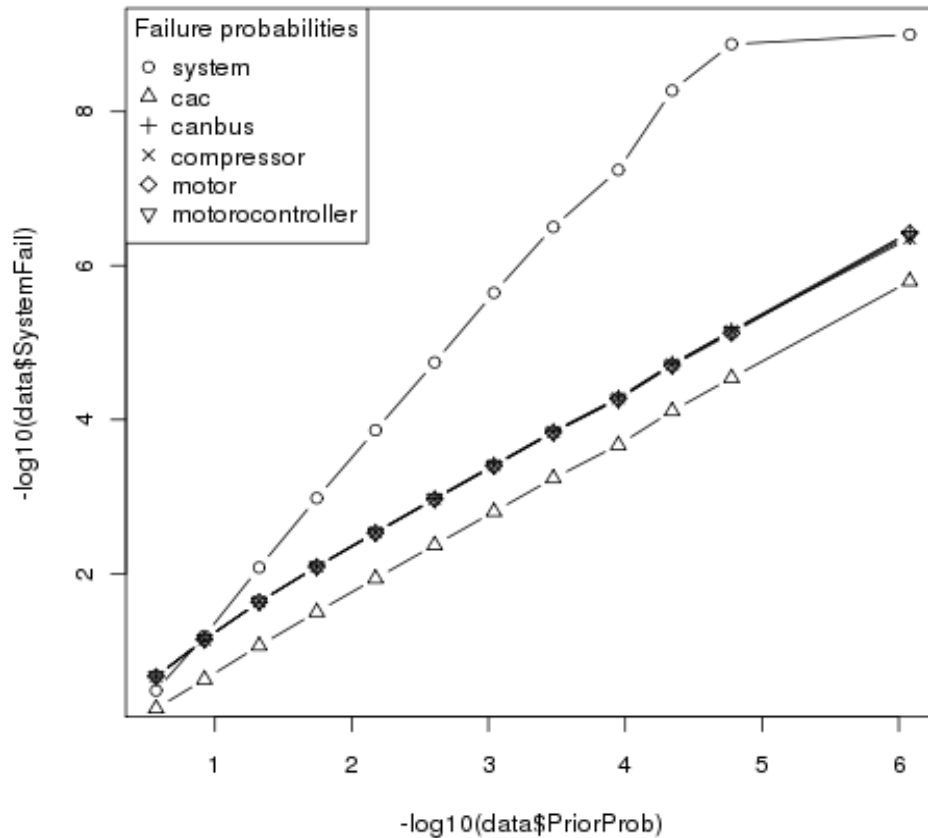


Figure 18: System Failure and Component Failure Probability Plots for two-cac architecture

Table 3. Component Failure Probabilities for Different Prior Weights

PriorWt	PriorProb	SystemFail	CacFail	CanBusFail	CompressorFail	MotorFail	MotorControllerFail
-1	2.69E-01	3.29E-01	5.65E-01	2.17E-01	2.16E-01	2.17E-01	2.16E-01
-2	1.19E-01	6.45E-02	2.37E-01	7.25E-02	7.18E-02	7.20E-02	7.17E-02
-3	4.74E-02	8.34E-03	8.64E-02	2.33E-02	2.32E-02	2.33E-02	2.33E-02
-4	1.80E-02	1.05E-03	3.16E-02	8.05E-03	8.10E-03	8.16E-03	8.14E-03
-5	6.69E-03	1.37E-04	1.16E-02	2.89E-03	2.92E-03	2.96E-03	2.95E-03
-6	2.47E-03	1.81E-05	4.27E-03	1.05E-03	1.07E-03	1.08E-03	1.08E-03
-7	9.11E-04	2.25E-06	1.57E-03	3.86E-04	3.92E-04	4.01E-04	3.97E-04
-8	3.35E-04	3.16E-07	5.79E-04	1.42E-04	1.44E-04	1.47E-04	1.46E-04
-9	1.12E-04	5.73E-08	2.14E-04	5.21E-05	5.29E-05	5.41E-05	5.47E-05
-10	4.54E-05	5.33E-09	7.75E-05	1.89E-05	1.91E-05	1.99E-05	1.96E-05
-11	1.67E-05	1.33E-09	2.88E-05	7.00E-06	7.09E-06	7.60E-06	7.14E-06

4.3.3 Theoretical Analysis

Here, we do a theoretical analysis of some of the empirical results we observed earlier.

Analysis of Crossover Points in Figure 16.

Using iid assumptions (i.e., ignoring the highLoad rule), we were able to derive some theoretical (symbolic) results about crossovers of the plots of the following architectures:

(A) 1-cac versus voting three-cac

Let ps = probability of system failure, pc = probability cac failure, p = probability of cac component failure (for each of the four components of cac).

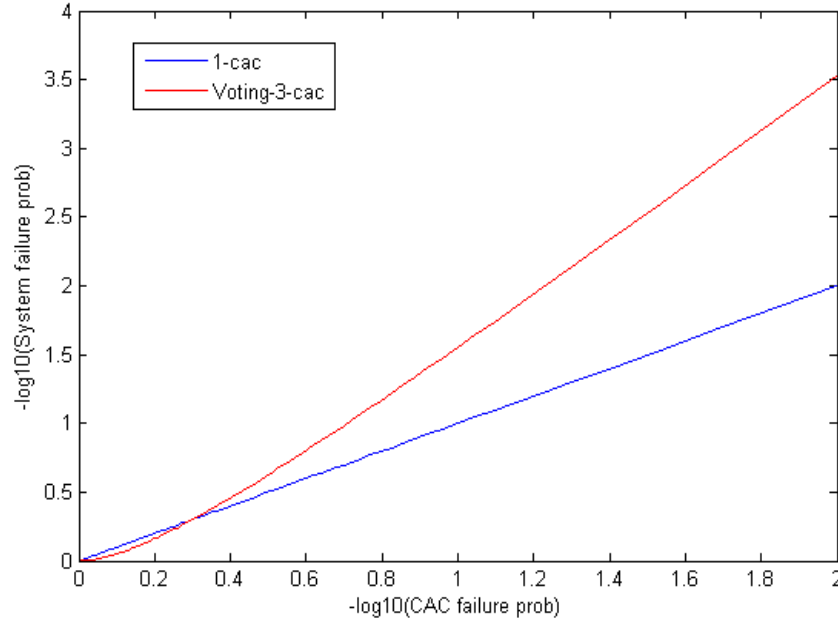
$$ps_{\text{voting}} = {}^3C_2 * (1-pc) * pc^2 + {}^3C_3 pc^3 \quad (1)$$

$$ps_{\text{1cac}} = pc \quad (2)$$

$$\text{They cross when } pc = 3pc^2 - 2pc^3 \quad (3)$$

$$\Rightarrow pc = 0.5 \quad (4)$$

$$\Rightarrow -\log_{10}(pc) = 0.301. \quad (5)$$



Now, the CAC behaves normally if all of its four components behave normally. So, if probability of component failure is p , then we have:

$$1-pc = (1-p)^4 \quad (6)$$

$$\Rightarrow p = 1 - (1-pc)^{0.25} = 1 - (1-0.301)^{0.25} = 0.0856 \quad (7)$$

\Rightarrow the crossover point in the negative log10-log10 scale of system failure probability versus component failure probability would be around 0.09 on the x axis.

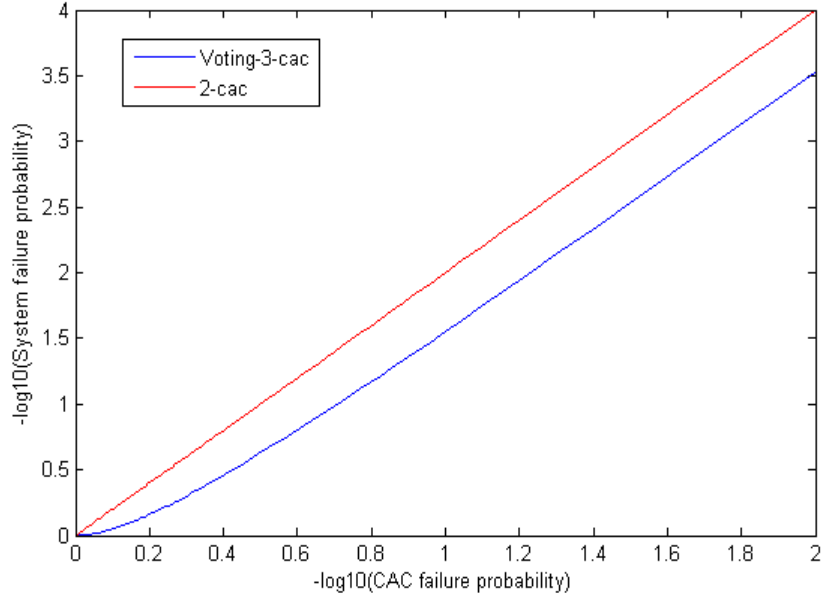
In the actual results from PCE simulation (Figure 16) we see that the plots for the one-cac and voting-three-cac models cross close to this value. So, we are getting results in PCE simulation (with the highLoad rule for the voting-3-cac, no iid assumption) in the same ballpark as the symbolic derivation with the iid assumption. Intuitively, this seems to make sense, since, in this case, the iid assumption for the voting-3-cac may not give us too much divergence from the actual crossover point with 1-cac as 1-cac does not have the highLoad rule.

(B) 2-cac versus voting-3-cac

$$ps_voting = {}^3C_2 * (1-pc) * pc^2 + {}^3C_3 pc^3 \quad (8)$$

$$ps_2cac = pc^2 \quad (9)$$

The solution for the crossover point ($ps_voting = ps_2cac$) is $pc = 1$, i.e., $\log(pc) = 0$, as verified by the following illustration:



However, we do see that the curves cross at a value $p_c < 1$ if we model using PCE (not iid anymore, due to the highLoad rule). So, here the iid assumption is quite divergent from the actual system behavior, which perhaps demonstrates the value of doing PCE-type modeling in this domain to get around iid assumptions, i.e., being able to model non-iid faults like conditional faults.

Analysis of Curvature in Figure 17.

Let p_s = probability of system failure, p_c = probability of cac failure, p = probability of CAC component failure (for each of the four components of CAC), where $p_c = 1 - (1-p)^4$.

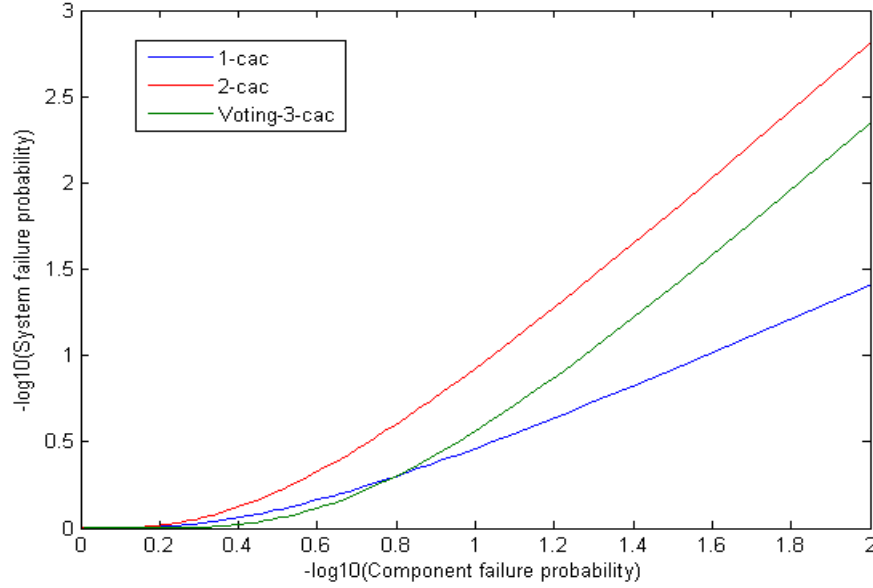
Therefore, for the one-cac, two-cac, and voting-three-cac architectures, the probabilities of system failures are given by:

$$p_{s_1cac} = p_c = 1 - (1-p)^4 \quad (10)$$

$$p_{s_2cac} = p_c^2 = (1 - (1-p)^4)^2 \quad (11)$$

$$p_{s_voting} = {}^3C_2 * p_c^2 (1 - p_c) + {}^3C_3 p_c^3 = {}^3C_2 * (1 - (1-p)^4)^2 * (1-p)^4 + {}^3C_3 (1 - (1-p)^4)^3 \quad (12)$$

When we plot the above three equations in the negative log scale, we get the following result:



Note that we get the curvatures in the higher probability ranges and linearity in lower probability ranges from the theoretical calculation, similar to the PCE simulations in Figure 17. However, in Figure 17, two-cac crosses voting-three-cac, whereas here (in the theoretical iid-based calculation) we get one-cac crosses voting-three-cac. This could be due to the iid assumption, as we are not taking into account the interaction caused by the highLoad rule in voting-three-cac.

Another interesting analysis is one in which there straight lines in the low-probability ranges and curves in the high-probability ranges. Here is an analysis of that for one-cac.

$$ps = 1 - (1-p)^4 \quad (13)$$

$$\Rightarrow \log ps \quad (14)$$

$$= \log(1 - (1-p)^4) = \log(4p - 6p^2 + 4p^3 - p^4) = \log(4p) + \log(1 - 1.5p + p^2 - 0.25p^3) \quad (15)$$

$$\sim \log(4p) + (-1.5p + p^2 - 0.25p^3), \text{ when } p \ll 1 \quad (16)$$

When $p \ll 1$, the residual second term in the above equation can be ignored compared to the first. This gives us a straight line in the “log(ps), log(p)” space with a slope close to 1.

For larger values of p , the residual second term in the above equation cannot be ignored compared to the first – the residual gives the curvature in the plot for higher values of p .

4.4 Conclusion

Fault analysis using probabilistic methods is a promising approach for CPSs. In order to derive error probabilities at the system level, it is often necessary to know component-level error probabilities, and this is seldom practical. Our approach has the advantage that we can do trade-off analyses to calculate component-level probabilities for which the required system-level probabilities hold. This provides input regarding component requirements that guide the CPS design.

4.5 Recommendations

4.5.1 Voting and Hybrid Faults

In general, when components are more likely to fail than not, we do not want to take a vote. Most of the components are likely to fail, giving well below ($1 - 10^{-1}$) in reliability. So, if we take a vote of them, the vote winner will “fail” more often than the right answer. In this case, it is better to simply go with one CAC (the one-cac model) and just live with the native probability of failure.

This is where hybrid voting can be very useful. With the hybrid fault model of OMH or OMH-File Transfer Protocol (FTP) or Scalable Processor-Independent Design for Extended Reliability (SPIDER), we can reason about extremely high (benign or manifest) fault rates well beyond 10^{-1} and still have a system that performs well. In particular, the hybrid voting system should outperform a single CAC everywhere along those curves.

Practically, what we will get is a system in which no number of manifest faults (e.g., powered-down units) could confuse one or more good units. The good units should “know” they are good and have real values and not get outvoted by “fail” even if there are hundreds of manifest failures. The additional complexity in the voter to handle this hybrid voting is extremely small – subject matter experts in several domains (spacecraft design, submarine design) say that in practice, this small overhead is worth it for their respective cases.

In future work, we would like to investigate hybrid faults in PCE, using a SPIDER model. Here are some properties of SPIDER:

- It is a fault-tolerant architecture developed at NASA Langley.
- SPIDER is a family of fault-tolerant, reconfigurable architectures providing mechanisms for integrating inter-dependent applications of differing criticalities.
- Applications communicate via a reliable optical bus (ROBUS) – a TDMA (time-division multiple access) bus providing basic fault-tolerant mechanisms of clock synchronization, group membership, and interactive consistency.
- Fault-tolerance mechanisms have been formally proved correct using the PVS theorem proving system.
- SPIDER architecture offers a robust foundation for safety and mission assurance.

SPIDER (see Figure 19) is a family of general-purpose fault-tolerant architectures useful for recovering from transient failures. An instance of the SPIDER architecture consists of several Processing Elements (PE) communicating over a ROBUS. There are two types of Fault Containment Regions (FCRs) internal to the ROBUS – the BIUs and the Redundancy Management Units (RMUs). The Bus Interface Units (BIUs) provide an interface to the PEs, while the RMUs provide the necessary replication for Byzantine fault tolerance [9].

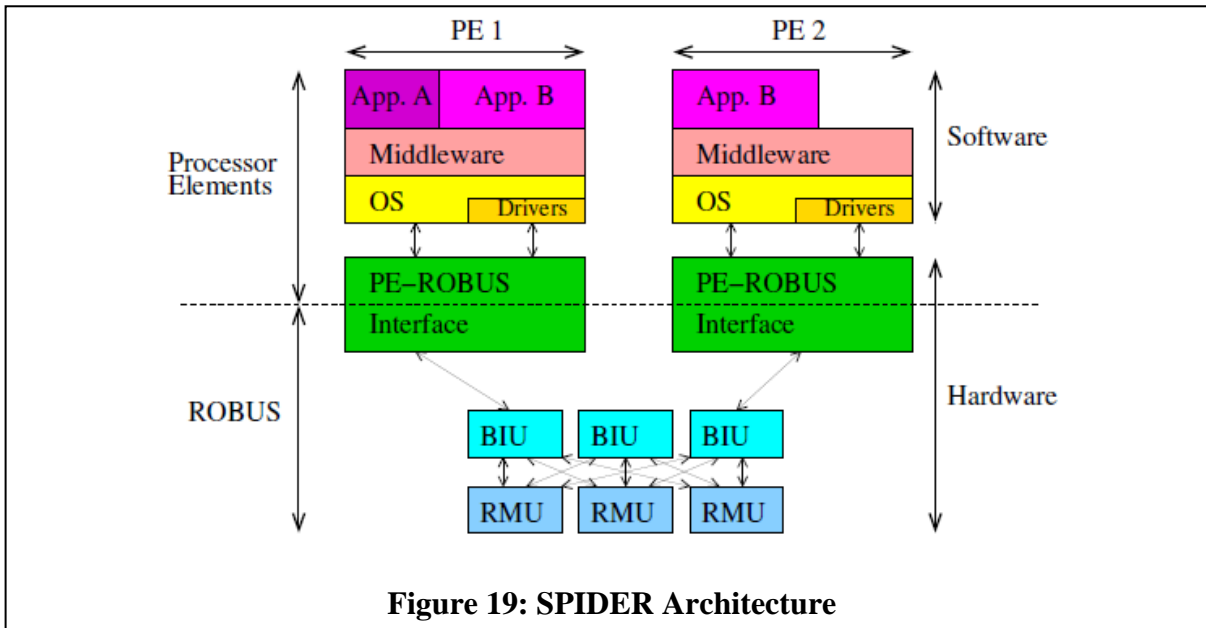


Figure 19: SPIDER Architecture

Here are some advantages of SPIDER over its predecessors:

- Recovers from combinations of faults including asymmetric faults and has optimizations for benign faults, e.g., fail-stop or symmetric faults
- Provides a generic interface to connect to any kind of processor, so that one is not locked into a particular semiconductor technology
- Supports a wide range of fault-masking strategies such as dual-dual, triple modular redundancy (TMR), or even multi-stage threshold voting
- Allows processing nodes to be grouped to provide differing degrees of fault tolerance for different applications
- Provides dynamic reconfiguration where less important functions can be eliminated so that critical functions continue to operate
- Provides a mechanism for dealing with transient faults to reduce the impact of temporarily faulty components
- Scalable to accommodate large networks of input/output resources

Here is what we can do with PCE modeling in SPIDER:

- Model the static part of the SPIDER architecture using PCE
- Compute the probability of system failure when the individual components fail.
- Start with Byzantine (asymmetric) faults, and then model hybrid faults: Byzantine faults + Manifest faults + Symmetric faults
- Consider different complexity of communication systems – start with the redundancy management unit considered part of the BIU, and later model faults in the BIU and the redundancy management unit (RMU) separately.

4.5.2 Analysis of Curvatures of Plots

We would like to symbolically explain the curvature of the system probability versus component probability plots that we observe in Figure 16 for other architectures (e.g., three-cac-highload).

5. NETWORK INTEGRATION ANALYSIS FOR FAULT AND TIMING REQUIREMENTS

5.1 Introduction

The broad goal of this subtask is to analyze the design of systems and network architecture in the context of failures and performance and thereby augment verification and validation artifacts through composable analysis tools and evaluation methodology. The rationale is that such composable analysis tools, used early in the design process and systematically throughout the different stages of that process, would be key enablers in reducing design costs, cost of change due to varying requirements, and verification and validation (V&V) costs, and would also increase reusability of network design with appropriate design changes for different ground vehicles and aircrafts.

Given this objective, we provide an approach to analyze network system architectures in terms of performance (latency/jitter/timing properties, bandwidth, buffer and other resources) and failure (fault modes, propagation) in a conjoint manner. In the past, such system analyses were done along one of those two dimensions in isolation due to the complexity of analyzing them together. The rationale behind this task is the insight that the trade-offs in the network architecture design space can be comprehensively explored only when both dimensions are systematically explored in conjunction. Further, both fault and performance requirements are irrevocably linked such that any design change in one dimension impacts the other. The linkage between these two dimensions is not always completely understood and not formally characterized or analyzed. This has led to point solutions in the network architecture design space whereby, though an individual network architecture is designed to satisfy the requirements known at the time of initial design, any subsequent changes to the requirements in the later stage forced the design to be reworked ground up. Further, the need to integrate more applications on the same network or to leverage new hardware/software enhanced capabilities or adhere to limitations of the available technology since the time of initial design or even reusing the same network design for different vehicles/aircrafts with different requirements forced network designers to go to the drawing board and begin all over again with a new design. The analysis framework described here is designed to reduce network change cost and redesign cost by composing the network architectures to be systematically analyzed in terms of system requirements and thereby enabling network reuse as well as reuse of analysis artifacts (and thereby V&V artifacts).

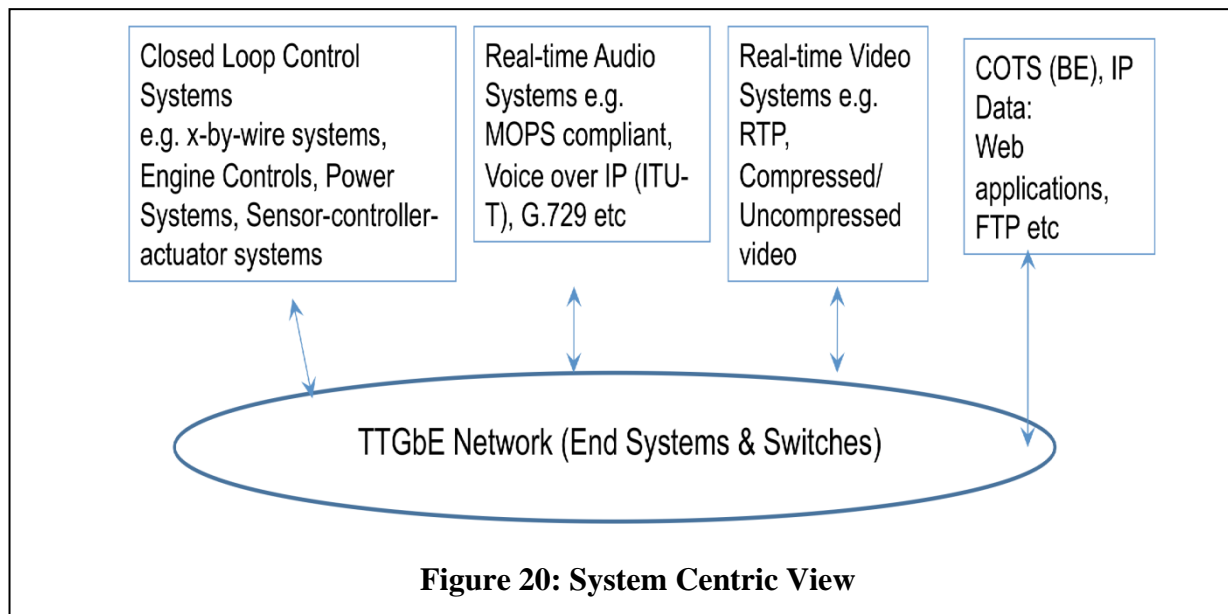
Here, we develop formal specifications of failure modes and their propagation based on the underlying protection mechanisms existing in the network hardware and software that prevent such failures, systematic evaluation and exploring trade-offs of system design with a set of analysis tools, dataflow and fault-tolerance modeling (availability, integrity), synchronization overheads characterization, hardware versus software trade-offs (implementing these services in hardware versus software and partitioning them), and path and system redundancy and system replication strategies.

5.1.1 High-Level Problem Description

As shown in Figure 20, we envision a more system-centric view of the network architecture where the focus is on integrating multiple different systems with varying criticality levels on a common network like the Time-Triggered Gigabit Ethernet (TTGbE) network and enabling an architecture that satisfies all the individual system requirements, including performance and fault tolerance.

We choose TTBgE as our preferred network, as it gives a flexible way to represent different types of networked architectures, including both synchronous and asynchronous systems. The design and verification objectives are

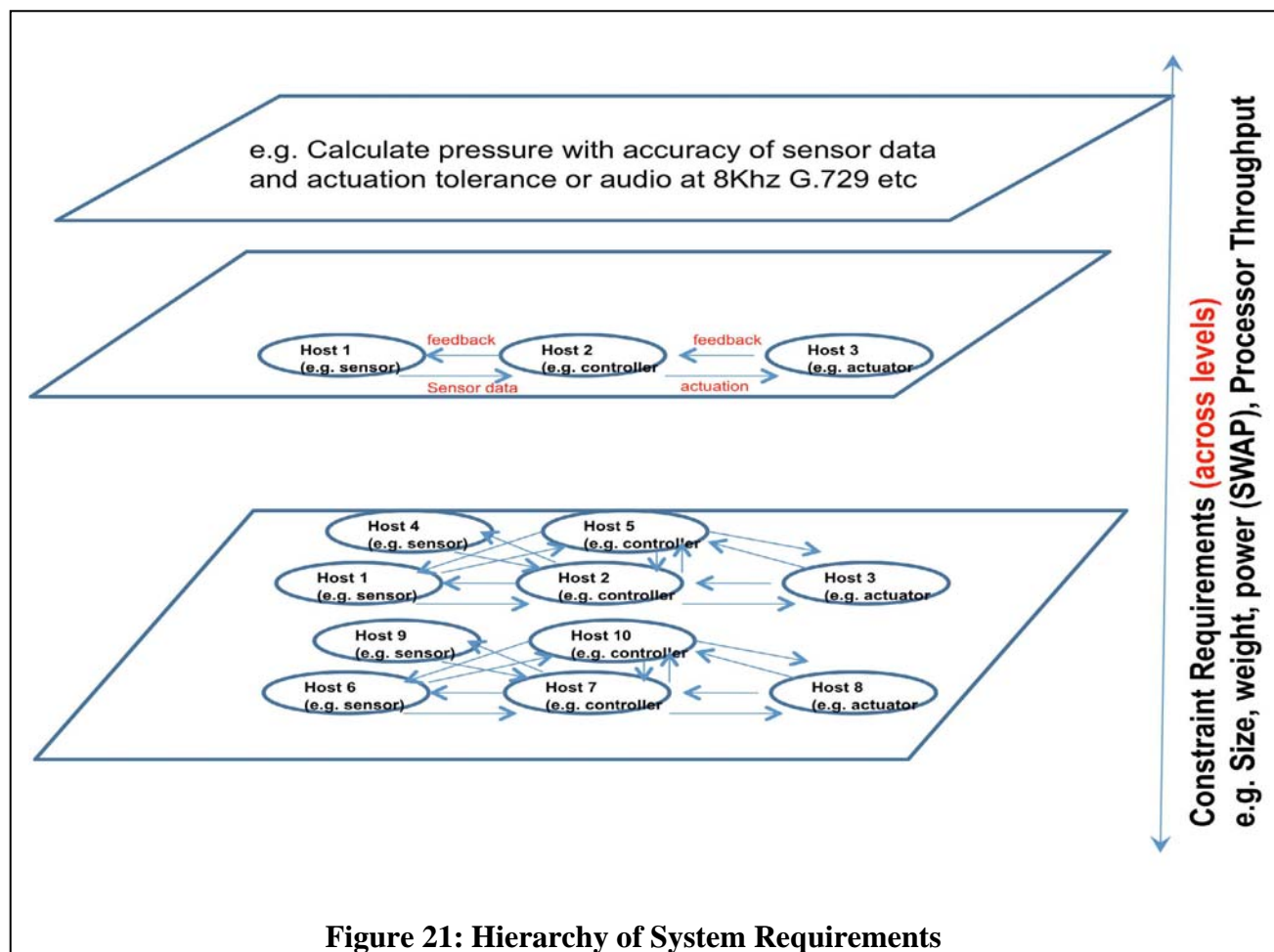
- Performance (latency, jitter, bandwidth) requirements of each application/system are met.
- Fault tolerance requirements for each subsystem are met in the presence of failure of network/host components.
- Emergent behavior that invalidates system assumptions and requirements is exposed.



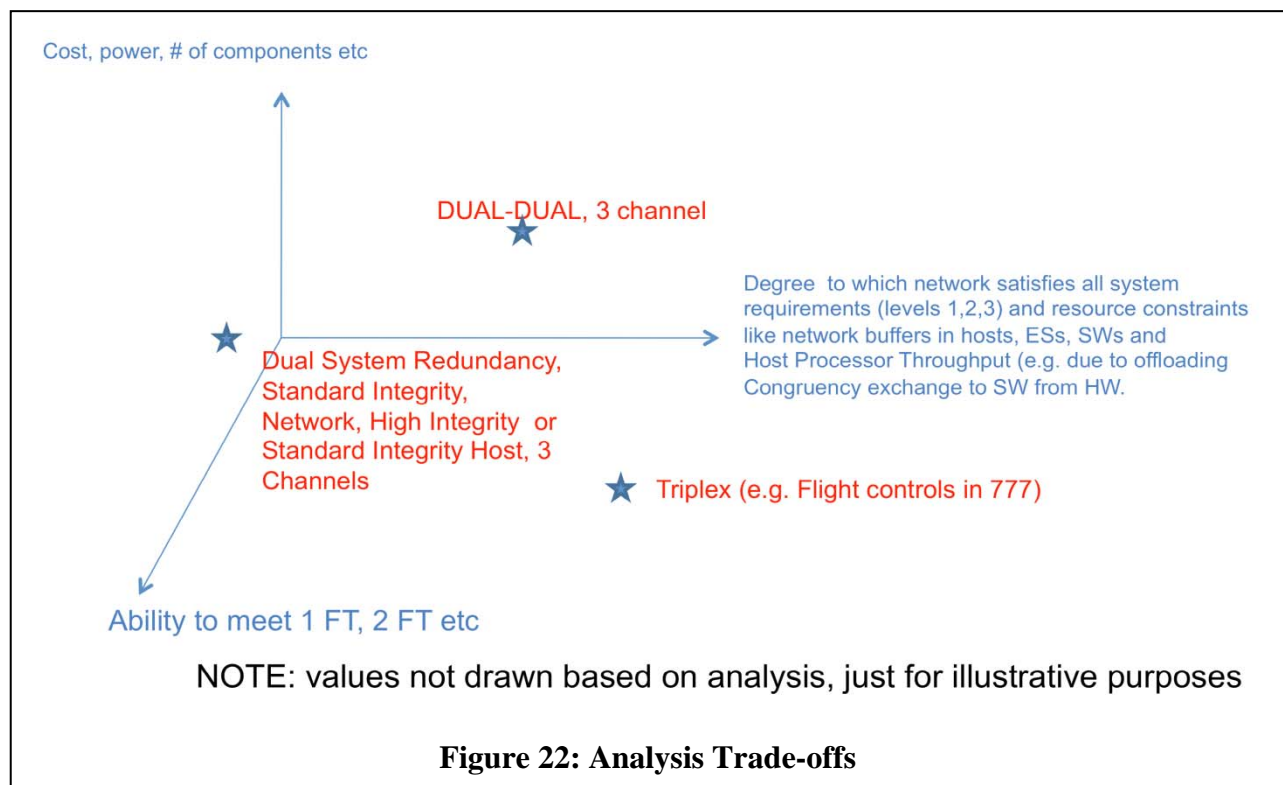
5.1.2 System Requirements and Analysis Objectives

For each of the systems that are being integrated on to the TTE network, there are requirements that need to be satisfied at different levels as indicated in Figure 21. At the top are the *level 1* requirements, which are the *functional* requirements of the system. Below it are the *level 2* requirements, which are derived workload (bandwidth) and timing (latency, jitter) requirements of the network due to *distributed architecture* of the application –, for example, different components of the closed-loop controls distributed on different nodes of the network. At the lowest level (*level 3*) are the derived workload and timing requirements, responding to *fault tolerance* needs of the system, such as probabilities of failure per flight hour being 10^{-9} for

critical systems, 10^{-7} for essential systems, 10^{-5} for noncritical systems to accommodate some failures based on resulting hazards of flight safety being categorized as catastrophic, hazardous, or major, respectively. For hazards that are categorized as minor or no effect, there may not be explicit safety objectives in terms of probabilities. At this level, the integrity of individual components (high versus standard and availability), path redundancy, and system redundancy including hardware and/or software replication in the system architecture will be taken into consideration. Note that there are additional *constraint requirements* on size, weight, power, and other resources such as buffers, link utilization, and processor throughput. These requirements are present at all three levels.



The analysis tools for failure and performance developed in this task will be instrumental in determining the ability of the underlying network to satisfy the different requirements and resource constraints of the different systems and also to aid in comparing and contrasting different network architectures for an understanding of the intrinsic trade-offs as illustrated in the Figure 22.



“Co-optimization” along both dimensions with verification proofs over the general space of all architecture is intractable. Our approach is analysis/verification for selected point(s) based upon a project’s domain expertise. We also enumerate a large number of different points in the design space in order to infer patterns of the network architecture that provide performance/fault-tolerant properties, which can then serve as a template for future network design and/or evolution of current network design. The latter part of extracting a pattern and building tools with reusable templates for network architecture design is future work.

5.2 Methods, Assumptions, and Procedures

5.2.1 Network Architecture Abbreviations and Descriptions

Table 4 introduces abbreviations used in network architectures.

Table 4. Legend and Description of Abbreviations Used in Network Design

HI	High Integrity, also referred to was COM(MANDER)/MON(ITOR) pair when 2 SI components are paired up.
SI	Standard Integrity
SW	Switch; either COTS (supports only 802.3 Ethernet traffic) or TTE (which supports traffic types TT, RC and 802.3 as Best Effort (BE))
ES	End System; TTE or COTS controller;
HOST	Host producer or consumer; processor with a partitioned operating system on which the system or application software is hosted
TT	Time Triggered Traffic (only on TTE Hardware)

RC	Rate-Constrained Traffic (only on TTE Hardware)
BE	Best-Effort Traffic (only on TTE Hardware)
Rx	Receive
Tx	Transmit
VL	Virtual link: logical connection from a single Tx ES to one or multiple Rx ESs
COTS	Commercial Off-The-Shelf (non-TTE Hardware)
FCS	Frame Check Sequence inserted into Ethernet message at Tx PHY and CRC is performed at the Rx PHY. Sometimes used synonymously with CRC.
CRC	Cyclic Redundancy Check, which is performed on the FCS of Ethernet message at Rx PHY. Sometimes used synonymously with FCS.
PHY	Physical media (link, interface, port). Representing the Ethernet links between any pair of ES \leftrightarrow SW and SW \leftrightarrow SW.
BUS	Physical media (link). Represents the PCI/CP Bus link between a pair of HOST \leftrightarrow ES

5.2.2 Model of Fault Tolerance Constructs for Network Hardware Components

Fault tolerance for a real-time safety-critical system is the ability of a system to continue normal operation despite the presence of hardware or software faults. The two key properties of the system we describe in this section from the perspective of fault tolerance are *Integrity* and *Availability*. *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 failsafe state or safe-output states in the event of failure and inability to recover. One such valid state is fail silence whereby the system either produces correct service or no service. *Availability* is a measure of the delivery of correct service with respect to the vacillation of correct and incorrect service, and it is measured by the fraction of time that the system is ready to provide the service and is in functioning condition. The instantaneous availability measure of a system is the probability that the system will be functioning correctly at any given time.

In this section we describe the different constructs in the network architecture and associated models for providing fault tolerance to different systems/applications on the network. This includes the network hardware components that support integrity (high vs. standard) and availability (path redundancies/multi-channel and/or system redundancies/replication). While the descriptions below make a clear distinction of fault-tolerant constructs that augment integrity vs. availability, we also describe scenarios in which the system redundancy construct helps augment integrity (in addition to availability). Lastly, we also briefly discuss the constructs being implemented in hardware vs. software and the inherent performance tradeoffs of the two implementations.

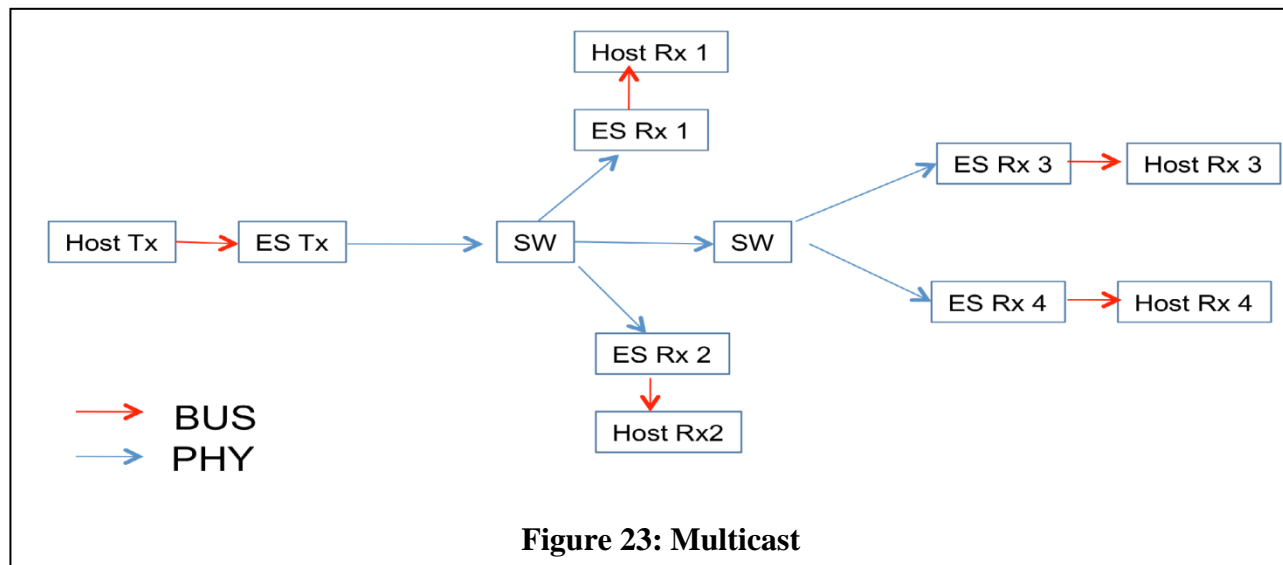
5.2.2.1 Multiple Receivers through Multicast Model for Availability

Figure 23 shows an example of a network architecture that is connected from a single producer (Host Tx) to four consumers (Host Rx). The underlying VL connecting ES Tx to ES Rx 1, 2, 3 and 4 is a multicast VL through the two different switches SW. Note that ES Rx 1 and ES Rx 2

are just one switch hop away from the source, while ES Rx 3 and ES Rx 4 are two switch hops away from the source.

Multicast can be used to increase the availability of the receivers in the system (i.e. the destination set of consumers), which is the availability set that—in the presence of loss of any of the consumers—ensures the reliability of the overall system is still satisfied because the remaining consumers can still keep the system operational. Note that the multicast approach *only* improves the availability *at the destination* and not for the source. If there is a loss of source, then the whole system fails. Also note that loss of any of the switches or any of the links connecting ES and SW can potentially result in some or all of the ES Rx 1, 2, 3 or 4 not being connected to the source ES Tx. Thus, the system may have partial or complete failure if a link or switch is lost.

Multicast may also be used without intending to improve availability. For instance, the producer can direct the traffic to a single consumer in the system and also to another consumer external to the system. An example scenario is when the external consumer is monitoring the health of the system or, more specifically, the producer. In this way, the system can be coupled. Another example is if the producer is actuating 2 different types of actuators or a single sensor is driving 2 different types of controls.



5.2.2.2 Multiple Channels through Path Redundancy Model for Availability

Figure 24 shows an example of a network architecture that is connected from a single producer (Host Tx) to two consumers (Host Rx). The underlying VL connecting ES Tx to ES Rx 1 and 2 is a multicast VL through one switch SW but there are three *redundant independent* paths between them; each pair (ES Tx, ES Rx 1) and (ES Tx, ES Rx 2) is connected via three independent switches. It is critical that the path be independent; that is, there are no common switches or shared links between the different paths. We call this triple-path redundancy or three-channel network. ES Tx would transmit an identical message redundantly over 3 ports to 3 different switches. Each ES Rx 1 and ES Rx 2 receive redundant identical messages from 3

different switches and it picks the first arriving message and subsequently discards the other redundant messages received.

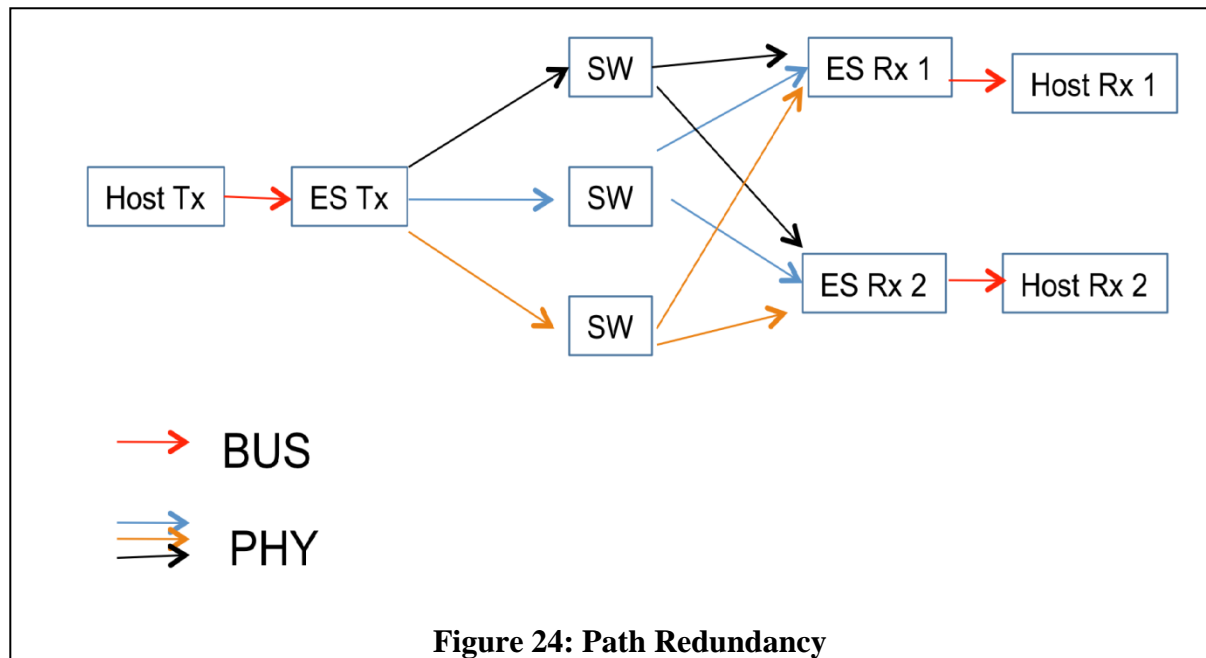


Figure 24: Path Redundancy

Path redundancy increases the availability of a message. It protects against losses that can occur in paths, meaning that there is no loss of message at the receiver despite loss of the switches and/or links between the ESs and SWs. Observe that, if in the above example there was only one Rx ES (instead of two), then path redundancy would not have protected against Rx ES availability; therefore, Rx ES fails, then the overall system fails. Thus, in the above example, multicast increases the availability of receivers while path redundancy increases switch availability and links-on-the-path availability. Like multicast, path redundancy also does not increase the availability of the source. If the source fails, the whole system fails.

5.2.2.3 Integrity Model for all Network Components

High Integrity (HI) Components are replicated *Standard Integrity (SI) Components* functioning as a single unit that uses “**cross comparisons** across replicated components” to fail-silently (i.e., when components do NOT agree), then, the combined unit (containing replicated components) is externally quiet and does not allow faults to propagate downstream. Therefore, high integrity components do not wrongly influence other components. By providing each SI component pair with identical inputs, using the same internal states and processing, and maintaining time synchronization, the pair’s outputs will match exactly unless there is a fault. The different components in a network are Host Tx, ES Tx, SW, ES Rx and Host Rx, and each of them can be instantiated in the network architecture in isolation as standard integrity or replicated with input and/or output cross compare for high integrity.

Figure 25 (A) and (B) are the SI and HI Host Tx components. The output comparison operator + controls the host access to the ES Tx via the bus. Figure 25 (C) and (D) are the SI and HI Host

Rx components and the input exchange operator + controls access from the EX Rx via the physical link.

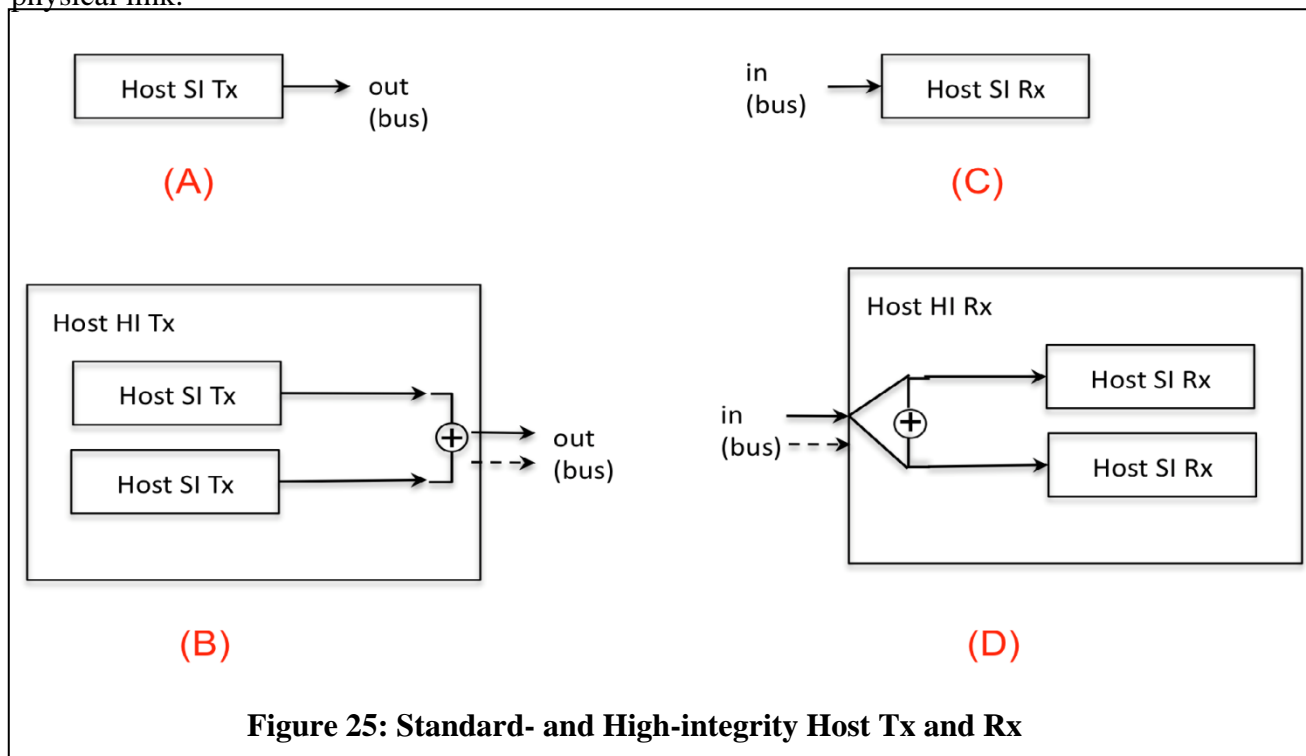


Figure 26 (A) and (B) show SI and HI ES Tx components for a single channel/path. The input exchange operator + controls access from the Host Tx via the bus. The output comparison operator + controls access to the network (SW) via the physical link. Figure 26 (C) and (D) show SI and HI ES Tx components but for multiple channels/path redundancy. Notice that in the HI components for multiple channels in Figure 26 (D), the output comparison operator is done independently per channel.

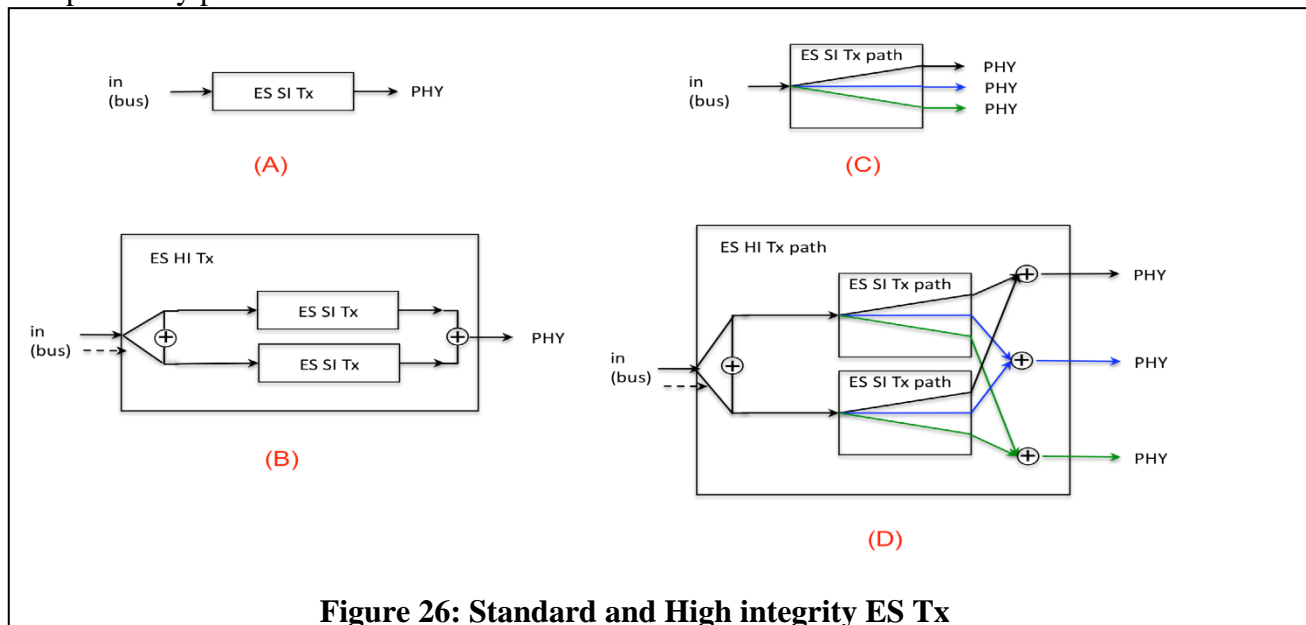
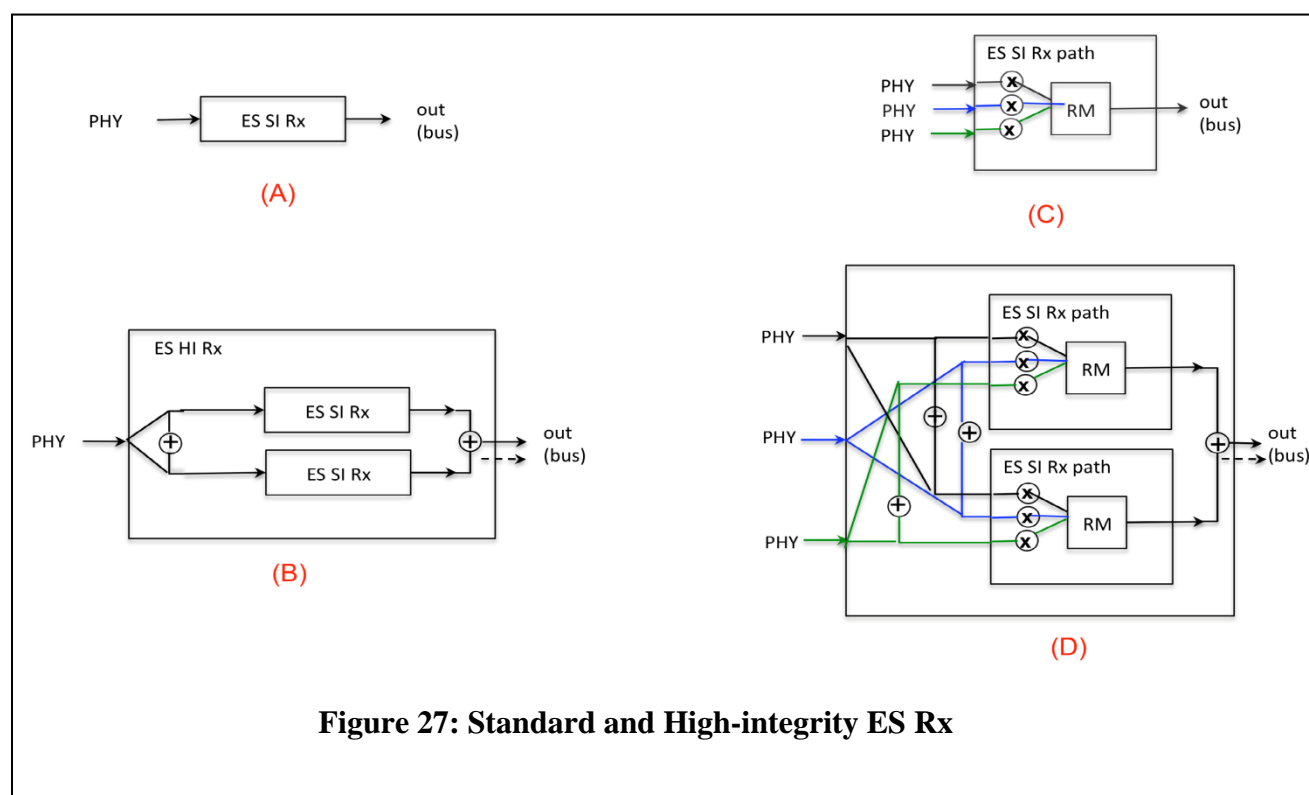
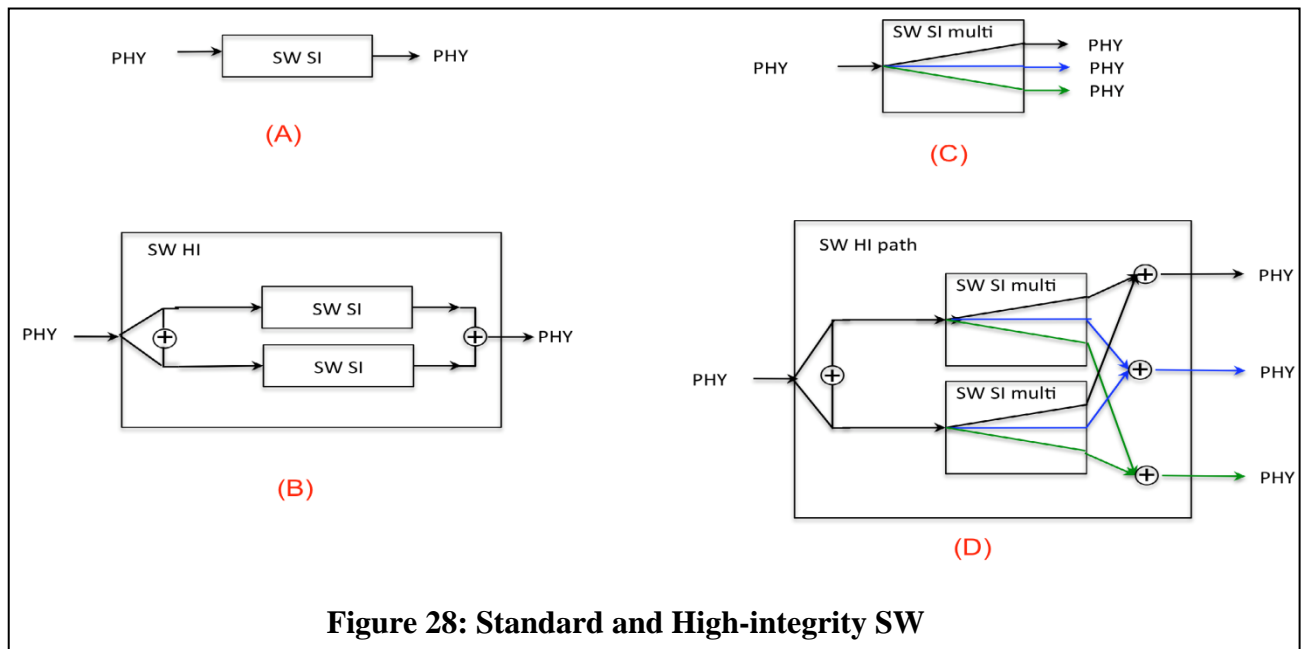


Figure 27 shows SI and HI ES Rx and conceptually but reverses the flow direction for SI and HI ES Tx indicated in Figure 26. The operators input exchange + controls access from the network (SW) via the physical link (PHY) and output comparison + controls access to the Host Rx via the bus are included. In Figure 27 (C) and (D) there is an additional operator “x” that represents Integrity checking mechanism for TT & RC; RM indicates redundancy management for TT & RC traffic arriving over multiple channel/redundant path. These mechanisms are described later in details on fault introduction and propagation and underlying mechanisms that exist to mitigate them (5.2.4 Probabilistic Fault Analysis: Failure Introduction and Propagation). Figure 28 below

shows SI and HI components for a switch. Though SW has some commonality and quite a lot of differences compared to ES Tx in terms of functional behavior, the integrity mechanisms behavior (namely, input exchange + operator and output compare + operator) is quite similar to that of ES Tx in Figure 26. For one, the operators + controls access from/to network (to other ES or SW) via the PHY is comparable. Secondly, the Figure 28 (B) and (D) indicates switch multicast (in order to support VL multicast) as opposed to ES Tx in Figure 26 (B) and (D) where they represent path redundancy/multiple channels.





5.2.2.4 Replication through System Redundancy Model for Availability and/or Integrity

We have just started working on specifying the descriptions for system redundancies based on replications strategies, characterizing their failure modes, and modeling different network architectures that support different forms of system redundancies. At this time, this work is too preliminary to report. For the time being, we enumerate the references to background material and related literature [10,11,12,13,14,15].

5.2.3 Fault Types

Below we list the current assumptions. Future work could remove these limiting assumptions.

- Wash-outs due to multiple faults are not considered at this time
 - Faults can be canceled out due to multiple faults
- Single faults are a source of failure being considered at this time
 - One faulty component is being evaluated at a time in the system. Future work will provide analysis methods for multiple faults.
- Temporal Considerations are being ignored
 - While the time at which faults occur and the duration at which the fault persists are both important, at this initial stage, our analysis ignores the temporal aspect and assumes that all faults are propagated instantaneously.

Borrowing terminology and descriptions from [16], we consider six types of faults: (i) Silent (ii) Omission (iii) Commission (babbling) (iv) Untimely (late, early) (v) Invalid (SA/DA – Ethernet Src/Dest Address, Length/Type, Msg, SN Sequence Number, Frame Check Sequence) and (vi) Inconsistent.

Silent

- Failure Mode: Receiver does not receive messages permanently. This is an error when a node fails to respond when it should have.
- Possible Causes: Faulty Transmitter or Faulty Receiver or Faulty Link/channel. Note in the case of faulty link/channel, messages may get through other channels or path if there is more than 1 path/channel in the architecture

Omission

- Failure Mode: Receiver does not receive messages temporarily (transiently or intermittently). This is an error when a node fails to respond when it should have.
- Possible Causes: Faulty Transmitter or Faulty Receiver or Faulty Link/channel; these can happen because of a faulty sender that produces Byzantine messages (that can cause asymmetric/incongruent/inconsistent receiver states) or to a faulty channel that selectively relay the messages to only a subset of nodes or a faulty receiver; in the case of a faulty link/channel, messages may get through other channels or paths if there is more than one path/channel in the architecture.

Commission

- Failure Mode: Receiver receives more messages than it should have and the error is temporary (transient or intermittent) or permanent. This is an error when a node responds when it should NOT have.
- Possible Causes: Faulty Transmitter or Faulty Receiver; this can happen because of a babbling node (faulty transmitter) or when logic in the transmitter or receiver is “latched” on to a valid state-generating spurious messages. The impact of this is an increased message rate that thereby takes up more bandwidth.

Untimely

- Failure Mode: Transmitter sends a message early or late or the receiver receives message early or late. The error can be *transient* or *permanent*. This is the timeliness property of the message when the message is not sent/received at the proper time.
- Possible Causes: Faulty Transmitter or Faulty Receiver. This can happen because of a faulty sender or receiver, which buffers the message and unduly delays it by holding on to the message and dispatching it later. Alternatively a node does not buffer a message and dispatch it at proper later time but instead sends the message immediately and thereby too early.

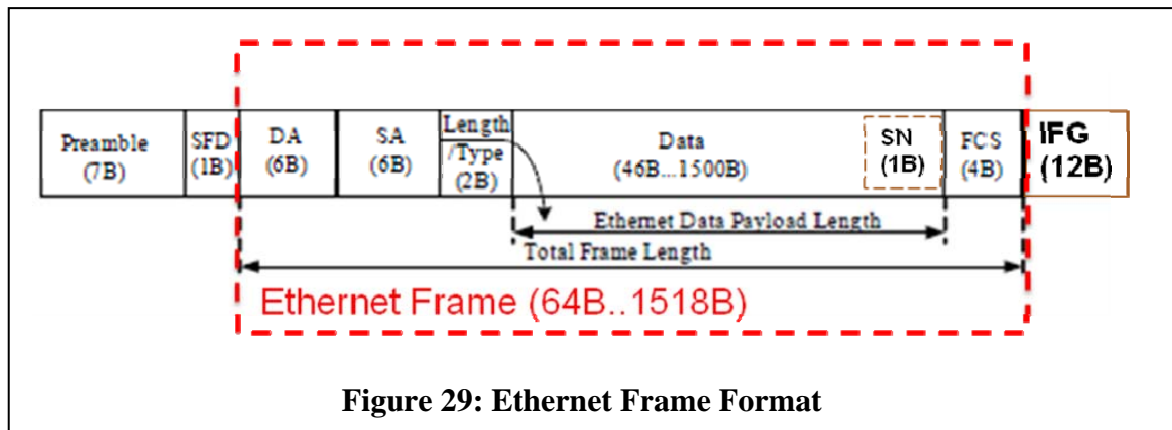


Figure 29: Ethernet Frame Format

Invalid

- Failure Mode: Invalid Ethernet messages and the error is *temporary* (transient or intermittent) or *permanent*.
- Possible Causes: Faulty Transmitter, Faulty Link or Faulty Receiver. This can happen because of an error in the message introduced at the transmitter or receiver (PHY or MAC layer), or on the link (bit errors/packet errors).
- Preamble, Static Frame Delimiter (SFD) and Inter Frame Gap (IFG) are used for MAC layer framing and NOT considered part of the Ethernet message as shown in Figure 29.
- Invalid message can be introduced in any (one or more) of the following fields in an Ethernet message:
 - **Destination Address (DA):** Virtual Link Identification (VLID) is the lower 16 bits of the 48 bits address with a fixed upper 32 bits (critical traffic marker) for all TT and RC frames. BE frames can be any value for the 48 bits as long as the value does not match the critical traffic marker of upper 32 bits. Note that for TT, RC and BE only valid DAs are allowed to be transmitted or received with appropriate configurations in Tx ES, Switch and Rx ES (valid DA at appropriate port).
 - **Source Address (SA):** Bits 5, 6, and 7 of the 48 bits address indicate the channel/redundant path the frame was transmitted on/received from for all TT & RC frames. BE frames can be any value in the 48 bits.
 - **Ethernet length/Type:** Typically this field can be used as Type field (e.g. 0x800 for IPV4, 0x806 for ARP, etc.) or as length field (value < 0x600), which indicates the actual length of the Ethernet data payload (46-1500B) following this field in the Ethernet frame. For our analysis, to simplify, we will use this solely to indicate the length.
 - **Data:** This is the Ethernet payload in which the application message is encapsulated.
 - **Sequence Number (SN):** This is the optional SN, which, when present, will be the byte preceding the Frame Check Sequence. This is required for RC Redundancy Management (RM) but not for TT RM. For our analysis we consider SN only for RC and not for TT & BE.

- **Frame Check Sequence (FCS):** This is for TT, RC, and BE the Cyclic Redundancy Check (CRC) code, based on a CRC32 polynomial, added into the FCS by the transmitter (only ES-Tx or SW-Tx). The transmitter computes the CRC over the complete Ethernet frame (DS, SA, Length/Type, Data, Optional SN) and adds this into the FCS at the end of the frame before sending to the PHY. The receiver when receiving from the PHY, computes the CRC on the Ethernet frame as the first thing and compares it against the included FCS in the message and processes it further only if it matches (or drops it otherwise). *Note that the CRC/FCS check protects (i.e., prevents it from being propagated) probabilistically against link errors introduced between the transmitter & receiver (e.g. bit flips), but DOES not protect message errors introduced in the logic before the FCS addition at the transmitter or after the FCS check in the receiver. There is also a non-zero probability that FCS may not protect against link error (bit flips in link).*

Inconsistent

- **Fault Mode:** Transient or Permanent faults that cause asymmetric/incongruent/inconsistent receiver states (i.e., divergence in receiver states). Different receivers detect different failure modes (or no failure mode at all). Orthogonal to consistent failure mode is seen when receivers all conclude on the same or identical failure mode (or no failure mode at all).
- **Possible Causes:** This can happen because of a faulty sender that produces Byzantine messages (selective transmission or dumbness) or sends to a faulty channel that selectively relays the messages to only a subset of nodes or a faulty receiver that selectively hears/receives (selective deafness) messages. Note in the case of faulty link/channel, these messages may get through other channels or path if there is more than 1 path/channel in the architecture. In case of HI/replication of identical hardware, the probability of the failure of the link is much smaller than the probability of the failure of a processor
- Four subcategories in which inconsistencies can occur or creep in for a single Virtual Link (VL) are illustrated in Figure 30 and Figure 31. The figures show where each of the inconsistencies listed below can occur either as part of the input (i/p) or output (o/p) of a component.
 1. **Path redundancy:** Inconsistency between multiple channels or paths at one receiver (can be introduced at source/transmitter)
 2. **Multiple receivers due to switch multicast:** Inconsistency between multiple receivers on one channel or path
 3. **High integrity sender/receiver:** Inconsistency between the COM & MON on the high-integrity device.
 4. **System-level redundancy:** Inconsistency between redundant systems, i.e., between 2 sets of one-to-many producer-to-consumer(s) systems

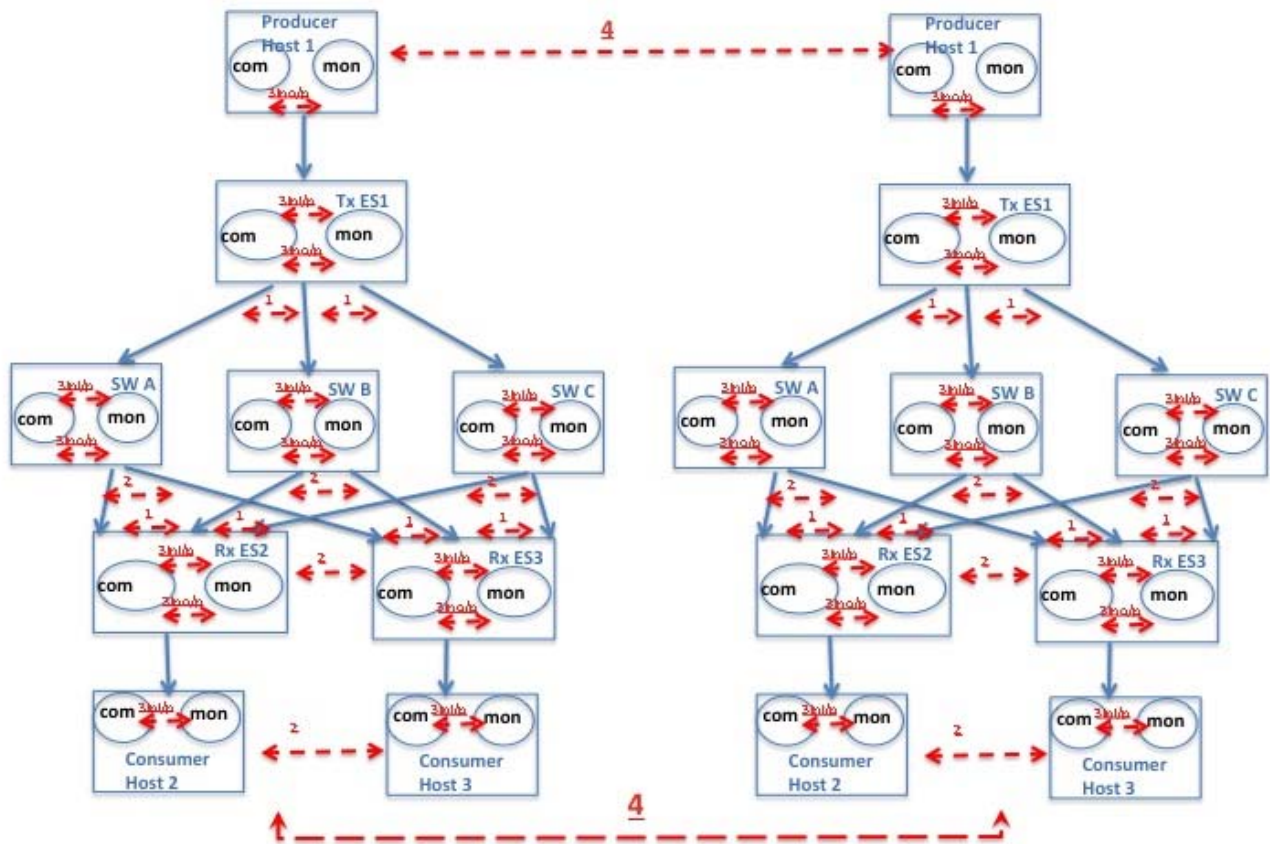
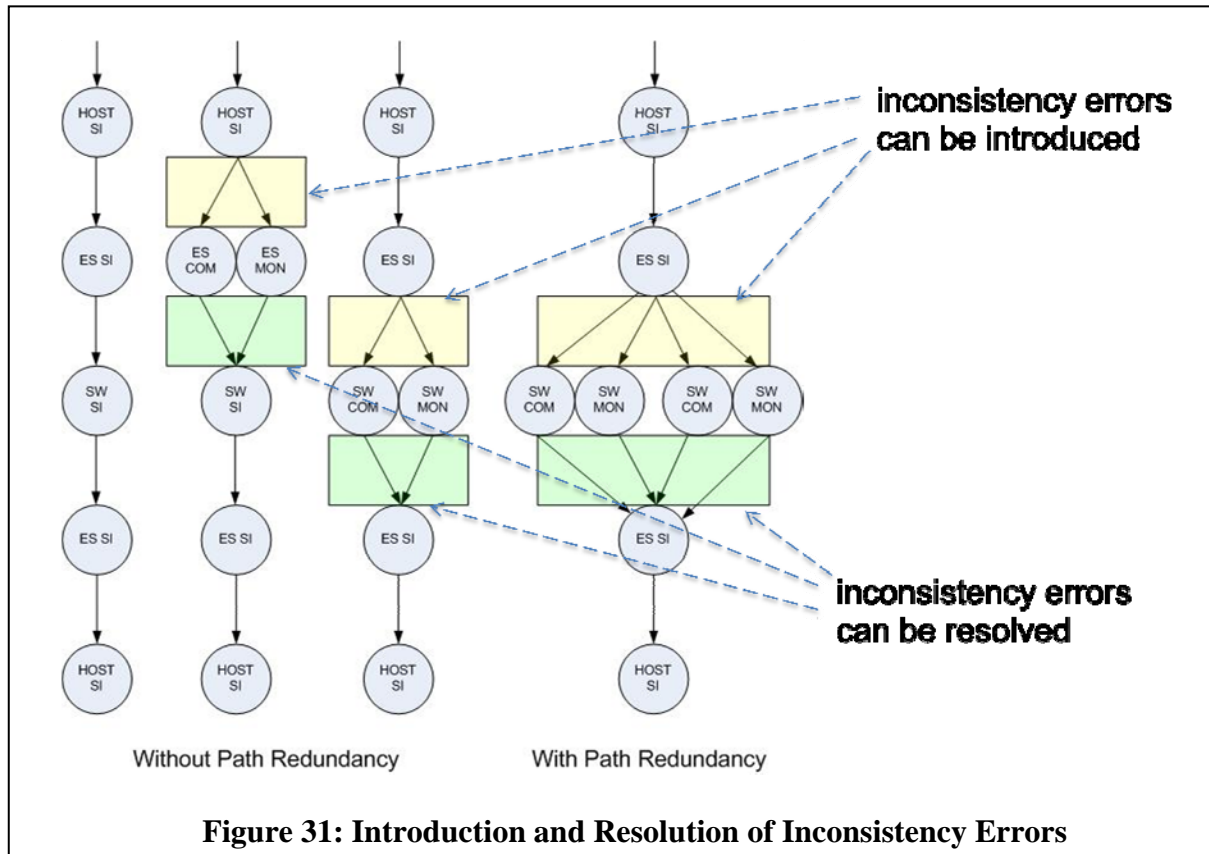


Figure 30: Introduction and Resolution of Inconsistency Errors



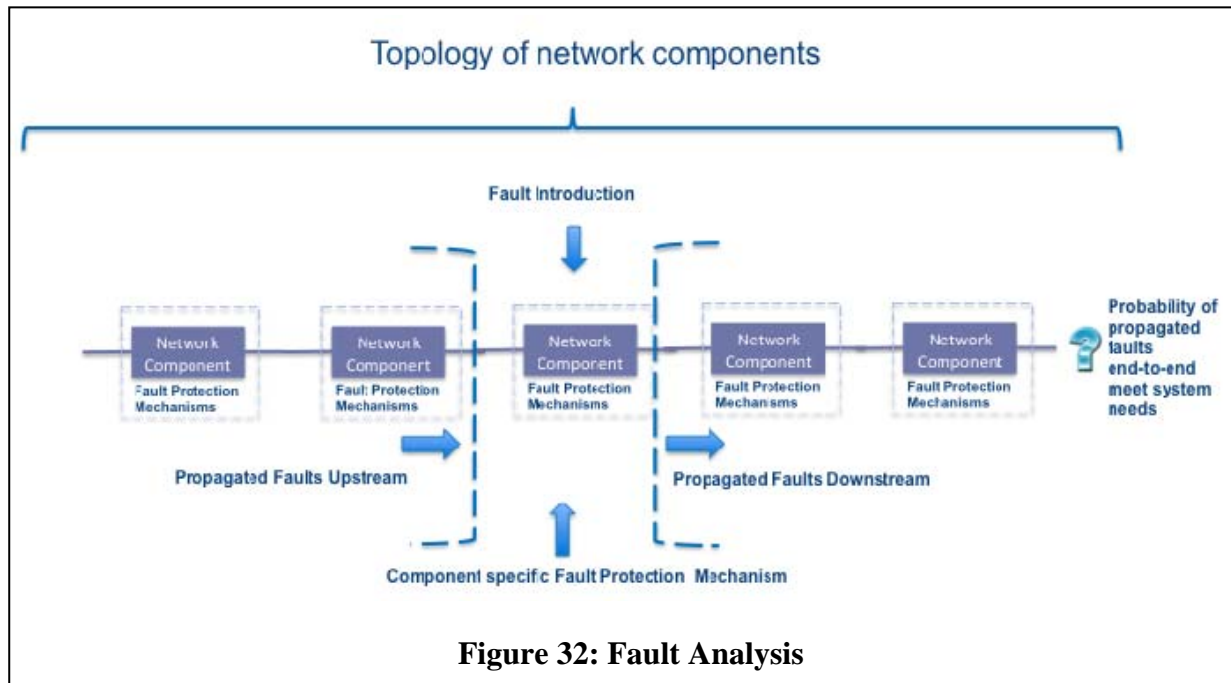
5.2.4 Probabilistic Fault Analysis: Failure Introduction and Propagation

As shown in Figure 32, we analyze faults in a probabilistic fashion, end-to-end on arbitrary topologies and network architectures, for all faults that are defined in previous section. The fault analysis would take into account *faults introduced* at each component including (i) types of faults introduced and (ii) their probability of occurrence, as well as how faults affecting a component are propagated. We calculate for each component how incoming faults (i.e., faults propagated up to this point from upstream network components) and internal faults manifest as outgoing faults and what probability the outgoing faults have (as a function of the incoming and internal faults). This is indicated as *propagated faults upstream* in Figure 32.

The *component specific failure protection mechanism*, referred in Figure 32 applies to both failures propagated upstream and up to and also introduced in this component. We then model the efficacy of protection mechanism by computing the probability of allowing failure to propagate downstream. Next, we compute the probability of occurrence of *propagated faults downstream* for all types of faults that may be propagated from this point in the component to any downstream network components. Note that the protection mechanism of the component may transform certain faults that have propagated up to the component into other faults downstream. For example, propagated *commission* faults at ES SI Tx, due to BAG policing at SW in a TTE network, manifest as omission fault downstream from the SW.

Finally, we analyze whether or not the system's fault tolerance requirements are satisfied by making sure the probabilities of propagated failures end-to-end for all types of failures (and the

number of failures) are less than the system's failure rate requirements (e.g., the probabilities of failure per flight hour being 10^{-9} for critical systems).



The set of different network components for which we need to individually model and analyze failure probabilities are:

- Host SI Tx
- ES SI Tx
- ES SI Tx with multiple channel/redundant path
- SW SI
- SW SI with multicast
- ES SI Rx
- ES SI Rx with multiple channel/redundant path (i.e., with RM & IC functions in the component)
- Host SI Rx
- Host HI Tx
- ES HI Tx
- ES HI Tx with multiple channel/redundant path
- SW HI
- SW HI with multicast

- ES HI Rx
- ES HI Rx with multiple channel/redundant path (i.e., with RM & IC functions in the component)
- Host HI Rx

The component-specific failure protection mechanisms are summarized in

Table 5.

Table 5. Failure Protection Mechanism of Components

Components	Traffic Type supported	Failure Protection mechanism description
ES SI Tx, ES SI Tx with multiple channel/redundant path, ES HI Tx, ES HI Tx with multiple channel/redundant path, SW SI, SW SI with multicast, SW HI, SW HI with multicast, ES SI Rx, ES SI Rx with multiple channel/redundant path, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC, BE, COTS	FCS/CRC addition at Tx (output) and check and Rx (input)
ES SI Tx, ES SI Tx with multiple channel/redundant path, ES HI Tx, ES HI Tx with multiple channel/redundant path, SW SI, SW SI with multicast, SW HI, SW HI with multicast, ES SI Rx, ES SI Rx with multiple channel/redundant path, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC, BE	VLID check (part of Eth DA) arrived on valid Rx (input) port for critical traffic table checks for TT & RC. Eth DA valid is checked for BE also at Rx (input) port via the routing and anti-masquerading checks.
SW SI, SW SI with multicast, SW HI, SW HI with multicast, ES SI Rx, ES SI Rx with multiple channel/redundant path, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC, BE, COTS	Valid Message length check at Rx (input) port (i.e., if actual payload [data] length matches Ethernet type/length field).
ES SI Tx, ES SI Tx with multiple channel/redundant path, ES HI Tx, ES HI Tx with multiple channel/redundant path, SW SI, SW SI with multicast, SW HI, SW HI with multicast, ES SI Rx, ES SI Rx with multiple channel/redundant path, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC	Actual Message (payload) length is less than configured maximum length possible check at Rx (input) port.
ES SI Tx, ES SI Tx with multiple channel/redundant path, ES HI Tx, ES HI Tx with multiple channel/redundant path, SW SI, SW SI with multicast, SW HI, SW HI with multicast	TT, RC, BE	Dispatch enforcement at Tx (output) port. Scheduled Dispatch (TT) strictly on a schedule and in gaps in the unscheduled timeline (RC) based on priority (FIFO within priority) and finally BE a slowest priority.
ES SI Tx, ES SI Tx with multiple channel/redundant path, ES HI Tx, ES HI Tx with multiple channel/redundant path	RC	At Tx (output) port. Traffic Shaping (ARINC 664)
SW SI, SW SI with multicast, SW HI, SW HI with multicast	TT	Strict Timing Window Enforcement (TT) at Rx (input) port.

Components	Traffic Type supported	Failure Protection mechanism description
SW SI, SW SI with multicast, SW HI, SW HI with multicast	RC	Bandwidth Allocation Gap (BAG) at Rx (input) port. This is a rate-limiting check.
SW SI, SW SI with multicast, SW HI, SW HI with multicast	BE	“Coarse” per-port BAG Enforcement at Rx (input) port. This is a rate-limiting check for all BE traffic that arrives on the configured port.
SW SI, SW SI with multicast, SW HI, SW HI with multicast	RC	At Tx (output) port, Age Check for RC Message. Check resident delay in SW (from Rx to Tx) against maximum configured delay and drops, if more delay occurred. Note that this check buys protection for timing delay and for “small” probability protection against timing for a global timebase.
ES SI Rx, ES SI Rx with multiple channel/redundant path , ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC	At Rx (input) port, Redundancy Management (RM) across channels; First SN accept and drop redundant SNs for RC; Accept first frame for TT within a period and excluding duplicate frames for configured time window within a period after accepting the first frame.
ES SI Rx, ES SI Rx with multiple channel/redundant path, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT (optional), RC	At Rx (input) port, Integrity Check (IC) per channel/redundant path; Additional SN check (Prev SN +1 or prev SN + 2). This was used as operator “x” in Figure 27.
ES HI Tx, ES HI Tx with multiple channel/redundant path, SW HI, SW HI with multicast, ES HI Rx, ES HI Rx with multiple channel/redundant path, Host HI Rx,	TT, RC, BE	High Integrity input-exchange operator “+” at Rx (input) port
Host HI Tx, ES HI Tx, ES HI Tx with multiple channel/redundant path, SW HI, SW HI with multicast, ES HI Rx, ES HI Rx with multiple channel/redundant path	TT, RC, BE	High Integrity output cross compare operator “+” at Rx (output) port

5.2.5 Analysis Tool Chain Overview

Figure 33 gives an overview of the performance and fault analysis tools that we implemented. The fault-free performance tool checks whether TT traffic is schedulable and what is the end-to-end latency for RC traffic. The fault analysis tool checks for each VL the probability of the various failure types. The tools are introduced in more detail below.

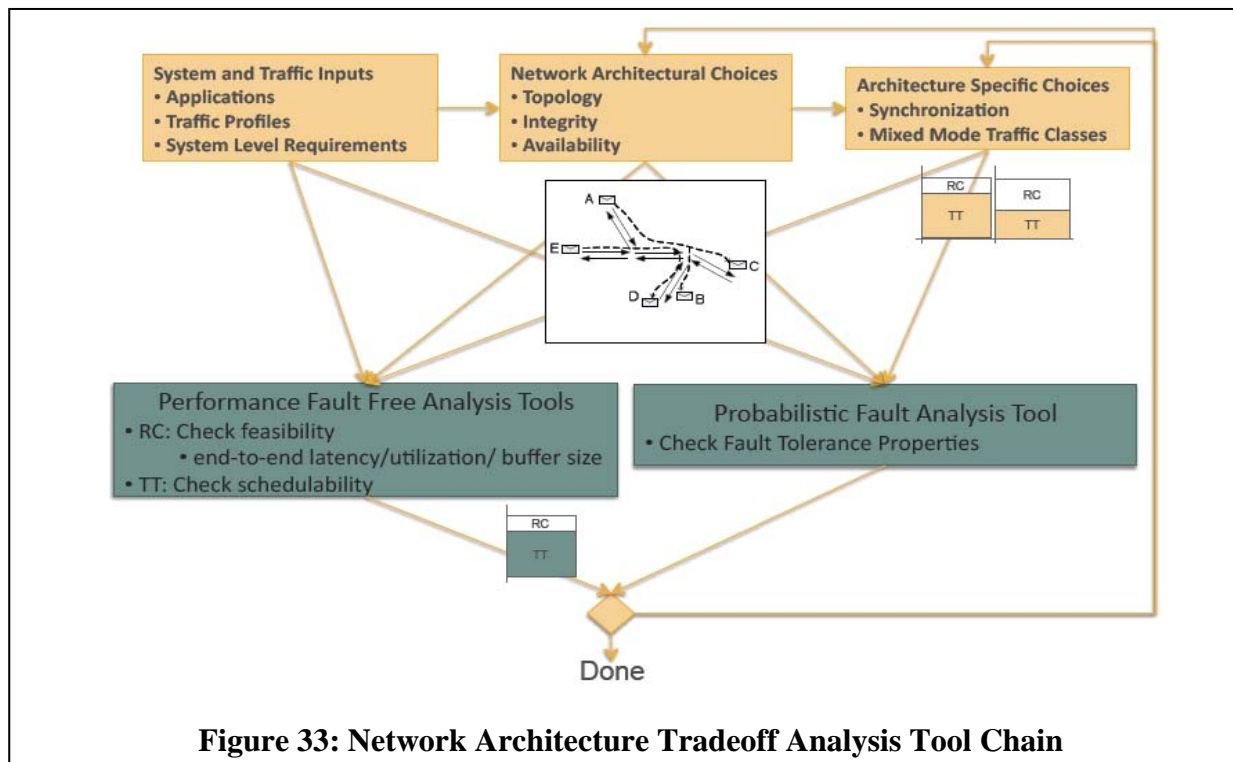


Figure 33: Network Architecture Tradeoff Analysis Tool Chain

Figure 34 summarizes the input needed for the analysis tools.

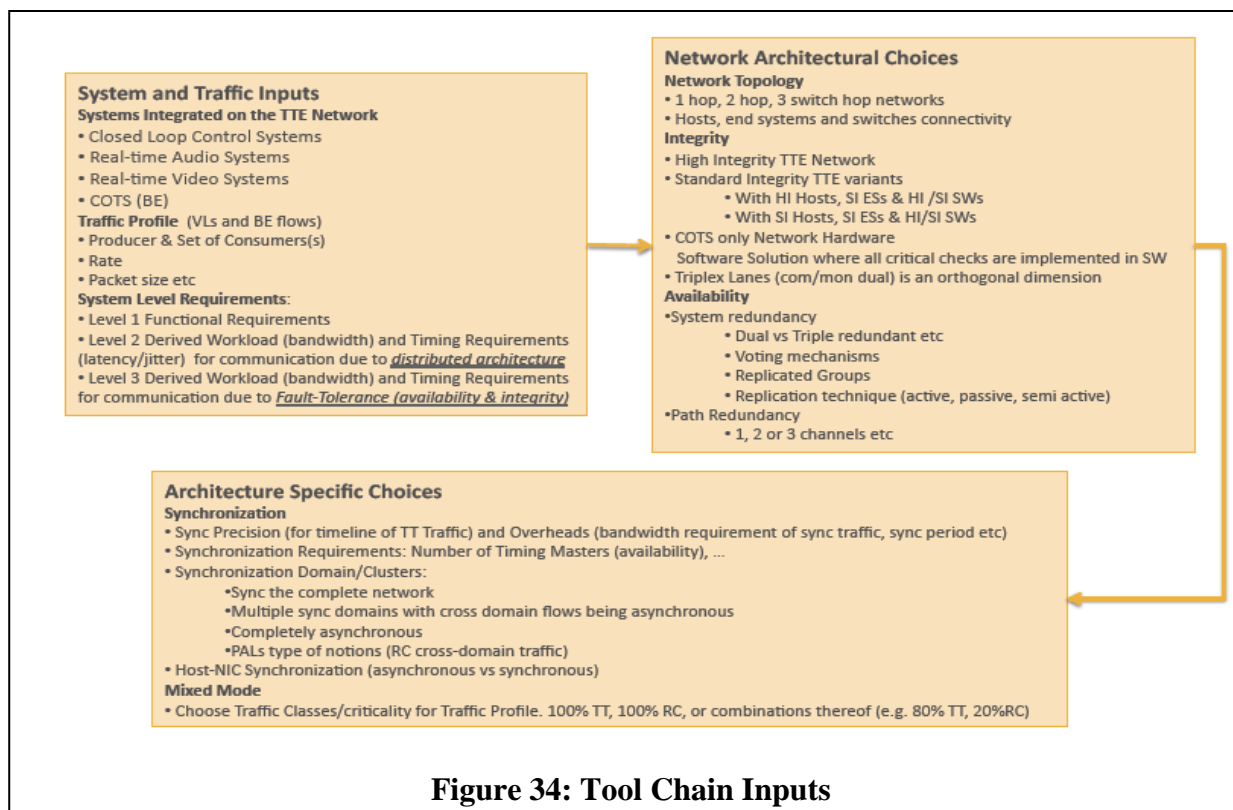


Figure 34: Tool Chain Inputs

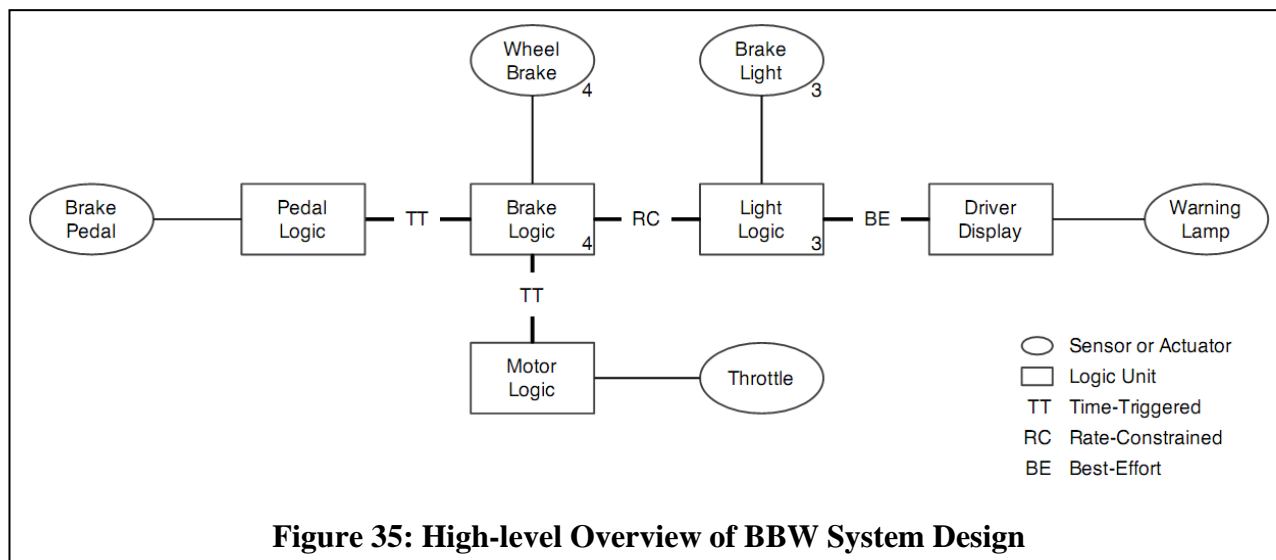
5.2.6 Brake-by-wire Case Study

Fault tolerance is one of the most critical features in the brake system of cars. Papadopoulos et al. [17] describes details of an initial brake-by-wire (BBW) model. However, we propose extending the scope to include communication of signals from the wheel brakes to the brake lights, as well as feedback to the driver from the light sensors about whether any of the bulbs need to be replaced. Finally, we also include safety-critical communication of brake signals to the motor control logic in order to prevent opening the throttle from opening while braking.

Our model assumes that the communication infrastructure supports TT Ethernet (TTE) with its different types of traffic, namely time-triggered (TT), rate-constrained (RC), and best-effort (BE).

Figure 35 is a high-level overview of our BBW system design. Each sensor and actuator has a corresponding logic unit, which interfaces with the communication infrastructure of the car. The expected behavior of this system is governed by the following rules:

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

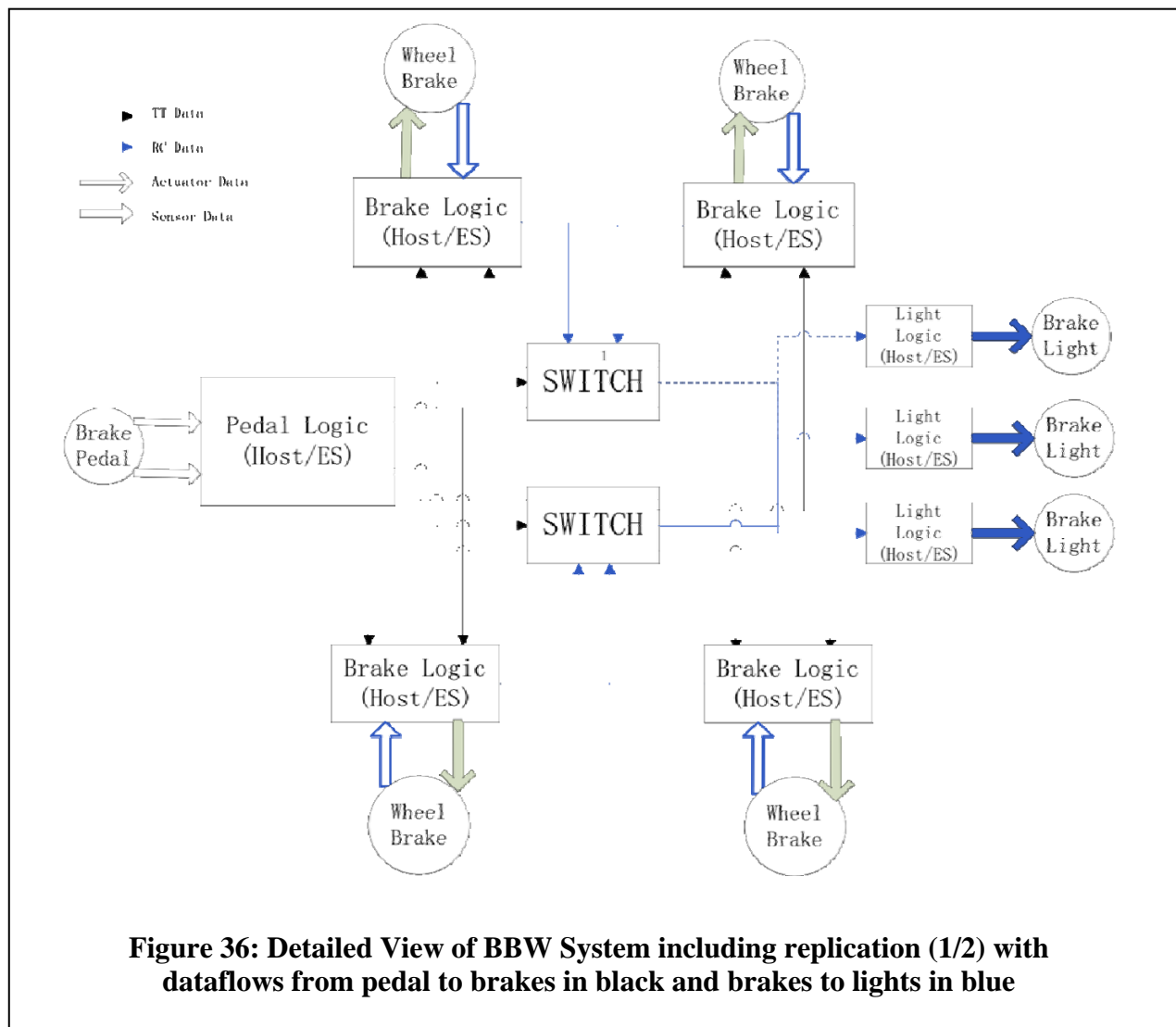


Corresponding to the criticality of the signal, the TTE communication links are labeled as time-triggered (high criticality), rate-constrained (medium criticality), and best-effort (low criticality). The highly critical paths must be protected by redundancy. We consider dual and triple modular redundancy in this study.

Furthermore, we assume that the brake signal is always communicated from the pedal to the wheels and from the wheels to the motor logic during operation. If the brake pedal is not engaged, the brake logic sends a value of 0 to the wheels. Similarly, if the wheel brakes are not

engaged, they send a value of 0 to the motor logic. Then, if messages are missing for a prolonged period of time, the wheel brakes should engage to come to a fail-stop. However, it may be advisable to slowly engage or perform consensus first, as asymmetrical braking may lead to instability of the car. Such advanced behavior of handling faults is not covered in this study.

Figure 36 and Figure 37 present a more detailed view of the architecture. Here, we already made a design decision to use dual redundancy for the highly critical communication paths from the brake pedal to the wheels and from the wheels to the motor logic. The communications of medium and low criticality are assigned to one of the two switches in this instantiation of the system.



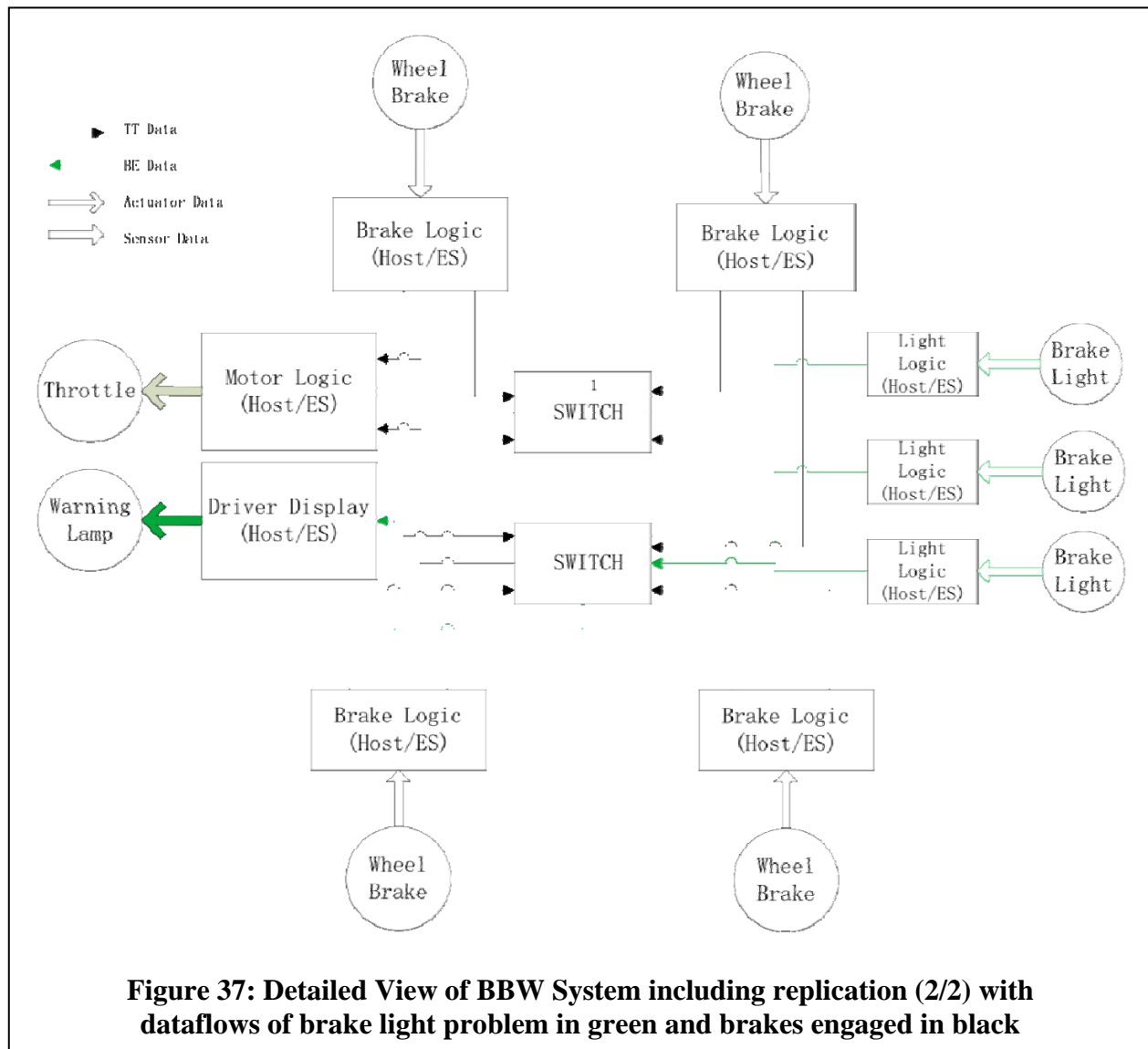


Table 6 lists all dataflows considered in our model. We use high-integrity devices and TT traffic to send the brake signal from the pedal to the wheel brakes. For the light signal from the wheel brakes to the brake lights, we use high-integrity devices and RC (rate-constrained) traffic. Finally, the signal sent from the light logic to the display logic uses BE (best-effort) traffic.

Table 6. Dataflows in BBW system

VL ID	Sender	Receiver	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

For the purpose of fault propagation analysis, each dataflow originates and ends at a host, and traverses the following components: host to end system (transmit) via physical link to switch via physical link to end system (receive) to host. We believe that any kind of Ethernet network can be evolved from such a basic model.

5.2.7 Equational Logic, Rewriting Logic, and Maude

Our network analysis tools build on the Maude language and tool, which is based on equational logic and rewrite logic. Maude is well suited to this role due to its *flexible syntax* and *powerful inference* capabilities. The flexible syntax lets us specify networks in an extensible way that can accommodate future extensions and interoperate with other tools using the Maude syntax as the interlingua between them. The powerful inference capability allows us to do fault analysis in Maude itself, as we will discuss below. In the following sections below, we introduce Maude and its logical basis.

Equational logic (EL) [18] is the subset of first-order logic with the = sign as the only predicate symbol, and equations as the only formulas (i.e., there are no logical connectives). The rules of deduction of EL are *reflexivity*, *congruence*, *transitivity*, and *symmetry*. 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) and using *reduction* as the operational semantics. Viewed as rewrite systems, theories in EL are expected to be terminating and confluent. This means that the order in which redexes and rewrite rules are chosen does not matter; we will reach the same result regardless and in a finite number of steps. Sometimes *conditional* equations are allowed, which means that Horn clauses can be used in addition to plain equations.

For example, given an encoding of the natural numbers as \emptyset , $s(\emptyset)$, $s(s(\emptyset))$, ..., one can define addition (using an infix + function) as:

$$\emptyset + x = x \tag{17}$$

$$s(x) + y = s(x + y) \tag{18}$$

Using the equations above, we can compute $2 + 3$ through the following steps (with the redexes underlined):

$$s(s(\emptyset)) + s(s(s(\emptyset))) \Rightarrow s(s(\emptyset) + s(s(s(\emptyset)))) \Rightarrow s(s(\emptyset + s(s(s(\emptyset)))) \Rightarrow s(s(s(s(\emptyset)))). \tag{19}$$

Rewriting logic (RL) [19] is similar to EL on the surface, in that it allows for (possibly conditional) rewrite rules. However, the rules are not semantically equations, and the rule systems are not expected to be confluent. Therefore, reduction cannot be used as the operational semantics, since different choices of redexes and rules can lead to different results. Instead, the rewrite rules are interpreted as nondeterministic state transitions, and the operational mechanism is *search* in the state space. Analogously to EL, RL can also support conditional rewrite rules.

Maude [20] is a multiparadigm executable specification language encompassing both EL and RL. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. Maude also provides efficient built-in search and model-checking capabilities. Maude is reflective [21], providing a meta-level module that reflects both the syntax and semantics of Maude. Using reflection, the user can program special-purpose execution and search strategies, module transformations, analyses, and user interfaces. Maude sources, executables for several platforms, the manual, a primer, cases studies, and papers are available from the Maude website <http://maude.cs.uiuc.edu> or its mirror <http://maude.csl.sri.com>.

We briefly summarize the syntax of Maude used in this report. Maude has a modular system, with:

- *Functional* modules, specifying equational theories, which are declared with the syntax `fmod ... endfm`
- *System* modules, which are rewrite theories specifying systems of state transitions; they are declared with the syntax `mod ... endm`

These modules have an *initial model semantics* [22]. Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword followed by the operator name, the argument, and result sorts. An operator may have mixfix syntax, with the name containing `_`'s marking the argument positions. A binary operator may be declared with equational *attributes*, such as `assoc`, `comm`, and `id`: `<identity element>` stating, for example, that the operator is associative, commutative, and specifying an identity element for the operation. Such attributes are then used by the Maude engine to match terms modulo the declared axioms. 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 `=`. Rewrite rules are introduced with the keyword `r1` (or `cr1` for conditional rules) followed by an optional rule label, and terms corresponding to the premises and conclusion of the rule separated by the rewrite sign `=>`. Variables appearing in axioms and rules (and commands) may be declared globally using the keyword `var` or `vars`, or “inline” using the variable name and its sort separated by a colon; for example, `n:Nat` is a variable named `n` of sort `Nat`. Rewrite rules are not allowed in functional modules.

Maude has a `reduce` command for equational reduction in functional modules, and a `search` command for breadth-first search in the state space of system modules. The search mechanism allows searching for the first answer, all answers, or only answers matching some goal term. The search mechanism encompasses the reduction mechanism, as equational reduction is performed before each application of rewrite rules.

5.3 Results and Discussion

5.3.1 Network Specification in Maude

We use the Maude language to specify networks. We need to be able to specify at least:

- *Components* in the network, such as routers, physical links, and end systems
- *Topology* of the network components, i.e., how they are connected to each other
- Required *dataflows* in the network, i.e., the desired traffic flows

In the following, we introduce (parts of) our Maude modules used to represent these elements, which can be found in the `network.maude` file.

We introduce a functional module `NETWORK` and declare the sorts used to specify our networks.

```
fmod NETWORK is
```

```
...
```

```
  sorts NetworkConfiguration Component Connection HISwitch SISwitch HIEndSystem  
  SIEndSystem HIHost SIHost Link Dataflow PhysicalLink .
```

```
  subsorts Component Dataflow < NetworkConfiguration .
```

HI and SI are short for *high integrity* and *standard integrity*, respectively, where high integrity refers to a self-checking pair of processor (e.g., as used extensively in TTEthernet networks and other modern network technologies where reliability is important). We define `Component` and `Dataflow` as subsorts of the `NetworkConfiguration` sort, to allow them to be used wherever a `NetworkConfiguration` is required.

Subsequently, we define constructors for all the network component types.

```
  subsort Connection < Component .
```

```
  op < conn(_)|_to_> : Nat OutPort InPort -> Connection [ctor] .
```

```
  subsort PhysicalLink < Component .
```

```
  op < pl(_)|_|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
```

```
  PhysicalLink [ctor] .
```

```
  subsort SISwitch < Component .
```

```
  subsort HISwitch < Component .
```

```
  op < si-sw(_)|_|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
```

```

SISwitch [ctor] .

  op < hi-sw(_)|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->

HISwitch [ctor] .


subsort SIEndSystem < Component .

subsort HIEndSystem < Component .

  op < si-es(_)|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
SIEndSystem [ctor] .

  op < hi-es(_)|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
HIEndSystem [ctor] .


subsort SIHost < Component .

subsort HIHost < Component .

  op < si-host(_)|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
SIHost [ctor] .

  op < hi-host(_)|_|_> : Nat InPortSet OutPortSet FailureAnnotationSet ->
HIHost [ctor] .

```

We use angle brackets <...> everywhere to give our components a uniform appearance. All components take a Nat (natural number) argument to give them a unique identifier. Most components have one or more incoming or outgoing *ports*. Ports are introduced in their own Port module.

```

fmod PORT is
  pr NAT .
  *** Ports
  sorts Port InPort OutPort .
  op in(_) : Nat -> InPort [ctor] .
  op out(_) : Nat -> OutPort [ctor] .
endfm

```

(The NETWORK module uses *sets* extensively – for example, sets of ports, as in InPortSet and OutPortSet. The Maude mechanism for defining such sets, or other “polymorphic data types” is somewhat intricate, and we do not show it here). Again, natural numbers are used as identifiers.

One important component type is the *connection*. Note the syntax of its constructor above:

```
op < conn(_)|_to_> : Nat OutPort InPort -> Connection [ctor] .
```

The “_to_” part specifies two ports that are connected through this connection, always an out-port to an in-port. Note that connections do not represent actual physical links. Those have their own type and constructor (see “pl” in the specification above) because we want to model the possibility of failures in the physical links. The connections here are used in addition to describe, for example, which physical link is connected to which port on a switch.

Many of the components also have an argument for *failure annotations*. These are used to show what types of failures can occur in the component. Failure annotations are defined in their own modules Failure and FailureAnnotation.

```
fmod FAILURE is
  *** component-internal temporary failure
  *** failures cause faults
  sort FailureType .
  op swfail : -> FailureType [ctor] .
  op hostfail : -> FailureType [ctor] .
  op esfail : -> FailureType [ctor] .
  op fcschfail : -> FailureType [ctor] .
  op plfail : -> FailureType [ctor] .
endfm

fmod FAILUREANNOTATION is
  pr FAILURE .
  pr FORMULA .
  sort FailureAnnotation .
  op _:_ : FailureType Formula -> FailureAnnotation [ctor] .
  op unknown : -> FailureAnnotation [ctor] .
endfm
```

A failure annotation combines a failure type, such as “swfail” (switch failure) with a *formula* that describes the logical conditions under which the failure occurs. We will return to these formulas later.

Going back to the main NETWORK module, we have one additional component type: *dataflow*.

```
sort TrafficType .
op tt : -> TrafficType [ctor] .
op rc : -> TrafficType [ctor] .
op be : -> TrafficType [ctor] .

op df(_|_|_) : Nat TrafficType DataflowComponentSet -> Dataflow [ctor] .
```

Dataflows are used to represent a network path through which a particular type of data travels. The traffic types are tt, rc, and be, for *time-triggered*, *rate-controlled*, and *best-effort*. Each dataflow also has a set of components.

```
fmod DATAFLOWCOMPONENT is
  pr NAT .
```

```

pr SET{FaultAnnotation} * (sort Set{FaultAnnotation} to FaultAnnotationSet) .
sort DataflowComponent .
op conn(_|_) : Nat FaultAnnotationSet -> DataflowComponent [ctor] .
endfm

```

Dataflow components are essentially connections annotated with fault annotations. Fault annotations are also defined in their own module.

```

fmod FAULT is
  sort FaultType .
  op sil : -> FaultType [ctor] .
  op om : -> FaultType [ctor] .
  op com : -> FaultType [ctor] .
  op vSA : -> FaultType [ctor] .
  op vDA : -> FaultType [ctor] .
  op vSN : -> FaultType [ctor] .
  op vLen : -> FaultType [ctor] .
  op vData : -> FaultType [ctor] .
  op vFCS : -> FaultType [ctor] .
  op te : -> FaultType [ctor] .
  op tl : -> FaultType [ctor] .
  op incPath : -> FaultType [ctor] .
  op incHI : -> FaultType [ctor] .
endfm

fmod FAULTANNOTATION is
  pr FAULT .
  pr PROB .
  sort FaultAnnotation .
  op _:_ : FaultType Prob -> FaultAnnotation [ctor] .
  op unknown : -> FaultAnnotation [ctor] .
endfm

```

Faults and fault annotations are similar to failures and failure annotations. The difference is that failures are internal to components, and are typically the *cause* of faults, which are symptoms of such failures. In the end, it is primarily the faults that we are interested in analyzing. The fault types are discussed in more detail in 5.2.3 Fault Types. Note the special unknown fault annotation.

Typically, a network configuration specified by the user will use this for all the fault annotations. The Maude fault analysis tool will then infer the actual non-unknown annotations using fault propagation/introduction rules. This process is discussed in detail in Section 5.3.3 Fault Analysis in Maude.

Finally, we have the top-level network configurations.

```

*** Network configurations
op none : -> NetworkConfiguration [ctor] .
op __ : NetworkConfiguration NetworkConfiguration -> NetworkConfiguration [ctor
assoc comm id: none] .

```


Essentially, a network configuration is just a set of network components, combined using the “empty syntax”, `__`.

With the syntax described so far, we can already specify arbitrarily complex network configurations. For example, the term (of sort `NetworkConfiguration`)

```

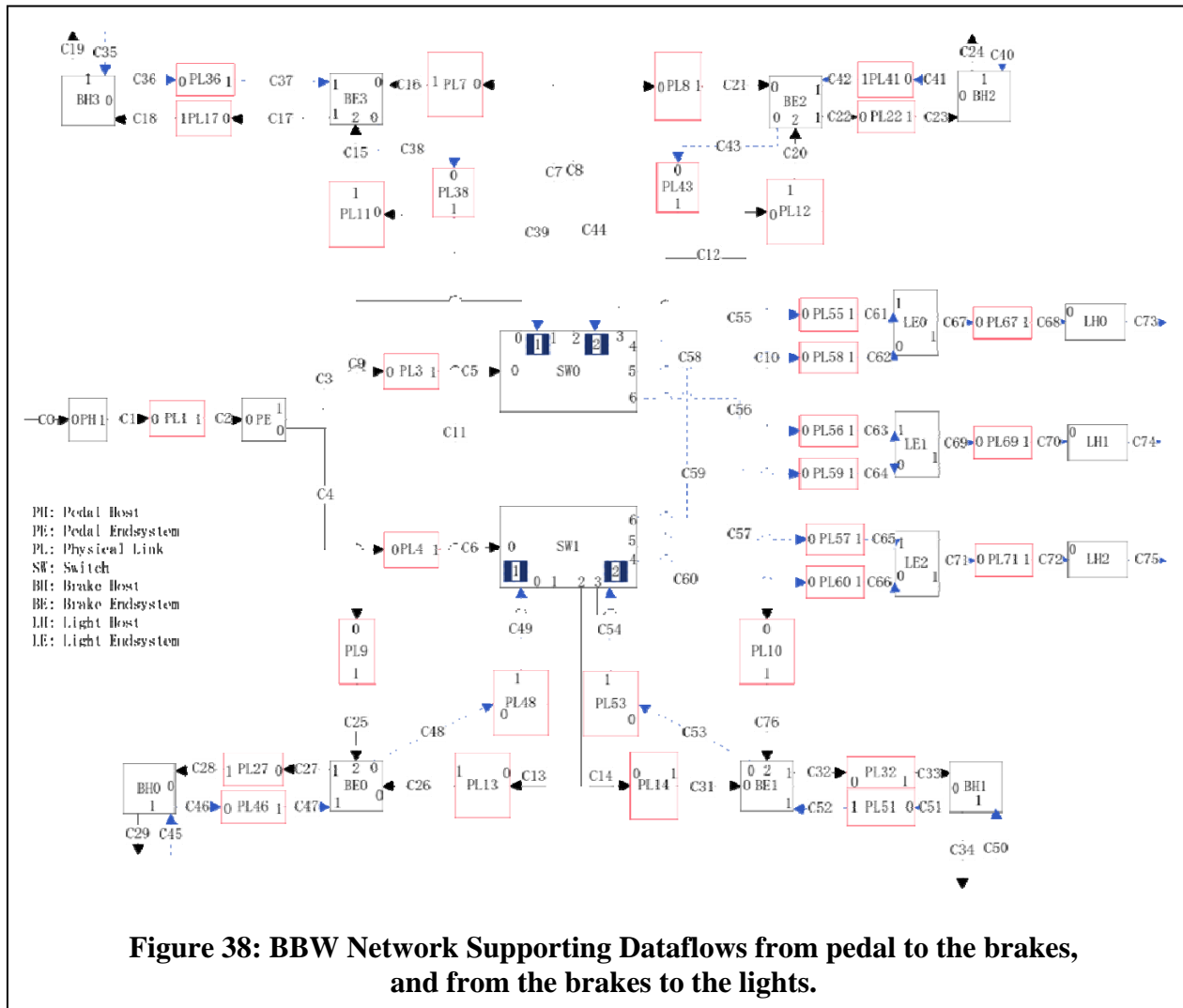
    < hi-host(0) | empty | out(0) | hostfail : pr('h0fail) >
    < conn(0) | out(0) to in(0) >
    < hi-es(0) | in(0) | out(1) | esfail : pr('es0fail), fcschfail :
pr('es0fcsfail) >
    < conn(1) | out(1) to in(1) >
    < pl(0) | in(1) | out(2) | plfail : pr('pl0fail)>
    < conn(2) | out(2) to in(2) >
    < hi-sw(0) | in(2) | out(3) | swfail : pr('sw0fail), fcschfail :
pr('sw0fcsfail) >
    < conn(3) | out(3) to in(3) >
    < pl(1) | in(3) | out(4) | plfail : pr('pl1fail)>
    < conn(4) | out(4) to in(4) >
    < hi-es(1) | in(4) | out(5) | esfail : pr('es1fail), fcschfail :
pr('es1fcsfail)>
    < conn(5) | out(5) to in(5) >
    < hi-host(1) | in(5) | out(6) | hostfail : pr('h1fail) >
    < conn(6) | out(6) to in(6) >
    df(0 | tt | conn(0 | unknown ),
        conn(1 | unknown ),
        conn(2 | unknown ),
        conn(3 | unknown ),
        conn(4 | unknown ),
        conn(5 | unknown ),
        conn(6 | unknown )) .

```

describes a network with two high-integrity hosts, two end systems, and one high-integrity switch, with the switch connected to the two end systems through two physical links. There is one dataflow from the first host to the second, through all the other components. All the failures are set to some propositional variable (`pr(...)`), as discussed in more detail below. The network described above is a simple straight-line configuration. In Section 5.3.2, BBW Network Specification in Maude, we show how the Maude specification is used to realize the BBW architecture.

5.3.2 BBW Network Specification in Maude

Figure 38 illustrates the network layout for the BBW system, in particular the dataflow from the pedal to the four brakes and from the four brakes to the three lights. Between each two components (host, end system, switch, and physical link) there is a connection (`Cn`) that connects the appropriate ports of each component.



5.3.3 Fault Analysis in Maude

As mentioned above, the Maude network specifications serve two purposes: as a common language for network specifications that can be used by different tools and, more directly, for fault analysis implemented in Maude itself. Here, we discuss the latter.

The fault analysis is based on “rules” (loosely speaking) that define how faults are propagated and introduced by each type of component. As an example, the rule for the hi-es (high integrity end system) type of component looks as follows.

*** HI ES, all traffic types

ceq

< conn(C1) | Out1 to In1 >

< hi-es(ES1) | In1,INS | Out2,OUTS | esfail : ESFail, fcschfail : Fcschfail >

```

    < conn(C2) | Out2 to In2 >

df(DF1 | TType |

    conn(C1 | om : OMin, com : COMin, vSA : VSAin, vDA : VDAin, vSN : VSNin, vLen
: VLENin, vData : VDATAin, vFCS : VFCSin, te : TEin, tl : TLin ),

    conn(C2 | unknown),

    DFS)

=

    < conn(C1) | Out1 to In1 >

    < hi-es(ES1) | In1,INS | Out2,OUTS | esfail : ESFail, fcschfail : Fcschfail >

    < conn(C2) | Out2 to In2 >

df(DF1 | TType |

    conn(C1 | om : OMin, com : COMin, vSA : VSAin, vDA : VDAin, vSN : VSNin, vLen
: VLENin, vData : VDATAin, vFCS : VFCSin, te : TEin, tl : TLin ),

    conn(C2 | om : OMout, com : COMout, vSA : VSAout, vDA : VDAout, vSN : VSNout,
vLen : VLENout, vData : VDATAout, vFCS : VFCSout, te : TEout, tl : TLout ),

    DFS)

if

    OMout := OMin or COMin or (VDATAin and not Fcschfail) or (VFCSin and not
Fcschfail) or TEin or TLin or ESFail /\

    COMout := false /\

    VSAout := VSAin /\

    VDAout := false /\

    VSNout := false /\

    VLENout := false /\

    VDATAout := VDATAin and Fcschfail /\

    VFCSout := Fcschfail /\

    TEout := false /\

    TLout := false .

```

The “rule” is actually an equation – more specifically, a conditional equation (ceq). As such, it has three parts: two terms separated by the equality sign (=), and a condition following the “if” keyword. The two terms are both of sort *NetworkConfiguration*. The equation is applicable to any network configuration, or part of a network configuration, that matches the left side. The

left-side term describes the situation where we have a high-integrity switch with incoming and outgoing connections, with known fault annotations on the incoming connection, and unknown faults on the outgoing connection. The right side has the exact same term except that it has some actual, meaningful annotations on the outgoing connection. Thus, the effect of the rule is to annotate the outgoing connection with a fault annotation. Performing this process for all components using their respective rules is the essence of our fault analysis.

To see how the rule adds the fault annotations, we must look at its condition. The outgoing faults are all variables like OMout, COMout, and so on. These variables are assigned values in the condition. For example,

```
OMout := OMin or COMin or (VDATAin and not Fcschfail) or (VFCSin and not Fcschfail)
or TEin or TLin or ESFail
```

Assigns a value to the OMout variable. The assigned value is a logical formula, and it depends on the values of a number of other variables, viz., certain incoming faults (OMin, COMin, VDATAin, VFCSin, TEin, and TLin) as well as certain failures in the component itself (ESFail and Fcschfail). Essentially, the high integrity end system can stop many types of faults and turn them into omission faults. This is known as *fail-silent* operation.

The syntax for the logical formulas used to describe the conditions under which a fault occurs is defined in the FORMULA module

```
fmod FORMULA is
  pr QID .
  sort Formula .
  sort Var .
  subsort Var < Formula .
  op pr : Qid -> Var [ctor] . *** Variables

  *** These have the same precedence values as their boolean counterparts
  op _and_ : Formula Formula -> Formula [assoc comm prec 55] .
  op _or_ : Formula Formula -> Formula [assoc comm prec 59] .
  op not_ : Formula -> Formula [prec 53] .
  op true : -> Formula [ctor] .
  op false : -> Formula [ctor] .

  *** Basic simplifications
  vars P Q P1 P2 A B C : Formula .
  eq P and P = P .
  eq P or P = P .
  eq true or P = true .
  eq true and P = P .
  eq false or P = P .
  eq false and P = false .
  eq P and not P = false .
endfm
```

Maude also has built-in modules for Boolean logic, but we need more precise control over the computations done with these formulas than we would get by using the built-in support.

Formulas are built using the usual Boolean operations (and, or, not), the basic truth values (true, false), and propositional variables (pr). There are also a few equations that perform trivial simplifications on formulas. Many other logical rules are valid and could be written as equations, but more is not needed for our purposes, and in some cases certain rules could be detrimental, as will become clear shortly.

Once we have a network configuration, we can simply ask Maude to execute our equations in order to add fault annotations to the network. For example, if we reduce the example network in the previous section, we get this result:

```
< hi-host(0) | empty | out(0) | hostfail : pr('h0fail) >
< conn(0) | out(0) to in(0) >
< pl(0) | in(1) | out(2) | plfail : pr('pl0fail) >
< conn(1) | out(1) to in(1) >
< pl(1) | in(3) | out(4) | plfail : pr('pl1fail) >
< conn(2) | out(2) to in(2) >
< hi-sw(0) | in(2) | out(3) | swfail : pr('sw0fail), fcschfail : pr(
'sw0fcsfail) >
< conn(3) | out(3) to in(3) >
< hi-es(0) | in(0) | out(1) | esfail : pr('es0fail), fcschfail : pr(
'es0fcsfail) >
< conn(4) | out(4) to in(4) >
< hi-es(1) | in(4) | out(5) | esfail : pr('es1fail), fcschfail : pr(
'es1fcsfail) >
< conn(5) | out(5) to in(5) >
< hi-host(1) | in(5) | out(6) | hostfail : pr('h1fail) >
< conn(6) | out(6) to in(6) >
df(0 | tt | conn(0 | om : pr('h0fail), com : false, vSA : false, vDA : false, vSN :
false, vLen : false, vData : false, vFCS : false, te : false, tl : false ),
      conn(1 | om : (pr('es0fail) or pr('h0fail)), com : false, vSA : false,
vDA : false, vSN : false, vLen : false, vData : false, vFCS : pr('es0fcsfail), te :
false, tl : false ),      conn(2 | om : (pr('es0fail) or pr('h0fail)), com : false, vSA
: false, vDA : false, vSN : false, vLen : false, vData : pr('pl0fail), vFCS :
pr('es0fcsfail), te : false, tl : false ),
      conn(3 | om : (pr('es0fcsfail) or pr('pl0fail) or pr('sw0fail) or not
pr('sw0fail) and (pr('es0fail) or pr('h0fail))), com : false, vSA : false, vDA :
false,
vSN : false, vLen : false, vData : false, vFCS : false, te : false, tl : false ),
      conn(4 | om : (pr('es0fcsfail) or pr('pl0fail) or pr('sw0fail) or not
pr('sw0fail) and (pr('es0fail) or pr('h0fail))), com : false, vSA : false, vDA :
false,
vSN : false, vLen : false, vData : pr('pl1fail), vFCS : false, te : false, tl : false
),
      conn(5 | om : (pr('es0fcsfail) or pr('es1fail) or pr('pl0fail) or
pr('sw0fail) or pr('pl1fail) and not pr('es1fcsfail) or not pr('sw0fail) and
(pr('es0fail) or pr('h0fail))), com : false, vSA : false, vDA : false, vSN : false,
vLen : false, vData : (pr('es1fcsfail) and pr('pl1fail)), vFCS : pr('es1fcsfail), te
: false, tl : false ),
      conn(6 | om : pr('h1fail), com : false, vSA : false, vDA : false, vSN :
false, vLen : false, vData : false, vFCS : false, te : false, tl : false ) )
```

The interesting parts are the fault annotations on the connections – in particular for connection number 5, which is the connection coming into the receiver host. The formulas for this connection are the result of applying the rules for all the previous connections, propagating and introducing different faults along the way. For example,

`vData : (pr('es1fcsfail) and pr('pl1fail))`

indicates that the `vData` fault can happen only if end system 1 has a frame check sequence (FCS) check failure, and physical link 1 fails. Note that the fault annotations were all unknown in the input above.

There is one more step in our fault analysis. While the formulas above describe the exact conditions for a certain type of fault to occur, we are also interested in the *probabilities* (or frequencies) of the faults. We would like to input the probabilities of the various component failures, and calculate the probabilities of the faults. To do this, we interpret the propositional variables as *probabilistic* variables (another reading of the “pr” constructor). For example, for any formula, such as

`(P and Q) or (P and R)`

given the individual probabilities of P, Q, and R, we would like to know the probability of the compound formula. To that end, there are several inference rules of probabilistic logic [23]:

$$\text{Pr}(\text{not } P) = 1.0 - \text{Pr}(P) \quad (20)$$

$$\text{Pr}(\text{true}) = 1.0 \quad (21)$$

$$\text{Pr}(\text{false}) = 0.0 \quad (22)$$

$$\text{Pr}(P \text{ and } Q) = \text{Pr}(P) * \text{Pr}(Q) \text{ if } P \text{ and } Q \text{ are independent} \quad (23)$$

$$\text{Pr}(P \text{ or } Q) = \text{Pr}(P) + \text{Pr}(Q) \text{ if } P \text{ and } Q \text{ are disjoint} \quad (24)$$

where “Pr” denotes “probability of”. The first three rules are easy to handle, but the last two are problematic because of their conditions. Most formulas are not independent or disjoint. While we assume that independence holds for our atomic formulas (the probabilistic/propositional variables), it often does not hold for larger formulas, because they might share some variables. For example, in `(P or Q) and (P or R)`, `(P or Q)` is not independent of `(P or R)` since they both contain P. Disjointness of P and Q means that $\text{pr}(P \text{ and } Q) = 0$. In other words, it is not possible for *both* cases of the “or” to be true at the same time. Again, this is not the case in general. For example, in `(P and Q) or (P and R)`, the two disjuncts could both be true (when all of P, Q, and R, are true) and are therefore not disjoint.

There are different approaches to computing with probabilistic logic [24]. In general, it is very expensive, and there are algorithms for computing approximate results. However, we have adopted an exact approach, which has been quite fast with networks that we have tried so far, despite having worst-case exponential complexity in the number of variables.

The idea behind our approach is to convert the formula into a form where

- For each conjunction, the conjuncts are independent, and
- For each disjunction, the disjuncts are disjoint

When we have a formula of that type, we simply apply the rules above to compute the probability of the formula.

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,

$$(P \text{ and } Q) \text{ or } (P \text{ and } R) \quad (25)$$

$$P \text{ or } (Q \text{ and not } R) \quad (26)$$

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

$$(P \text{ and } Q \text{ and } R) \text{ or } (P \text{ and } Q \text{ and not } R) \text{ or } (P \text{ and not } Q \text{ and } R) \quad (27)$$

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 as follows:

Table 7. Truth Table for Example Full Disjunctive Normal Form

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 “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 with which we can easily compute.

Our `network.maude` specification includes the modules `DNF` and `FULLDNF` to convert any formula to full DNF form. The details are omitted here. Next we need to assign probabilities to the probabilistic variables. We do that using what we call a *valuation*, defined in the `VALUTATION` module

```

fmod VALUATION is
  pr FORMULA .
  pr FLOAT .
  sort Valuation .
  op _ := _ : Var Float -> Valuation [ctor] .
endfm

```

Finally, the PROB module pulls it all together and includes the probabilistic rules mentioned above. Note that we do not check for the side conditions, because we know that our full DNF form guarantees that they hold.

```

fmod PROB is
  pr FULLDNF .
  pr SET{Valuation} * (sort Set{Valuation} to ValuationSet) .
  op evaluate : ValuationSet Formula -> Float .
  var V : Valuation . var VS : ValuationSet . vars P Q R : Formula .
  var Var : Var . var F : Float .
  eq evaluate(VS,P) = eval(VS,fulldnf(P)) .

  *** internal, use evaluate above
  op eval : ValuationSet Formula -> Float .
  eq eval((V,VS),P or Q) = eval((V,VS),P) + eval((V,VS),Q) .
  eq eval((V,VS),P and Q) = eval((V,VS),P) * eval((V,VS),Q) .
  eq eval((Var := F,VS),Var) = F .
  eq eval(VS,not P) = 1.0 - eval(VS,P) .
  eq eval(VS,true) = 1.0 .
  eq eval(VS,false) = 0.0 .
endfm

```

The evaluate operator takes a set of valuations and a formula and calculates the probability of the formula given the valuations. The NETWORK module includes some operators to make it easier to extract some or all of the fault annotations of a network configuration and evaluate them. For example, to calculate the probabilities on connection 5 in the example above, with some given probabilities for the probabilistic variables, we execute the Maude command

```

red evaluateFaultAnnotations((pr('h0fail) := 1E-3, pr('es0fail) := 1E-3,
pr('es0fcsfail) := 1E-9, pr('pl0fail) := 1E-6, pr('sw0fail) := 1E-3,
pr('sw0fcsfail) := 1E-9, pr('pl1fail) := 1E-6, pr('es1fail) := 1E-3, pr('es1fcsfail)
:= 1E-9, pr('h1fail) := 1E-3),
getFaultAnnotations(getDataflow(net1,0),5)) .

```

and get the result

```

om : 3.9959970059990028e-3,
com : 0.0,
vSA : 0.0,
vDA : 0.0,
vSN : 0.0,
vLen : 0.0,
vData : 1.0000000000000001e-15,

```


vFCS : 1.0000000000000001e-9,
te : 0.0,
tl : 0.0

5.3.4 Performance Analysis Using Time-Triggered and Rate-Constrained Communication Paradigm

Networks for real-time systems have stringent end-to-end latency and jitter requirements. One cost-efficient way to meet these requirements is the time-triggered communication paradigm, which pre-plans the transmission points of the frames in the network off-line. This plan prevents contentions of frames on the network and is called a time-triggered schedule (tt-schedule).

In general the tt-scheduling is a bin-packing problem, known to be NP complete, where the complexity is mostly driven by the topology freedom of the network, its associated hardware restrictions, and application-imposed constraints. Multi-hop networks, in particular, require the synthesis of path-dependent, communication-link schedules in order to maintain full determinism of time-triggered communication from sender to receiver.

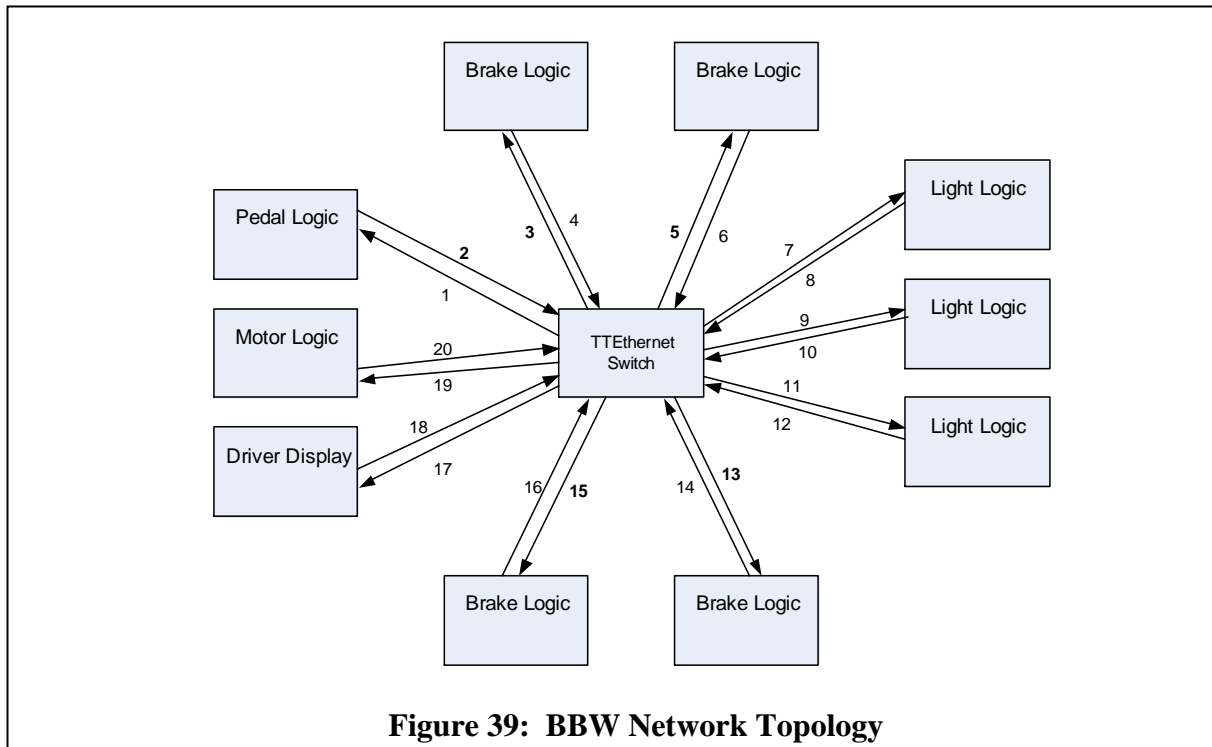
In previous experiments using the Yices Satisfiability Modulo Theories (SMT) solver, we have shown that the scheduling problem can be solved by Yices out-of-the-box for a few hundred random frame instances on the network. A customized tt-scheduler using Yices as a back-end solver allows to increase this number of frame instances up to tens of thousands.

Time-triggered communication provides minimal latency and jitter guarantees. However, the rate-constrained multicast paradigm is a standard communication paradigm used in modern avionics networks (e.g., in form of ARINC 664-p7) and also gains attraction in adjacent industrial markets. In the multicast paradigm, a single message that is dispatched by a sender can be forwarded to a group of receivers. The forwarding function is typically executed by a network switch, which receives a multicast message on one port and forwards the message to a pre-configured set of outgoing ports. Messages will typically arrive unsynchronized at the switches, and consequently the maximum queuing delays have to be calculated to determine the transmission latency of a message in the rate-constrained paradigm.

For the analysis of networks that communicate rate-constrained messages we have previously developed a second tool that checks the end-to-end latency. This tool, the rc-checker, incorporates the Yices SMT solver as well. In addition to pure analysis, the SMT-based approach is also capable of assigning messages to nodes in a way such that the network usage remains balanced.

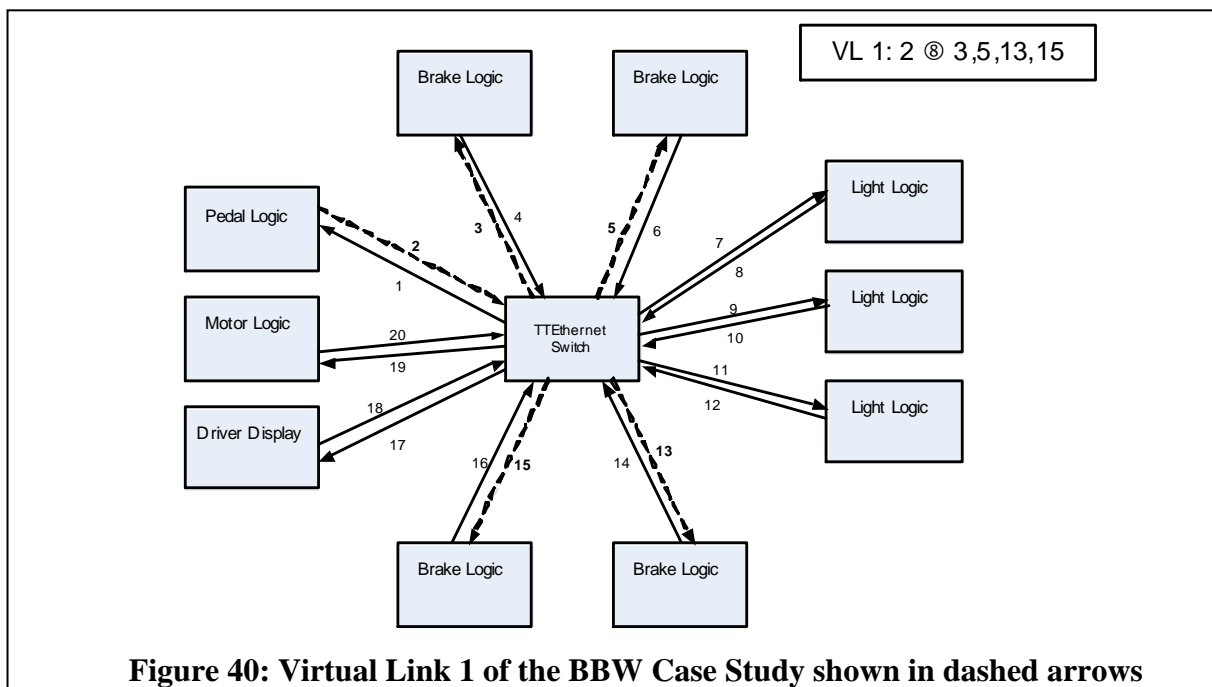
In the following we report on using the SMT-based tt-scheduling and rc-checking approach in a brake-by-wire case study.

Figure 39 depicts the network representation for the “tt_scheduler” and “rc_checker.” Each physical Ethernet connection (each “wire”) is represented by two arrows – for example, the pair of arrows 1,2 represents one wire. The file “BBW.top” specifies the topology of the BBW case study (it can be used as input to an automatic traffic generator).



Using the representation of the topology as specified in Figure 39, we can define virtual links (VLs).

Figure 40 shows the example of VL 1 (see Table 6 for all VLs).



The file “BBW.top.msg” specifies VL 1 – VL 5 from the BBW case study (VL 1 is highlighted in red):

5;50000;20;100;10;500;10;0;0

1;15;4;1

paths:

2;3;0;0;0;0;0;0;0;0;

2;5;0;0;0;0;0;0;0;0;

2;13;0;0;0;0;0;0;0;0;

2;15;0;0;0;0;0;0;0;0;

splits:

3;5;13;15;0;0;0;0;0;0;

2;15;1;0

paths:

4;19;0;0;0;0;0;0;0;0;

splits:

3;15;1;0

paths:

6;19;0;0;0;0;0;0;0;0;

splits:

4;15;1;0

paths:

14;19;0;0;0;0;0;0;0;0;

splits:

5;15;1;0

paths:

16;19;0;0;0;0;0;0;0;0;

splits:

task_dependencies

0;0;0

rand_msg

1;2;3;4;5;

5.3.4.1 Time-Triggered Communication Paradigm

BBW.top.msg is the input to the TT scheduler (tt_scheduler_adv), which can be executed by

```
./tt_scheduler_adv BBW.top.msg -f 1
```

When the scheduler finds a feasible TT schedule, it generates an input file for gnuplot™. Gnuplot can then be used to generate the plots in an appropriate file type. The schedule of VL 1 – VL 5 of the BBW case study is shown in Figure 41. The plot depicts the dataflow links on the x axis and the slots of the TT schedule on the y axis. VL 1 is represented by “1”, VL 2 by “2”, and so forth.

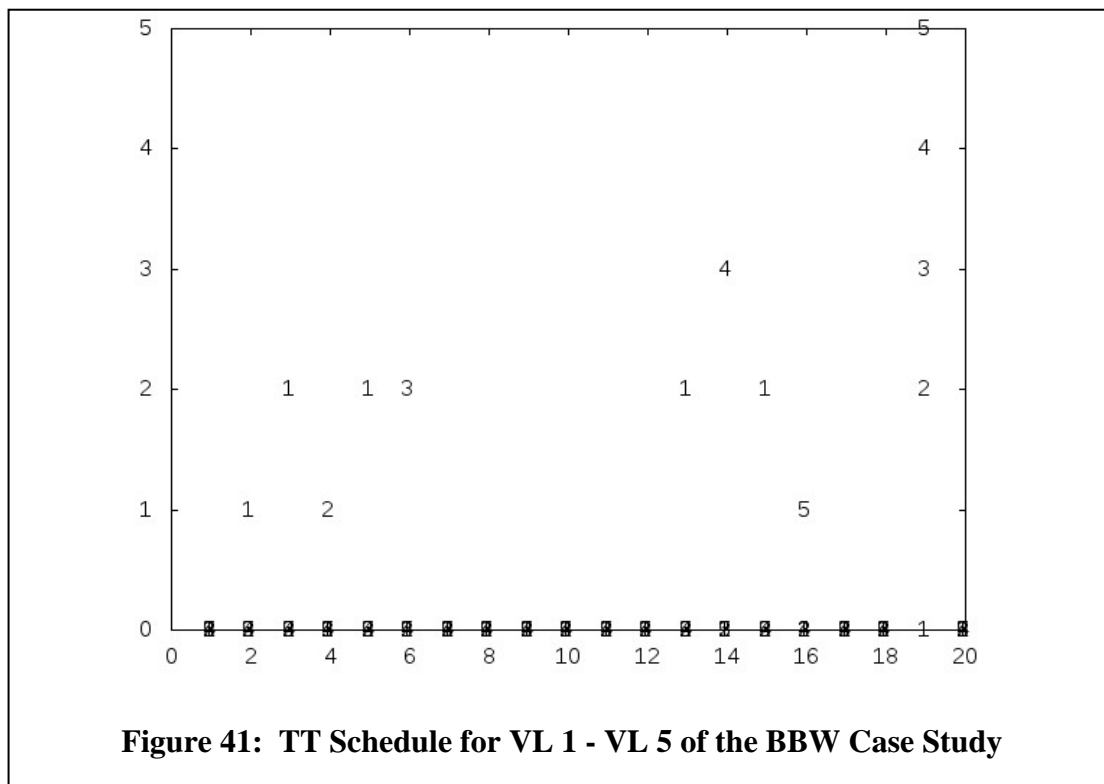
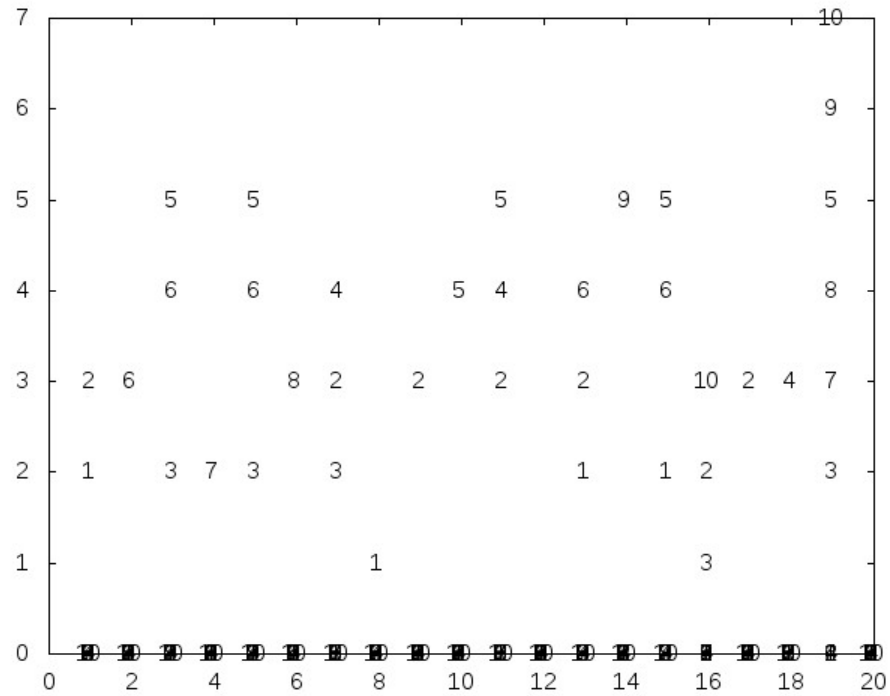
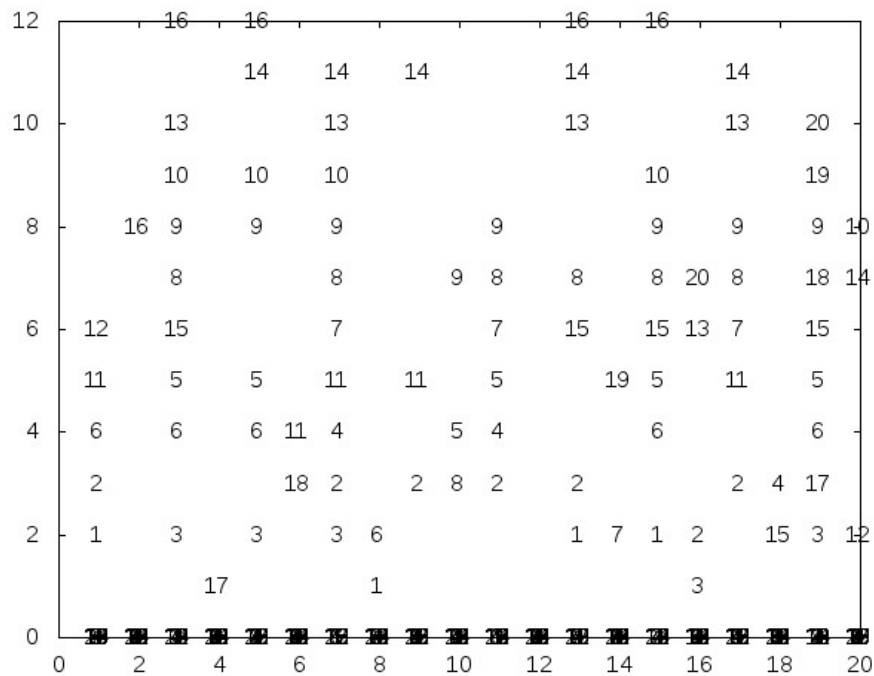


Figure 41: TT Schedule for VL 1 - VL 5 of the BBW Case Study

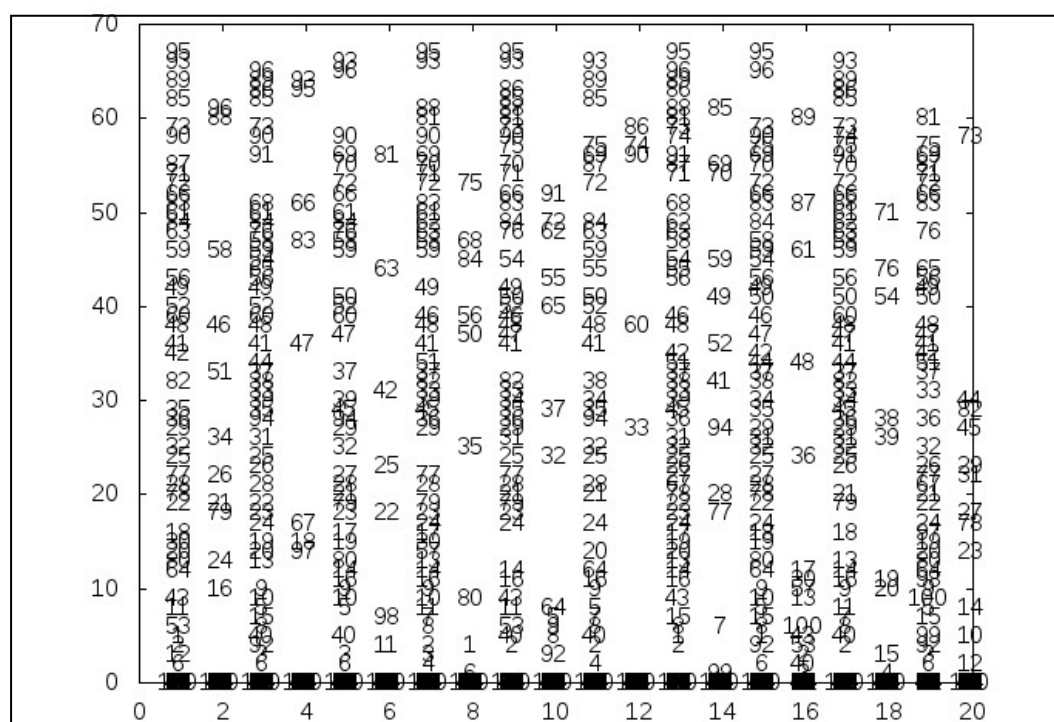
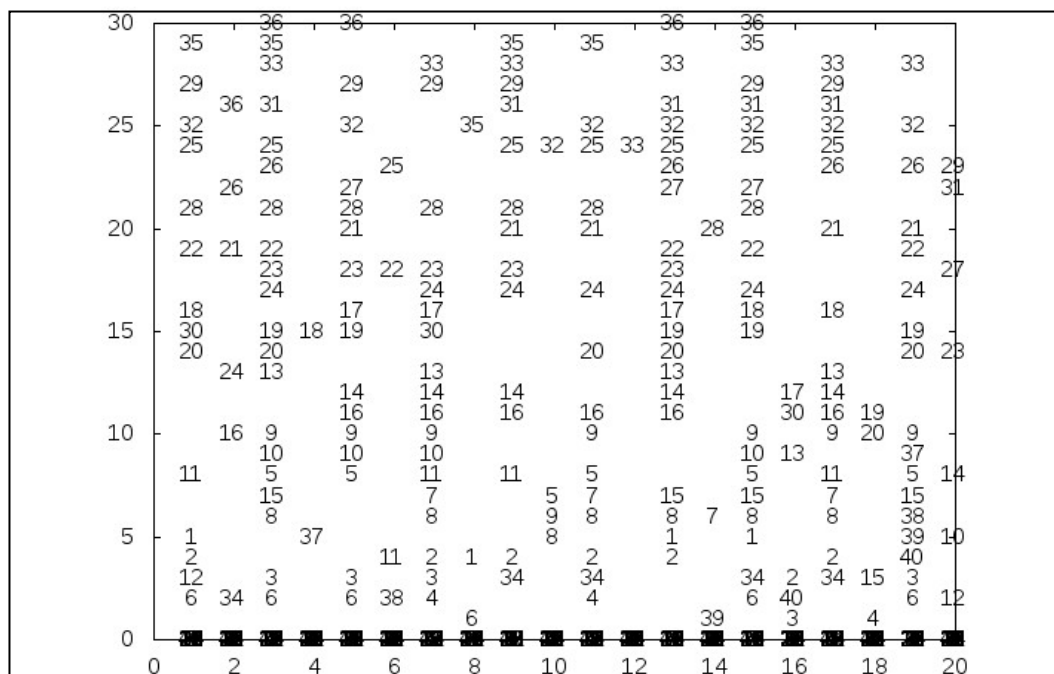
To demonstrate the tt scheduling under higher dataloads we can add artificial “dummy” data. We schedule the dummy data first and then add the VLs from the BBW case study (see Figures 42-45).



**Figure 42: TT Schedule with Five Additional Frames (VL IDs 1-5),
BBW VL IDs 1-5 are Translated to VL IDs 6-10**



**Figure 43: TT Schedule with 15 Additional Frames (VL IDs 1-15),
BBW VL IDs 1-5 are Translated to VL IDs 16-20**



**Figure 45: TT Schedule with 95 Additional Frames (VL IDs 1-95),
BBW VL IDs 1-5 are Translated to VL IDs 96-100**

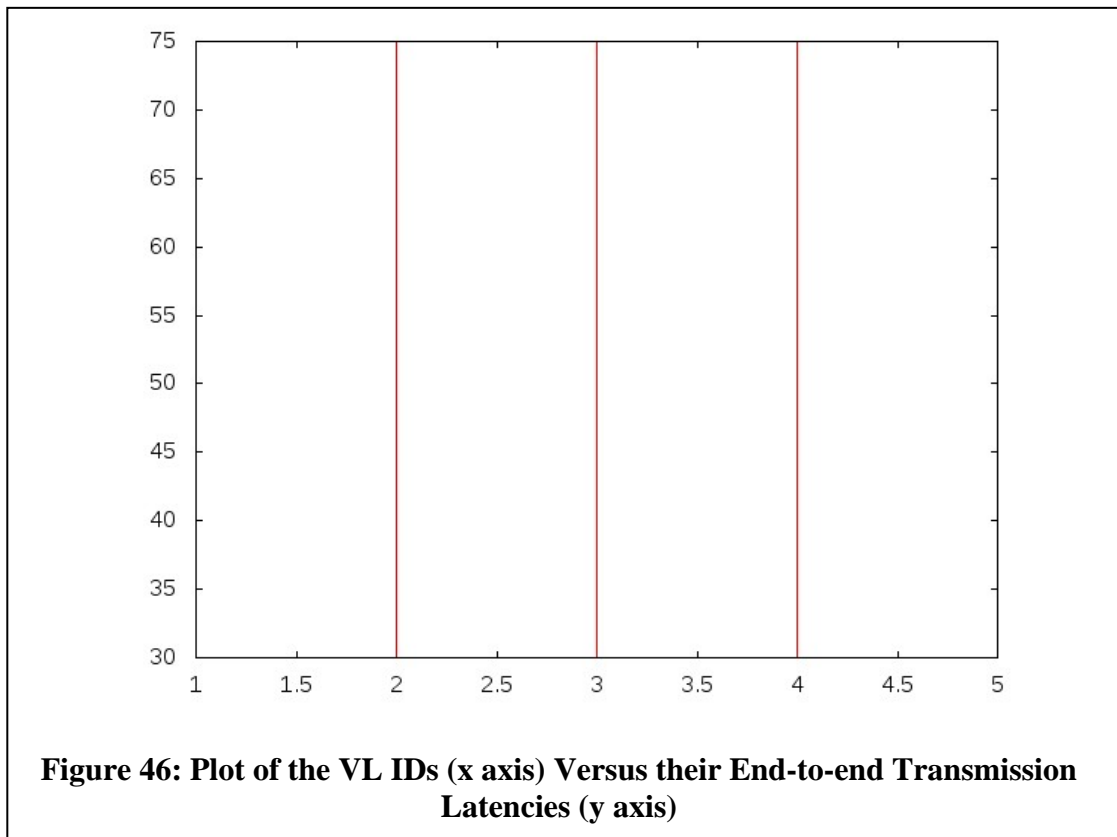
5.3.4.2 Rate-Constrained Communication Paradigm

The topology file BBW.top and the messages file BBW.top.msg are the inputs to the RC traffic analyzer (rc_checker), which can be executed by

```
./rc_checker BBW.top BBW.top.msg -u -n 5
```

The rc_checker tool then approximates the end-to-end latency of the messages specified in the message file considering the topology specified in the topology file (the -n parameter specifies that only the first five messages from the message file will be analyzed). The result of the analysis can be plotted by gnuplot™ as for example in Figure 46. The x axis lists the VL IDs.

For simplicity of description of the examples we assume that the unit on the y axis is microseconds. In the scenarios, frame lengths are randomly distributed in an interval [1:100]. In a 100 Mbit/s Ethernet system the message lengths will be [6:124] microseconds and in a 1 Gbit/s Ethernet system in [0.6:12.4] microseconds. So, when we say the unit on the y axis is 1 microsecond, the frame sizes in the example are representative for a 100 Mbit/s Ethernet (or TTEthernet) system. Figures 46-49 show the results for various loads on the network (i.e., additional 15, 35, or 95 frames).



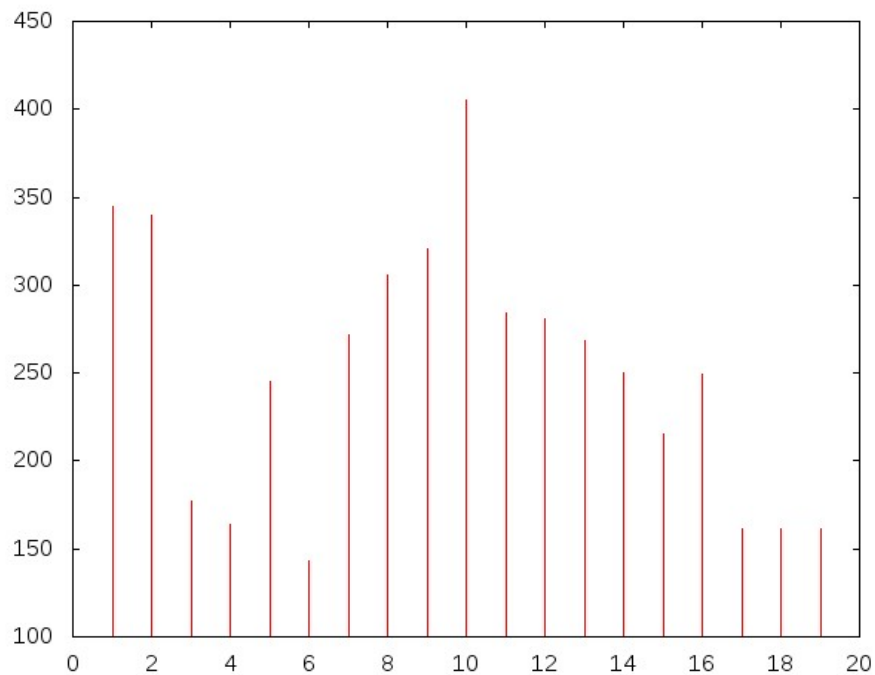


Figure 47: Plot of the VL IDs (x-axis) Versus their End-to-end Transmission Latencies (y axis) Considering 15 Additional Dummy Frames

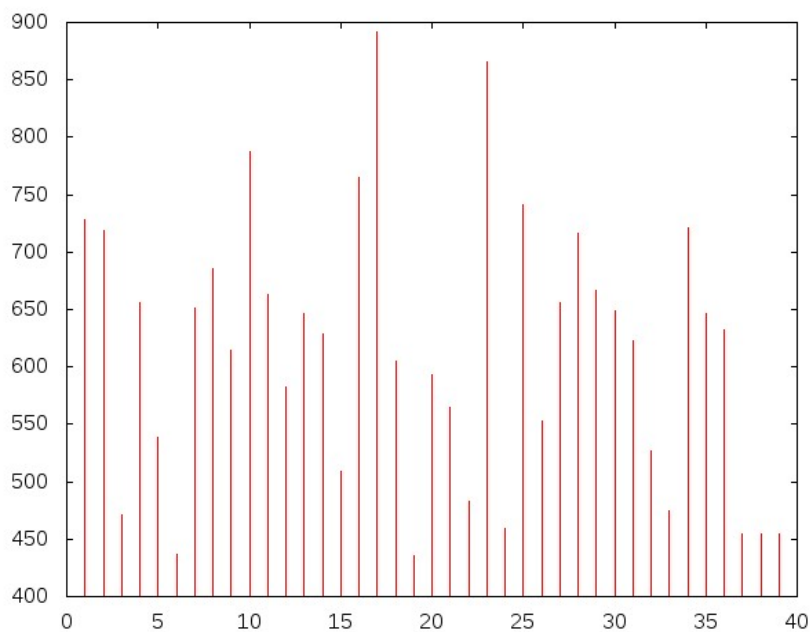


Figure 48: Plot of the VL IDs (x axis) Versus their End-to-end Transmission Latencies (y axis) Considering 35 Additional Dummy Frames

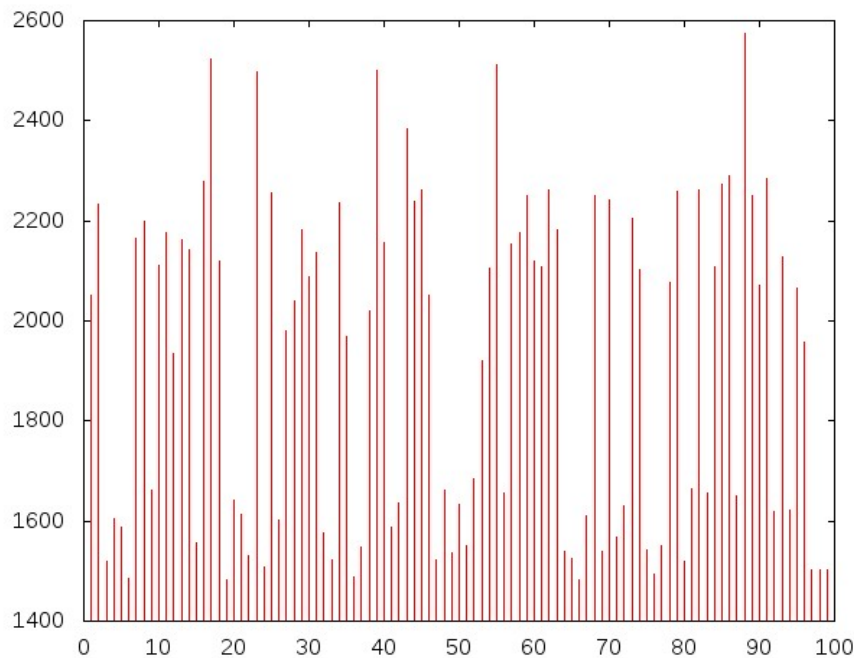


Figure 49: Plot of the VL IDs (x axis) Versus their End-to-end Transmission Latencies (y axis) Considering 95 Additional Dummy Frames

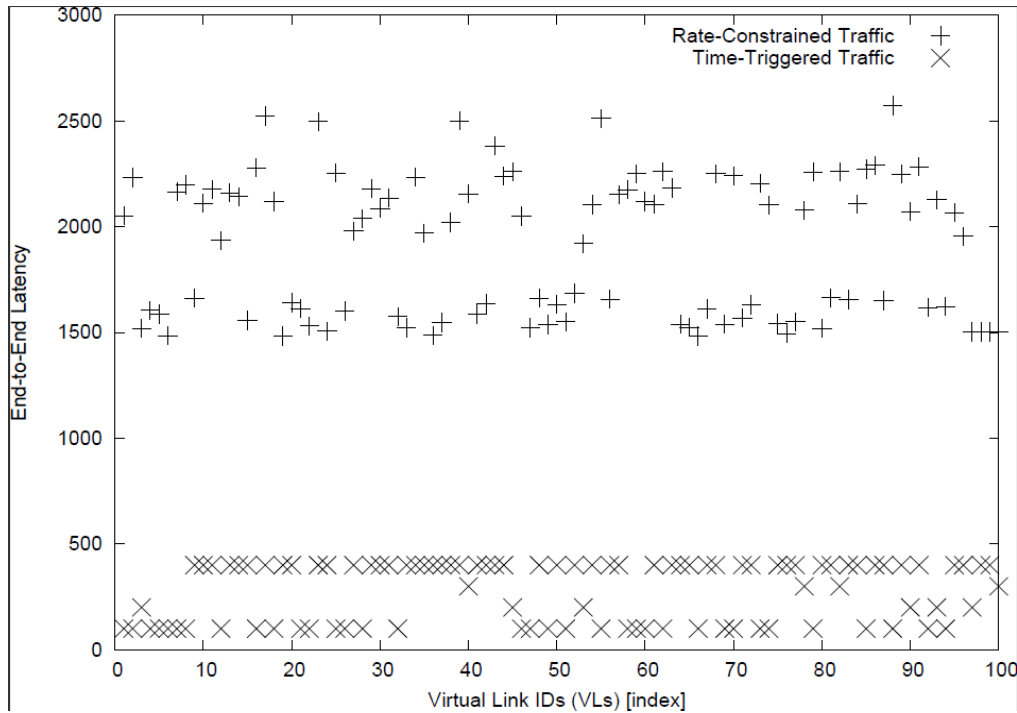
The RC tool calculates the end-to-end latency in two parts. In the first part, it calculates the maximum time a frame may be buffered in each node (e.g., a switch) before it can be relayed. In the second part, it calculates the number of physical links on which a frame is transmitted (i.e., always two in the BBW example). This number is then used as a multiplier of the frame length (this is pure time it takes to transmit a frame over the physical links, without considering the buffering time in the switches). The sum of the first part and the second part is the end-to-end latency as depicted in the figures for RC traffic.

Part one is a bit more complex to calculate: essentially if you had a switch with three inputs IN1, IN2, and IN3 and two outputs OUT1 and OUT2, then the tool calculates the queuing delay of the frames transmitted on OUT1 as the sum of all frames received on IN1, IN2, and IN3 that are relayed on OUT1, minus the maximum sum of all frames received on IN1, IN2, and IN3 relayed on OUT1. Essentially, the tool is concerned with "burst scenarios" in which all frames that may be sent on OUT1 are sent back to back on IN1, IN2, and IN3.

5.3.4.3 Comparison of TT vs. RC End-to-End Performance

For TT, again, we assume message sizes randomly distributed in [1:100], and we assume that a slot is 100 microseconds long. (Here we are omitting overheads that would arise from the clock synchronization. If one wanted to include this as well, we could add another microsecond or so more for the slot length).

For the end-to-end latency of a TT frame the tool simply sums up the slots from the transmission on the first physical link until the transmission on the last physical link; for example, if a VL is transmitted initially in slot 1 and then to its final destination in slot 3 then 1, calculate the end-to-end latency as $(3-1)+1 * \text{slot_length}$.



5.3.4.4 The ASIIST Tool

ASIIST stands for “Application Specific I/O Integration Support Tool” and was developed by the University of Illinois at Urbana-Champaign. Here, we discuss the relation of the ASIIST tool to the tt scheduling and rc checking as presented earlier in this report. The basis for our assessment is the set of presentations available via youtube [25,26] and the slide deck “ASIIST-IMA Partitioned System Analysis.”

As it appears from the aforementioned presentations, ASIIST is primarily concerned with analysis and scheduling inside a single end system (also called “node” or “LRU” [line-replaceable unit] in the avionics context). In particular, the end system targets at integrated modular avionics (IMA), so that the single end system may host a multitude of processors. Each processor in turn can be configured to host several partitions. Typically, a partition is assigned an exclusive part of the memory and a guaranteed percentage of processor time (e.g., by a fixed-length time slot in case the processor time is defined according to a round-robin scheduling scheme).

The “scheduling” problem in such an environment is to assign application tasks to partitions. When a system is not schedulable, a reason might be that the set of tasks assigned to one partition requires too much processor time. Consequently, enlarging the time slot of the respective partition may make the system schedulable.

Requirements for Running the GUI

The GUI application itself is written in Java 6, so this runtime environment must be installed. To access all the tools from Java, we assume a Unix shell like bash and standard Unix tools such as “awk” and “sed.” Furthermore, the GUI calls Maude, which we tested in version 2.6. The network graph is generated using graphviz’ dot in version 2.28.0. The executable must be located (or linked from) /usr/local/bin/dot. When using the TT scheduler tools, these must be compiled for the specific platform and located in a directory to be supplied to the Java application. The TT scheduler tools also require gnuplot and perl.

Using the GUI

We supply a shell script called “gui.sh” to wrap the call to the Java application with all the dependencies. A command line switch -h prints the usage and options:

```
$> ./gui.sh -h

usage: java -cp ... com.sri.promise.gui.GUI [options...]

with:

FILE      : choose given Maude file
-h        : display usage and exit
-l LEVEL  : set console log level
-mExe FILE : path to Maude executable (default is $MAUDE_LIB/maude)
-mLib PATH : path to MAUDE_LIB directory -- mandatory!
-sDir PATH : path to scheduler directory -- mandatory!
```

To invoke the GUI with a certain Maude file chosen for analysis, we use for example:

```
$> ./gui.sh -mLib $MAUDE_LIB \
          -mExe /usr/local/bin/maude \
          -sDir ../Performance/demo-20110719 \
          ../Maude/BBW/brake-by-wire-AllHI.maude
```

Once the GUI is open, we enter the Maude operator to be reduced into a NetworkConfiguration, here “bbw6”, into the text field below the Maude file (see Figure 51).

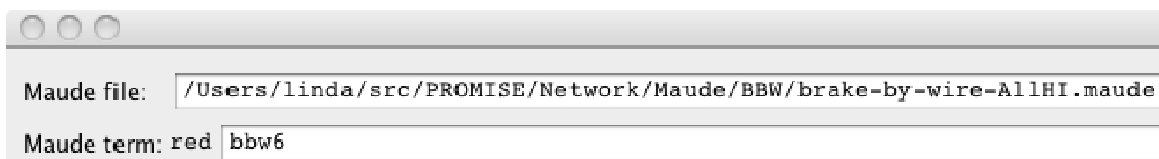


Figure 51: Analyzing Dataflow “bbw6”

Then, we can run Maude using the button or simply hit <ENTER> when done typing the operator. Maude should run in the background and the result is printed in the text area in the middle. If we find the strings "result NetworkConfiguration:" and "Bye." inside this text, the text between these two terms is automatically selected. Without changing the automatic text selection, simply hitting <SHIFT>-<ENTER> generates the graph corresponding to the network configuration. Figure 52 shows part of the resulting graph.

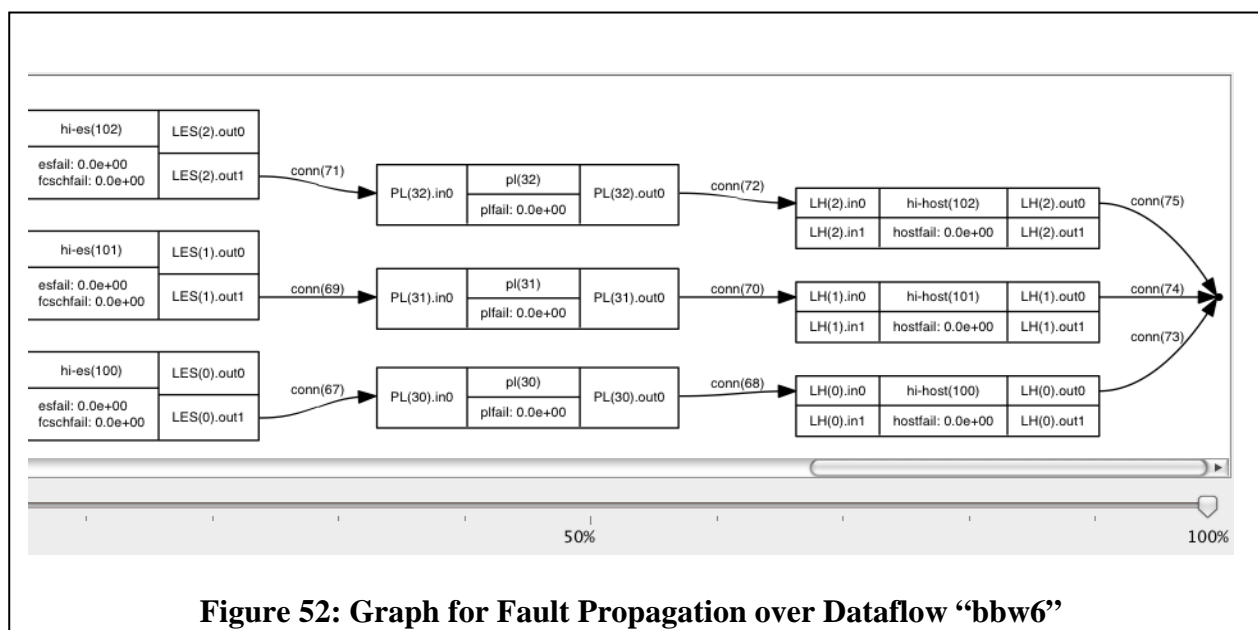


Figure 52: Graph for Fault Propagation over Dataflow “bbw6”

Once the graph is showing, the user can zoom in and out, zoom to fit using a double click, and scroll with the bars and mouse wheel.

To further investigate the failure values for an existing connection, the user enters the connection number into the corresponding text field and hits <ENTER> or the adjacent button. Maude will print the result as a

NeSet{FaultProbAnnotation} with each line containing a failure value as seen in Figure 53.

```

-- Running maude to analyze connection "74"...
--
--      \|||||
--      --- Welcome to Maude ---
--      /|||||
--      Maude 2.6 built: Dec 10 2010 11:12:39
--      Copyright 1997-2010 SRI International
--      Tue Jul 26 09:13:35 2011
-- Maude> reduce in brake-by-wire :
evalConn(bbw6, 74) .
-- rewrites: 374 in 96ms cpu (97ms real) (3893
rewrites/second)
-- result NeSet{FaultProbAnnotation}:
--   om : 1.0e-4,
--   com : 0.0,
--   vSA : 0.0,
--   vDA : 0.0,
--   vSN : 0.0,
--   vLen : 0.0,
--   vData : 0.0,
--   vFCS : 0.0,
--   te : 0.0,
--   tl : 0.0
-- Bye.
-- ... success running Maude.

```

Figure 53: Fault Propagation at Connection 74 of “bbw6”

To run TTTech's scheduling tool, the user selects the type (time-triggered or rate-constrained) and supplies the number of Virtual Link IDs (must be between 5 and 100). Clicking the adjacent button performs the scheduling analysis and opens the resulting plot in a PostScript viewer.

5.4 Conclusions

Co-optimization of fault tolerance and performance with verification proofs of the general space of architectures is intractable. Our approach is to provide feasible trade-off analysis and verification of selected points in the design space. We support design choices through network architecture design space characterization at network components, host, and application-level redundancy management. Our approach is novel in that we integrate latency, utilization, buffer size and fault tolerance analysis and it is scalable to vehicle-sized network architectures.

6. INTEGRATING VERIFICATION INTO EARLY DESIGN FLOW

6.1 Introduction

We integrated two verification tools into the META workflow: (1) the HybridSAL hybrid model checker and (2) the Honeywell Lifecycle Tools & Environment (HiLiTE). The integration of HybridSAL and HiLiTE into the CyPhy language and design environment enhances the META workflow with essential verification capabilities: (1) model checking of dynamic properties based on qualitative and relational abstractions and (2) static analysis of signal properties in large-scale, embedded control software. Further, we adopted a general approach, which is independent of the verification tools and provides the infrastructure to integrate additional tools. The implementation provides generic templates for: (1) integrating models suitable for verification and linking them to architecture models in CyPhy and (2) incorporating verification certificates into the system models that can be used in the design space exploration.

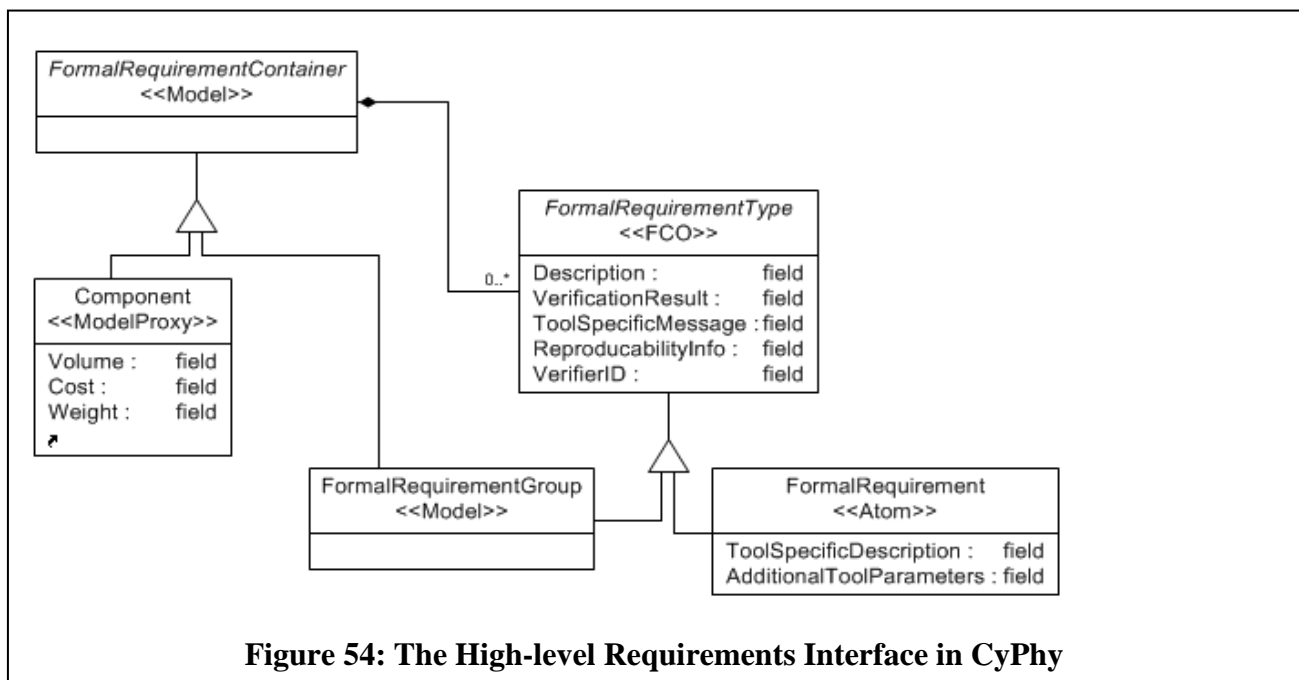
6.2 Methods, Assumptions, and Procedures

6.2.1 High-Level Integration Concepts

6.2.1.1 High-Level Requirements Interface

CyPhy is an integration language: thus, it does not contain all the information of the specifics of the models – it only references them. The verification tool interface must comply with this notion, which means that it is necessary to have a requirements language that is tool independent on the language level, but can contain verification tool-specific information on the modeling level. Moreover, this requirement language must capture all the necessary information to verify the integration, enable the selection of components with specific formally verified properties, and design space exploration.

Figure 54 depicts this requirement language. One can define one or more formal requirements, and attach it to a component. For more complex situations, the formal requirements can be recursively grouped. The requirement contains a human-readable description, a verification result returned by the verification tool along with a more detailed tool-specific message. Formal requirements were carefully designed to support reproducibility. The reproducibility information contains all the information to repeat the execution of the tools with exactly the same model and environment. Finally, a formal requirement can be assigned to a tool-specific integrator component, referred to as verifier and described in detail in the next section.



6.2.1.2 Tool Integration Architecture

The integration architecture must be open to accommodate tools with various interfaces. To this end, we use a generic tool integration architecture, which consists of a general manager component and arbitrary number of plugins called verifiers.

The Vanderbilt Verification Manager is a generic integration component that is able to manage tool-specific plugins, edit the requirements specified in CyPhy in a tool-independent way, and assign tool-specific plugins to the formal requirements. The component supports a wide variety of programming languages (C++, Java, Python, C#, etc.).

Figure 55 illustrates the component while we register a HiLiTe-specific verifier plugin. The tool integration discussed in the next sections is realized by verifier plugins via the Vanderbilt Verification Manager.

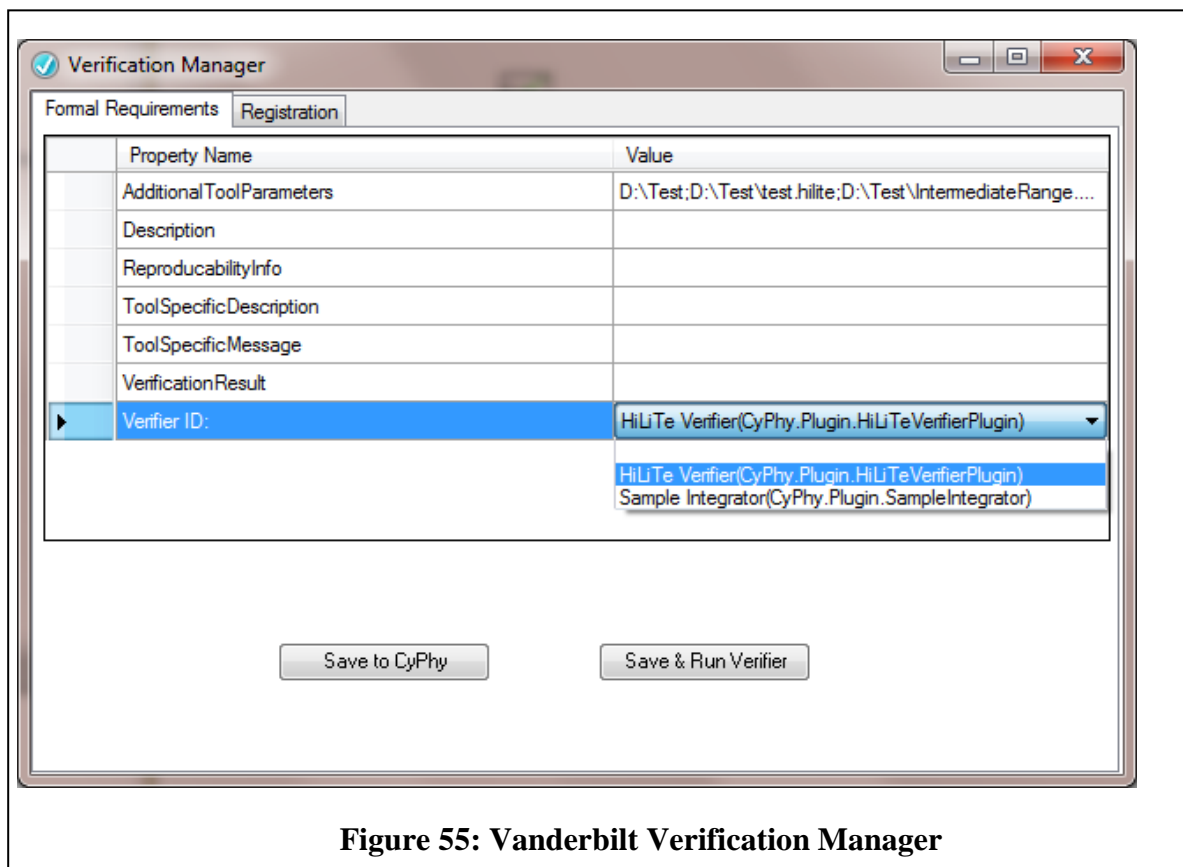


Figure 55: Vanderbilt Verification Manager

6.2.2 Integrating HybridSAL with Design Flow

6.2.2.1 Example Verification Problem

A case study of a cabin air compressor (CAC) provided by Honeywell is the basis of our example description. An overview of the Simulink model is shown in Figure 56.

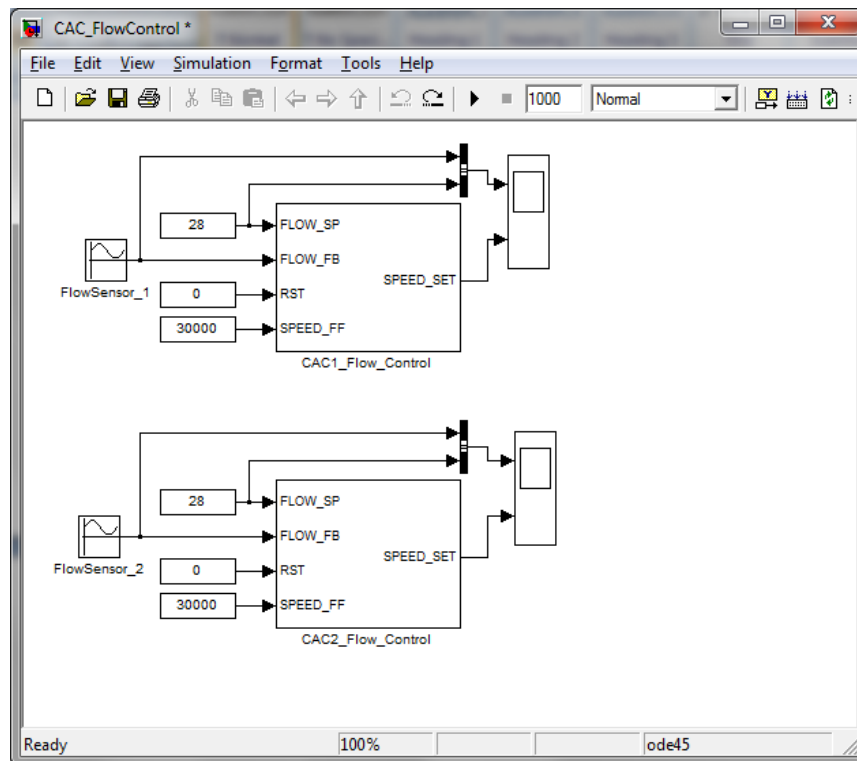


Figure 56: CAC Control Overview

This system has two parallel paths of two compressors; both work together to control the air and the pressure in the cabin. Although the two systems work at the same rate – that is, they must provide the same rate of flow – the speed of the compressors can be different. The figure depicts an open loop simulation; in practice, *FlowSensor_1/FlowSensor_2* originate from the plants.

Figure 57 depicts the internals of the flow control boxes. Essentially, the solution is a PI control realization. The signal *FLOW_SP* is the set point coming from an upper layer, while *RST* is a reset signal. Moreover, *FLOW_FB* is the feedback – that is, the actual sensed data. *SPEED_FF* is used when the sensor fails. It is a signal describing the altitude; from the altitude an approximation can be computed and used instead of the erroneous feedback.

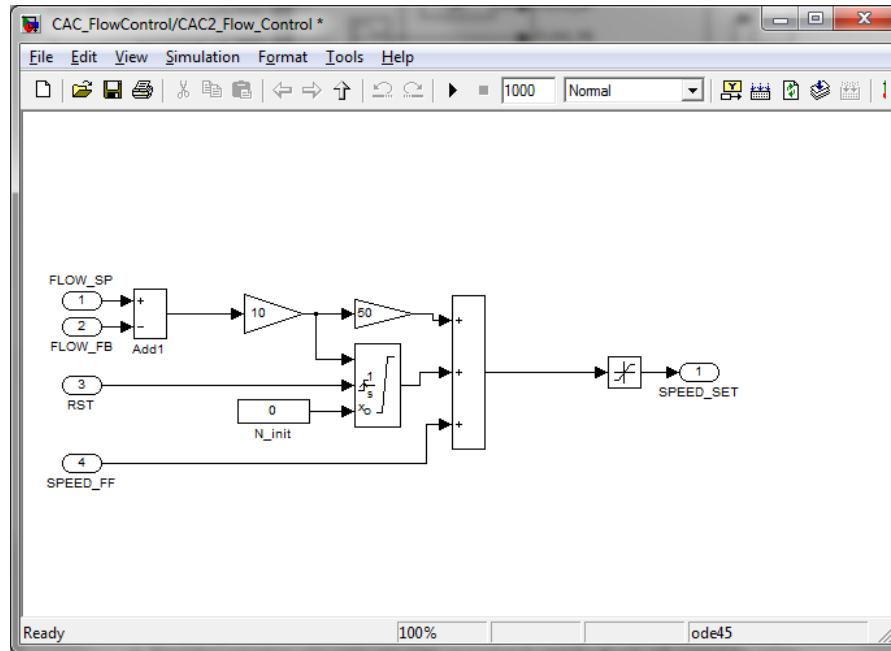


Figure 57: Flow Control Internals

6.2.2.2 CyPhy and other Tools

The META language, called CyPhy, is composed of several modeling languages. The most important for us are the Embedded System Modeling Language (ESMoL), and the Bond Graph language. ESMoL (Figure 58) describes models, and facilitates the step-by-step refinement of the design, along with physical implementation and scheduling in real time environments. The BondGraph language describes plants and processes from different physical domains, such as mechanical, thermodynamical, and electrical. There exists also a requirements language that captures the validation results. The requirements language is integrated with DESERT, the META design space exploration tool. It is significant for us in the sense that the variables in the requirements model store the result of the validation. The relevant part of the META workflow is shown in Figure 59.

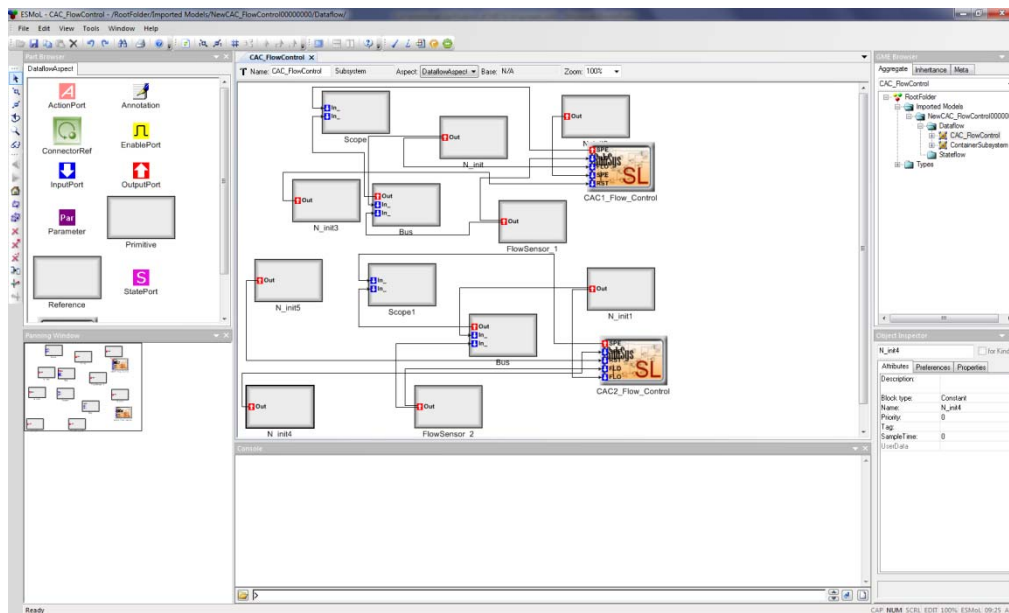


Figure 58: Case Study in ESMoL

The META toolset includes a translator tool from MATLAB Simulink/Stateflow to ESMoL. Currently, this translation supports both discrete and continuous time systems; however, the tool chain makes use of only the discrete part.

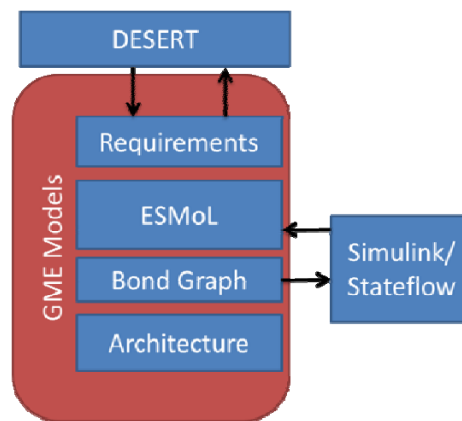
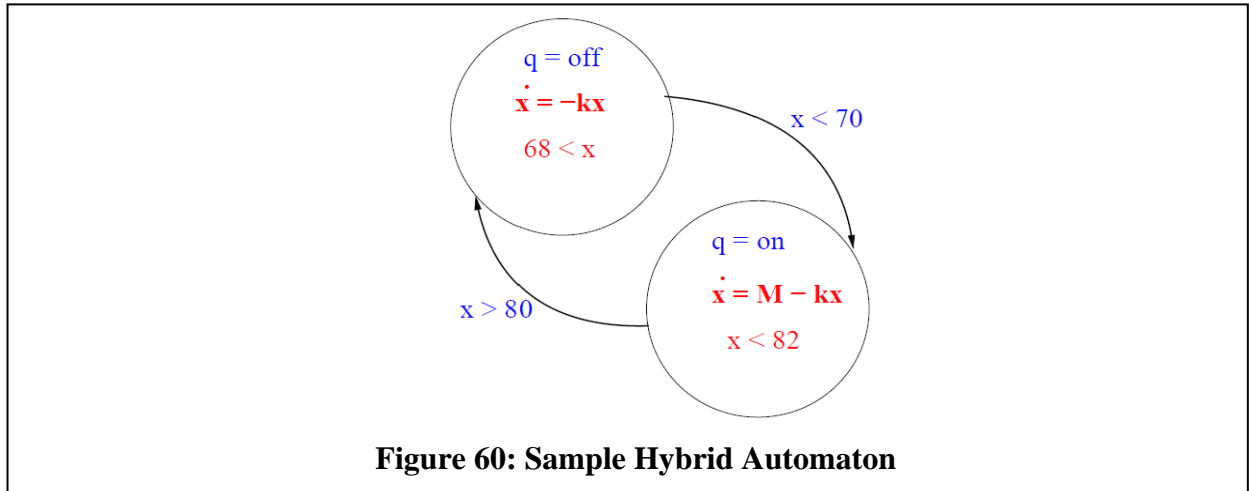


Figure 59: META Workflow

A translator from Bond Graphs to MATLAB is provided as a part of the META workflow. It is conceptually important here that the abstraction level is significantly different: bond graphs are low-level formalisms compared to a Simulink/Stateflow model. In the case of this transformation, this means information loss, which implies that the inverse transformation is hard to create and involves ambiguity.

6.2.2.3 Overview of HybridSAL

HybridSAL is used to verify safety properties of hybrid automata models. A sample hybrid automaton is depicted in Figure 60. Roughly, a hybrid automaton is a state machine, where the states contain ODE-s (state space description of a linear continuous time system), and the transitions are enabled by Boolean expression defined over the state variables.



The hybrid automata representation used by HybridSAL differs from the general notion because the transitions are not taken immediately when the guard conditions enable them. HybridSAL uses state invariants (last line in the states) to force a transition before it is too late – that is, they invalidate the state. HybridSAL takes a textual representation as its input; however, on the conceptual level, it is satisfactory for us to use the graphical representation in this document.

6.2.3 Scalable, Multi-Component Static Verification Integrated with Design Flow

6.2.3.1 Motivation

When integrating several reusable control software components for a new vehicle mission requirement or platform configuration, parameters within the application and platform components need to be selected and tuned. These changes impact the results of computations in a component, which in turn influence correctness of certain properties in a downstream component that uses the results of other components. Even though component designs (models) and code can be parameterized and reused across vehicle configurations, verification must be redone on exact bounds/constraints of the parameters in a set of integrated components for a specific configuration, because the state space of all interactions of generic parameter bounds of multiple components is undecidable and cannot be properly enumerated.

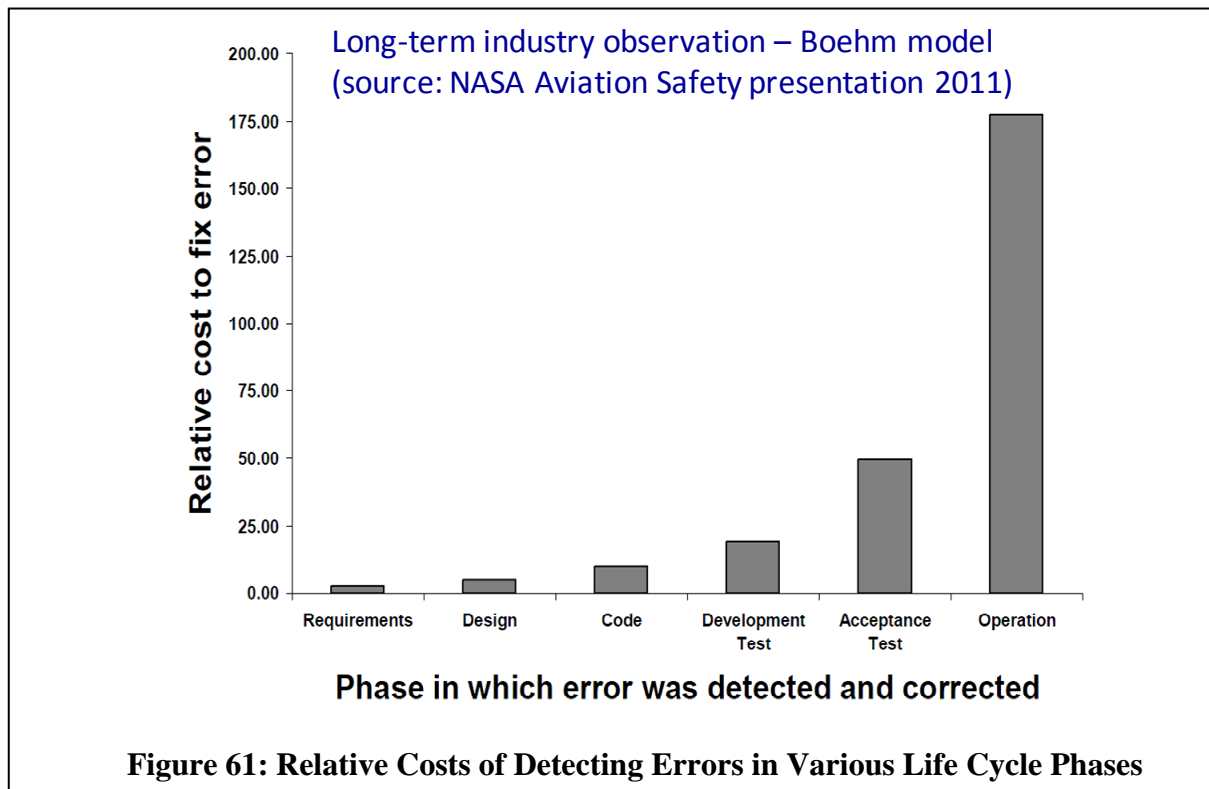
Therefore, not all verification “assumptions” of a component (inputs, environment) can be a priori specified; certain properties must be verified in the integrated context of all components for specific values of configuration parameters. The following classes of properties can be automatically verified by static analysis of all internal function blocks of models and cross-model intermediate signals:

- Signal data type and dimensions compatibility
- Overflow (e.g., divide-by-zero): signal range bounds exceed data type representation limits
- Reachability: e.g., not all branches can be exercised, not all states can be reached
- Control function specific: e.g., time constant of digital filter less than $2 * \text{sampling frequency}$

Benefits and Cost Impact: A significant part of the costs of integrating a large number of reusable components is the verification that the internal design of each component operates correctly in the integrated context and there can be no anomalous behavior to adversely impact system safety. This verification must be done in a comprehensive, automated manner; otherwise, the cost benefits of reuse cannot be realized.

There is also a strong benefit of performing this verification in early design stages on the component algorithm models in the same way integrated simulations are performed currently. Figure 61 illustrates the industry observation on the relative cost impact of detecting errors in various life cycle phases; errors detected late in the development can have a much larger cost impact than if detected in early design. Other industry observations surmise that approximately 40-50% of all defects are injected in model design phases and this contributes to a significant portion of system development cost and time.

In addition to saving cost and time, the design verification of integrated components provides stronger, correct-by-construction integration assurance than by simulation, code analysis, and testing approaches.



Static Analysis Automation Objectives: In order to realize the benefits of detecting design problems early and associated lifecycle cost savings, it is essential to minimize the cost of performing the static analysis across multiple integrated components. Static analysis at the level of detailed design properties can be quite labor intensive, even at the individual component level, using standard verification tools such as model checkers, since the properties need to be manually enumerated and several “invariants” from model semantics need to be translated and/or manually generated. The approach used in META uses the HiLiTE tool that automates the property enumeration and model semantics translation/generation such that no manual intervention is required.

Integrating the analysis across multiple components and into the design flow is also labor intensive in current methods, requiring manual translation and inference steps. Our META approach performs this integration in the CyPhy environment where the intermediate propagation of analysis results can be controlled and also annotated/visualized directly into the design models at the appropriate levels for system architects and application designers. Furthermore, the intermediate analysis results (such as range bounds on control parameters) can be fed back into the design optimization process or can be used for constraining other verification analyses on other specific aspects of the system.

Scalability Objectives: Scalability of the verification automation is extremely important since a specific configuration can consist of hundreds of interconnected MATLAB models used as software components, which can translate to approximately 2-4 M SLOC. In each model, hundreds to thousands of properties must be automatically enumerated and verified, with negligible amounts of human input. This requires that very scalable and fast static analysis methods must be employed for verification.

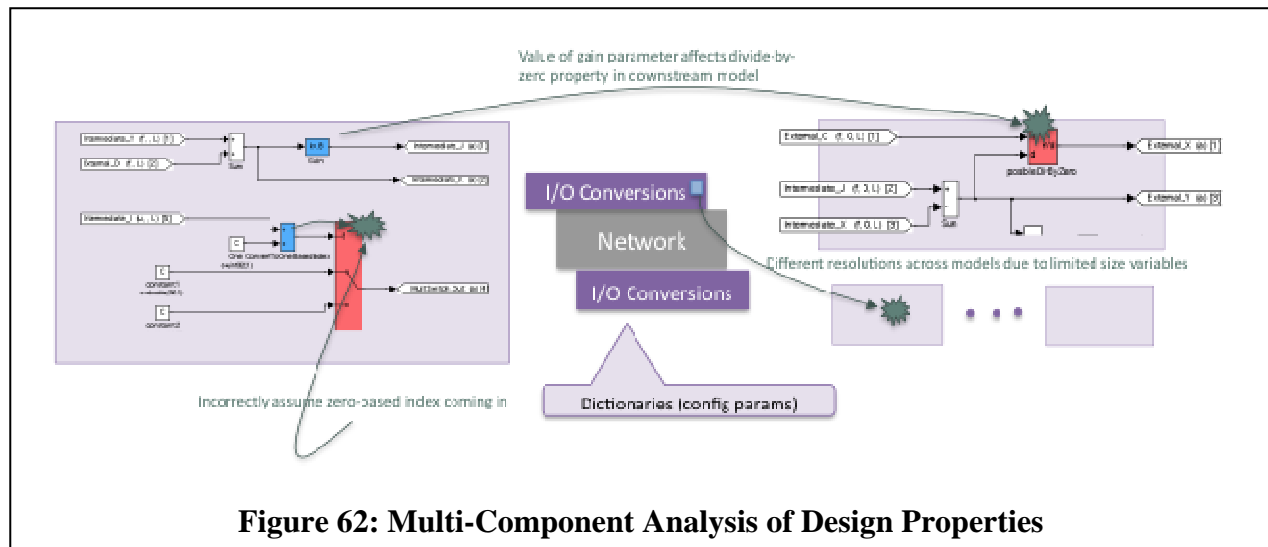
6.2.3.2 Static Model Analysis Background and Approach

For several application domains such as flight controls, engine controls, and environment controls, Honeywell (and other industry members) follow a model-based approach to control software development, using MATLAB Simulink and Stateflow modeling tools. Honeywell’s HiLiTE has been used for static analysis and test generation in the aforementioned domains.

HiLiTE was designed to perform static analysis on individual models. That is, the options are set to process one or more models specified in a HiLiTE command file. It then analyzes those models in sequence without any information about inter-model connections. It propagates supplied ranges for the model inputs (or uses the maximum ranges of the data types) through the blocks to the model outputs. Along the way, it identifies possible division by zero when the range of a divisor includes zero. In addition, it accommodates overflow when, say, the maximum magnitudes of the ranges for values being added or multiplied exceed the data type, or underflow when the ranges span such broad magnitudes that adding one lower value to a larger one will not affect the larger value. Other clear model errors such as a multiport selector that may exceed the number of ports are also reported.

Moving beyond single models, we have extended the HiLiTE interfaces and execution control within the CyPhy environment to verify properties for multiple integrated models. HiLiTE can

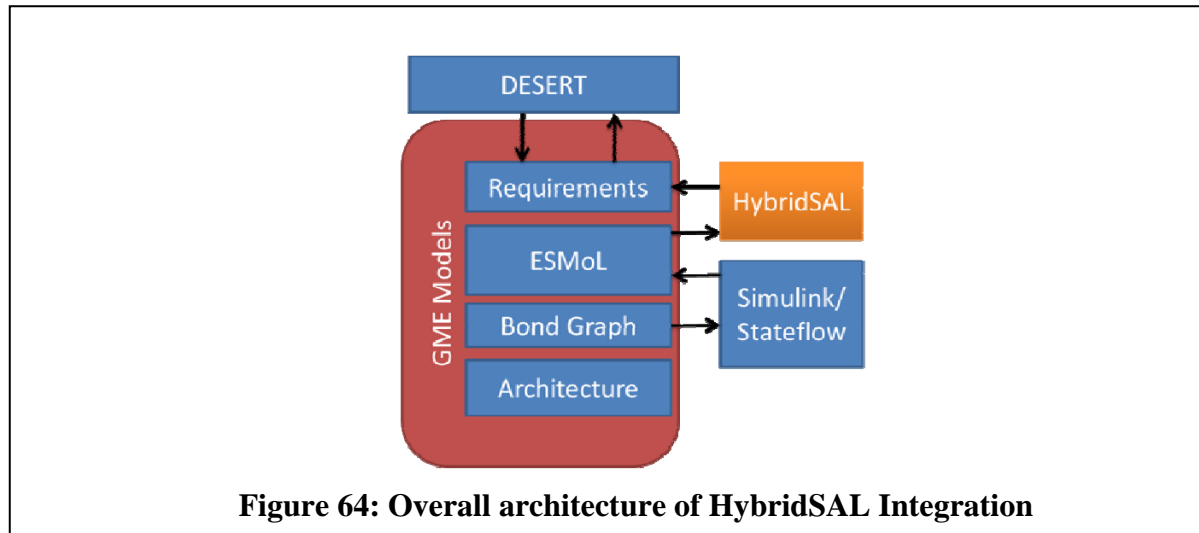
output the ranges it propagates in the same file format used to input model input ranges, so HiLiTE can be run in sequence on multiple models, using the computed output ranges from one model as the input ranges for the next. These computed ranges then inform the divide-by-zero, overflow, underflow, and other block-specific analyses. Figure 62 illustrates examples of how the tuning of parameters in a component or the interactions of multiple assumptions in several components can lead to violations of design properties. The approach is to model the component interconnections and system dictionaries in the CyPhy design flow and provide designers a view of the property violations directly in the context of architecture designs and component models.



6.2.3.3 CyPhy—HiLiTE Integration Approach

CyPhy captures the inter-model relationships of a system – that is, the connections from one model’s outputs to another’s inputs. Given the ranges for the system inputs, CyPhy can then generate a HiLiTE command file referencing the first model (in a topological ordering where model A runs before model B when model A’s outputs are connected to model B’s inputs) and the supplied ranges to analyze the model. HiLiTE propagates the input ranges for the given model to the outputs and returns a new range file that CyPhy can then reference in a new HiLiTE command file for the next model in the system. In the absence of feedback loops, a single topologically ordered run through the system will result in the system output ranges. If feedback loops are present in the system, multiple runs are required until the ranges converge. If a loop diverges, a maximum number of runs can be specified, after which CyPhy-HiLiTE reports the divergence and terminates the run.

Moreover, the transformation from Bond Graphs to Simulink/Stateflow might be problematic, since it allows generic functions that might not have an equivalent expression in the domain of ODEs. This problem is orthogonal to the model representation; it is conceptually present in Simulink and in ESMoL. Figure 64 shows the overall architecture of the integration.



In the next sections, we give a detailed explanation of the Simulink/Stateflow to HybridSAL transformation.

6.3.1.2 Simulink/Stateflow to HybridSAL Transformation

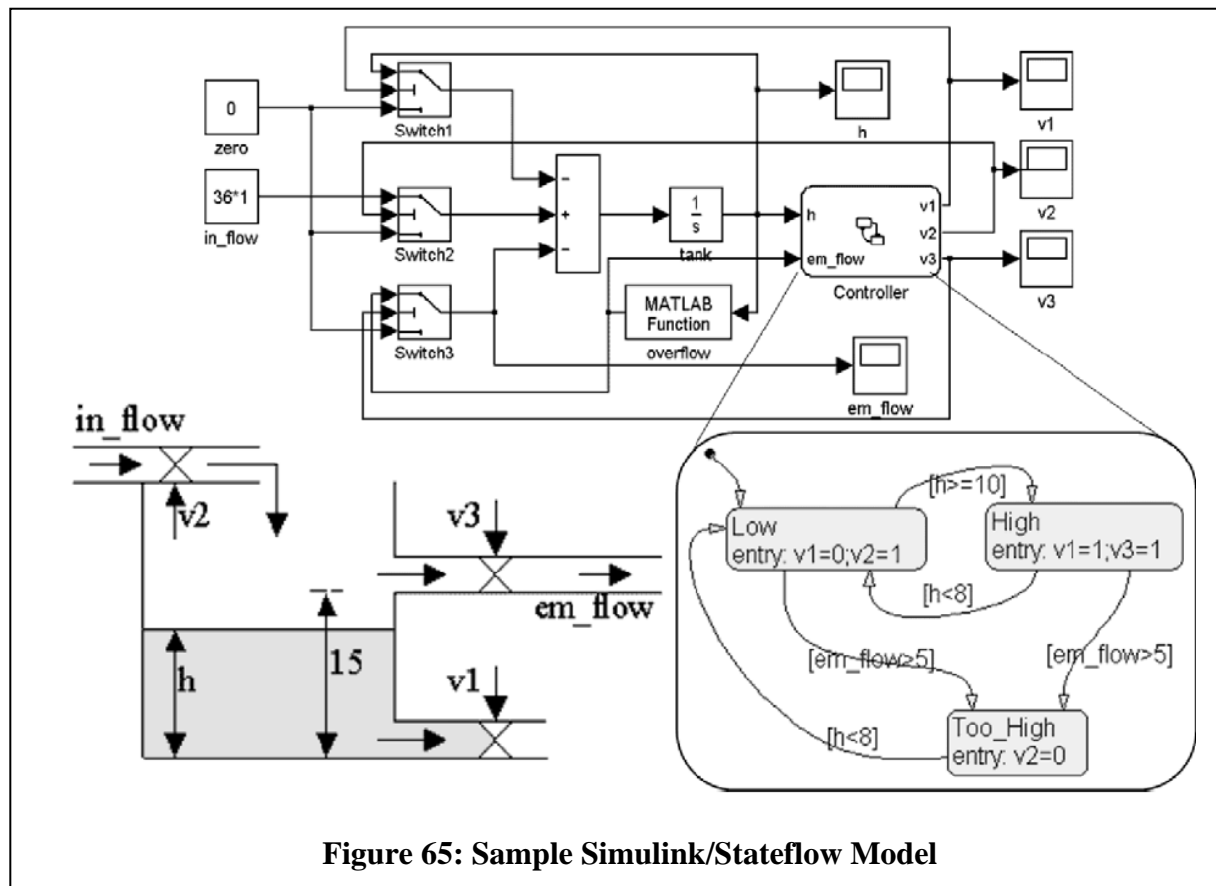
The main challenge of this transformation is that while hybrid automata represent the continuous time part and the discrete behavior separated, these concerns are scattered across a Simulink/Stateflow model. The transformation consists of two main steps:

1. Identifying the states of the hybrid automata and constructing the conditions
2. Converting the continuous time blocks into algebraic differential equations

Below we describe each of these steps. We rely upon research described in [27], and extend the results as explained below.

Identifying the states.

Figure 65 shows a Simulink/StateFlow diagram. Switch elements and a state machine define the discrete behavior. We can observe that the state *Low* does not define a value for *Switch3*. This means that *Switch3* can be either on or off, which needs a separate state to describe the different states for *Switch3* – that is, state *Low* must be split into two states.

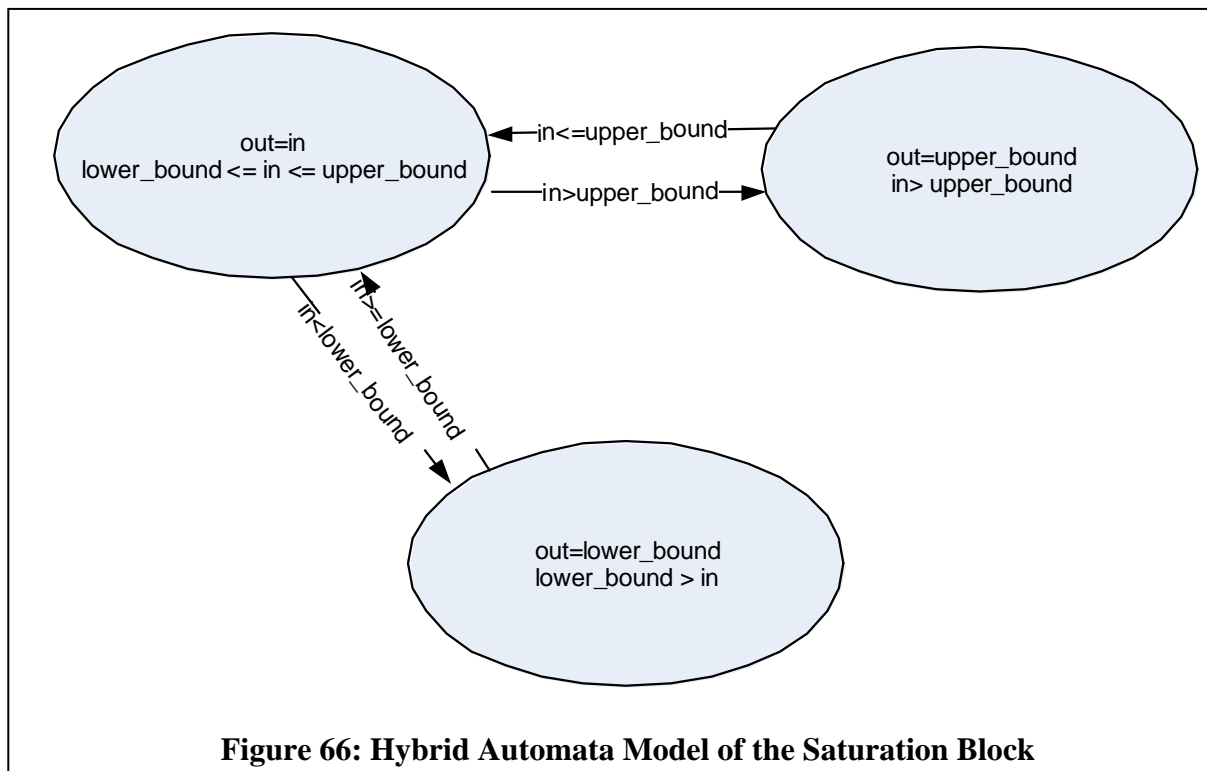


This gives the notion of undefined state, for which new states must be generated. Some of these states might turn out to be dead states, so the resulting automata must be pruned after this state generation.

If we look closely at Figure 58, we can see two additional examples for discrete behavior. The saturation block limits with lower and upper bounds:

1. If the signal is between the lower and upper bounds, the result is the signal.
2. If the signal is below the lower bound, the result is the lower bound.
3. If the signal is above the upper bound, the result is the upper bound.

The hybrid automaton given in Figure 66 models this definition.



The integrator with saturation is very similar:

- When the integral is less than or equal to the lower saturation limit, the output is held at the lower saturation limit.
- When the integral is between the lower saturation limit and the upper saturation limit, the output is the integral.
- When the integral is greater than or equal to the upper saturation limit, the output is held at the upper saturation limit.

Thus, we must model this case as an integrator and a saturation component. There is one more component in our case study that does not have either a continuous or discrete counterpart: the bus creator groups signals together, which does not change the signals but only groups them.

Creating Continuous Time Blocks.

Since the Simulink representation is not in state space format, we must transform the complex frequency domain components to a state space representation. This conversion has a well-known solution. For the elements with discrete behavior, the hybrid automata states identified according to the method described in the previous section determine the state of the switches and saturation components. Using these states, we need to focus on the continuous behavior in a certain state only.

The Transformation of the Case Study.

The result of the transformation is depicted in Figure 67. The discrete behavior is defined by the saturation components.

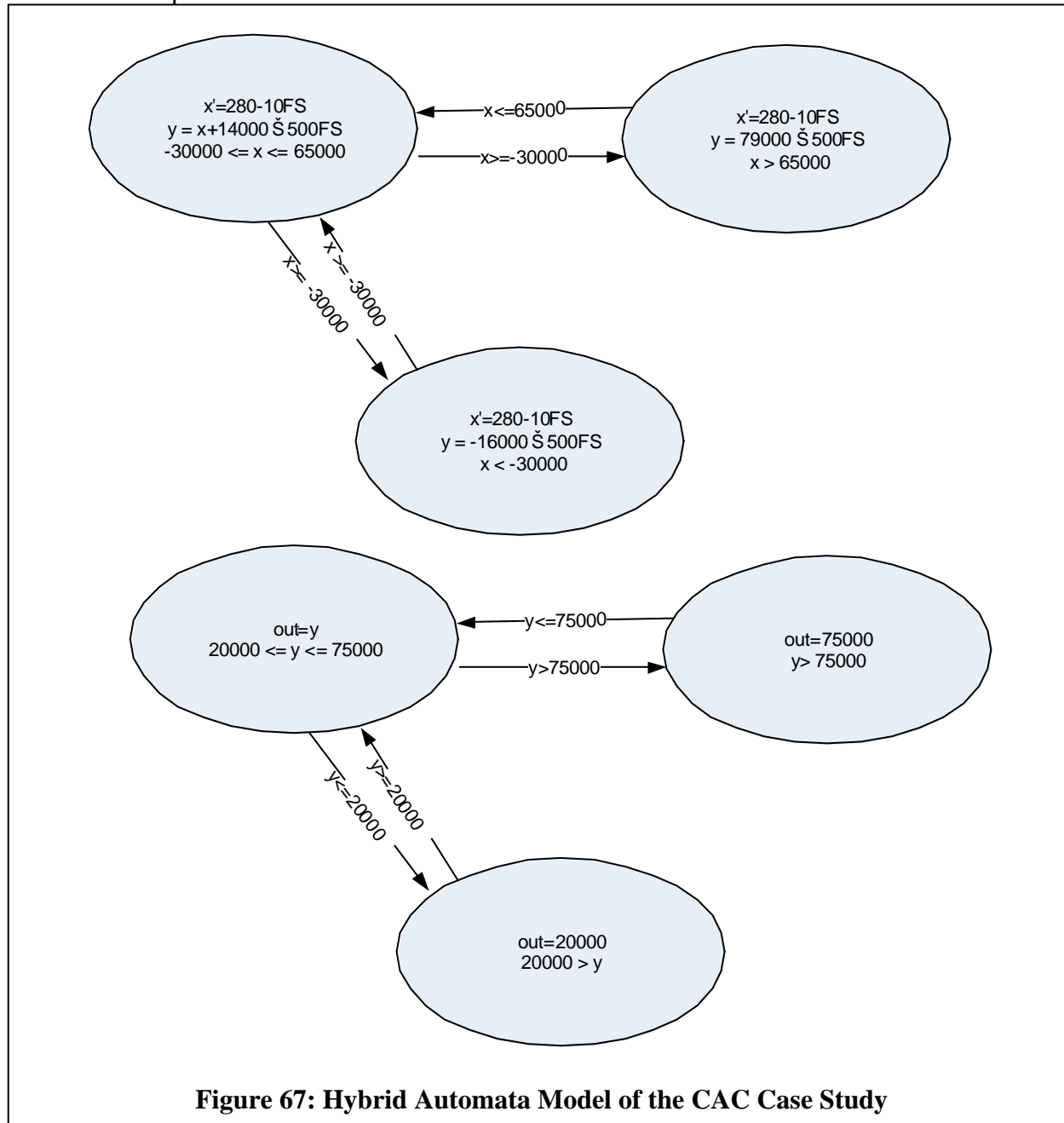


Figure 67: Hybrid Automata Model of the CAC Case Study

We modeled the system with two automata sharing the signal y . The $SPEED_SET$ is labeled out , while the $FLOW_FB$ is labeled FS . Adding an extra state for each integrator state, and setting the input signal to the initial value (in our case, to zero) can model the reset signal.

6.3.2 Scalable, Multi-Component Static Verification Integrated with Design Flow

6.3.2.1 Design Flow Views of Models, Requirements, and Verification Results

HiLiTE's static analyses cover most static code analyzers and provide additional checks that would require a model checker. HiLiTE does so with less user input than a model checker, and it can scale over thousands of analyses across hundreds of components. Indeed, by running the model analyses alone and skipping test generation, HiLiTE has been able to process real-world models in a matter of minutes. This speed makes it practical to run HiLiTE earlier in the development process when bugs can be addressed at a much lower cost.

The combination of CyPhy and HiLiTE supports the capture of inter-model relationships to support cross-component analyses. This will allow HiLiTE to identify problems at blocks that are triggered by blocks in upstream models. CyPhy can reference the MATLAB Simulink models, as shown in Figure 68.

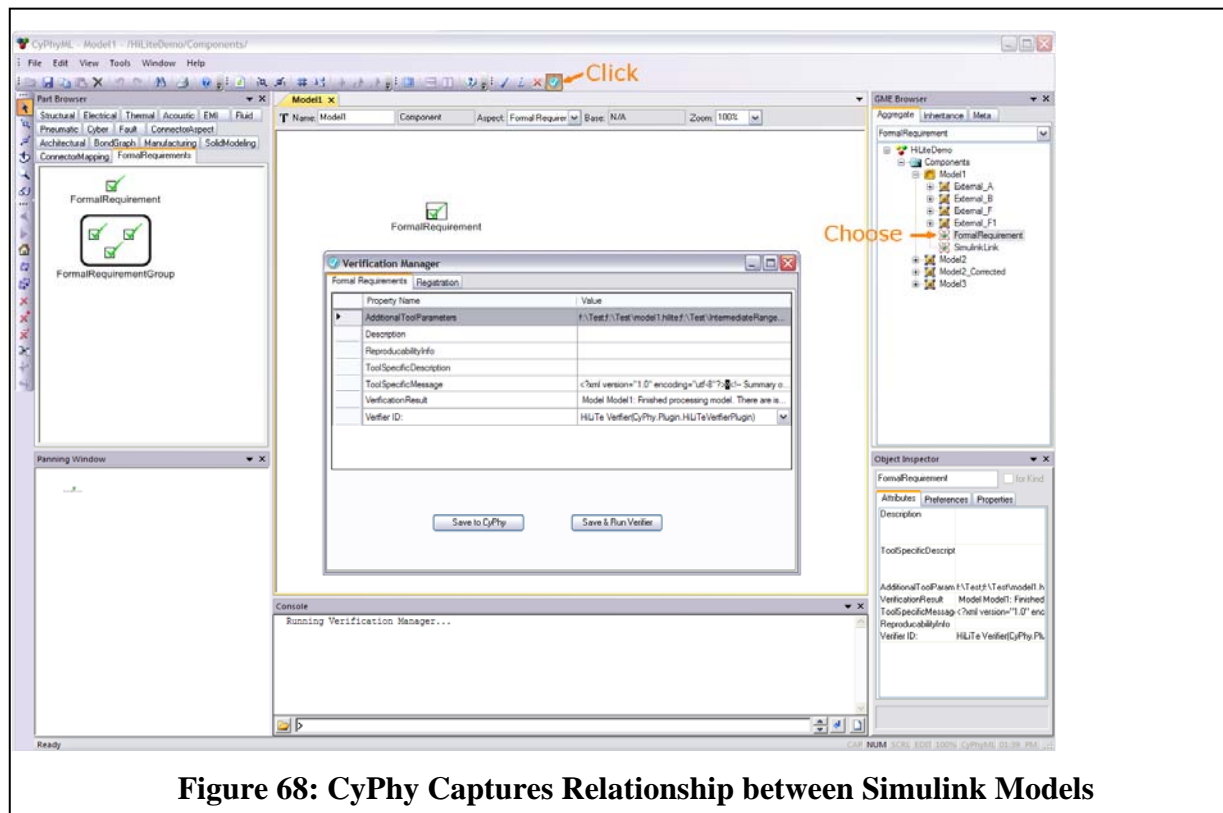


Figure 68: CyPhy Captures Relationship between Simulink Models

CyPhy can then invoke HiLiTE on the Simulink models, as shown in Figure 69.

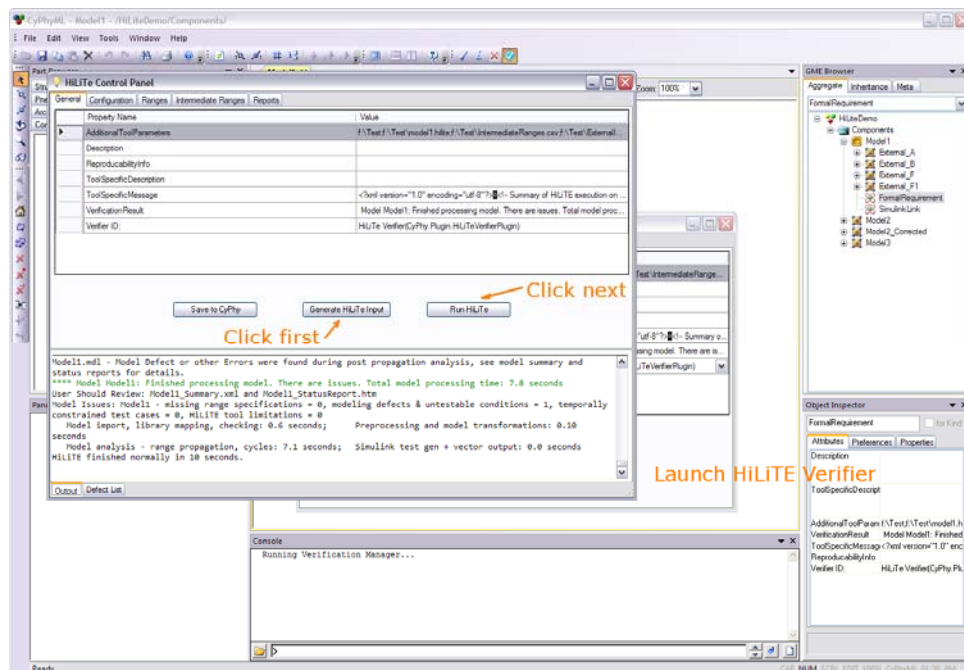


Figure 69: CyPhy can Generate HiLiTE Inputs to Run Offline, or Directly Invoke HiLiTE

Finally, the outputs of the HiLiTE propagation and analysis are imported back into CyPhy for subsequent runs, as shown in Figure 70.

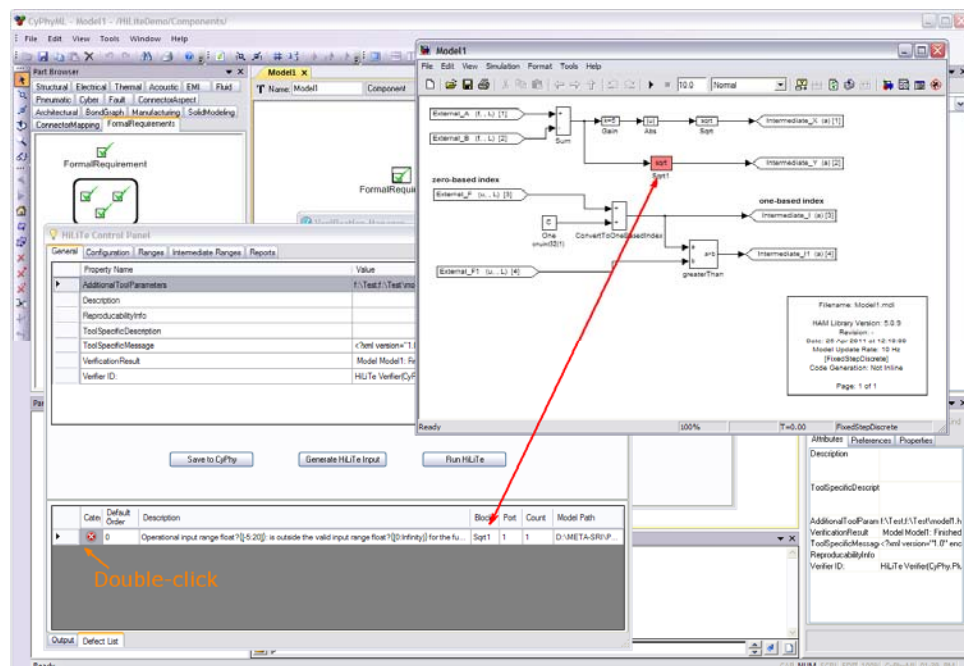


Figure 70: Verification Results are fed back into the design tool

6.3.2.2 Examples of Property Violations in Integrated Components

Example 1: Divide-by-Zero Overflow due to Control Parameter Tuning

The diagram shown in Figure 71 is a structure excerpted from real avionics systems that includes mathematical computations, one of which (shown in color) can produce an overflow if the value on the denominator (d) can be zero. This can cause an arithmetic exception and adversely impact safety. In the particular model, the signals driving the value at 'd' are Intermediate_J and Intermediate_X that are produced by other models from different sources.

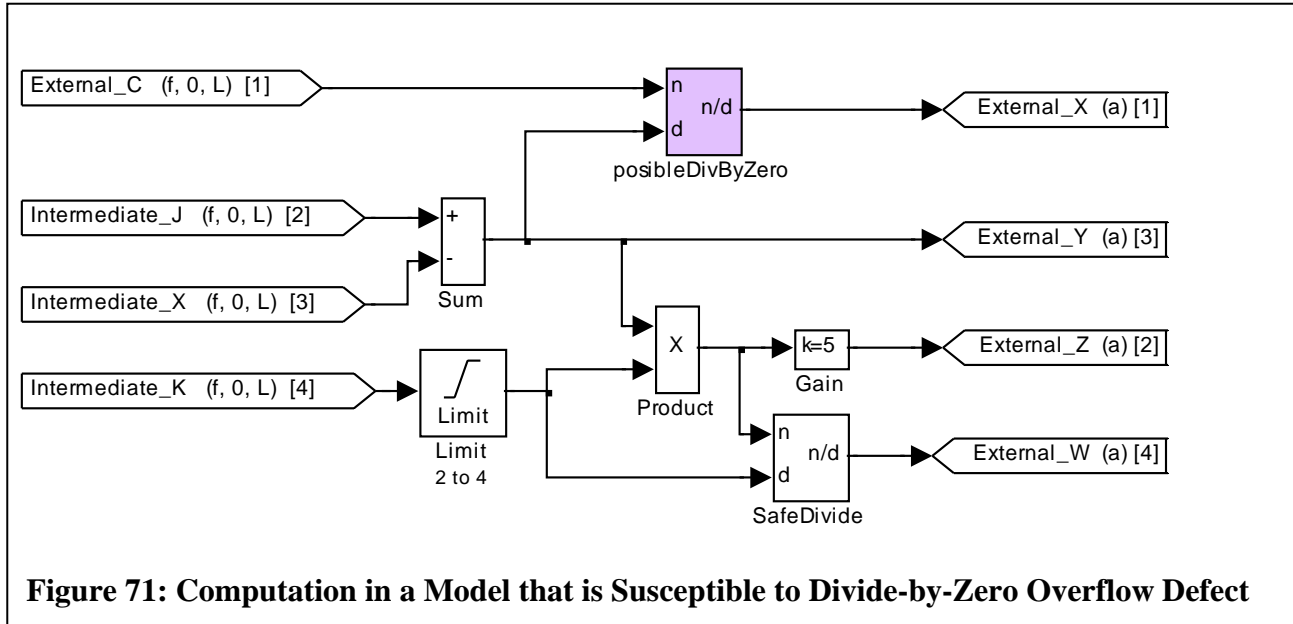
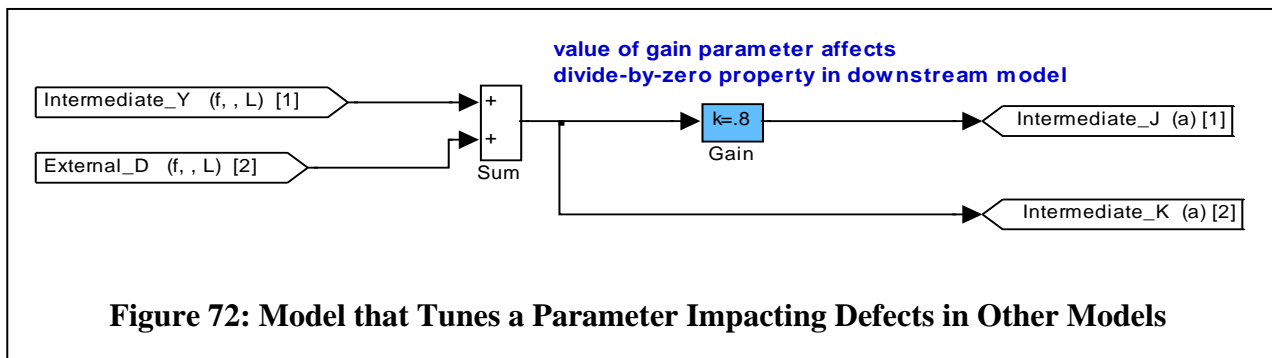


Figure 72 shows an excerpt from a model that tunes a gain parameter controlling the signal Intermediate_J that is used by the previous model as input. The verification shows that for the value of the gain parameter shown in this model, the divide-by-zero defect exists in the model shown in Figure 73. When the value of the gain parameter is changed from 0.8 to 1.1, then the range bound on signal Intermediate_J is constrained such that the static analysis can prove the absence of the divide-by-zero defect in the model of Figure 73.



Example 2: Unreachable Models in Flight Modes Testing

The FlightModesTest model in Figure 73 is a structure excerpted from real avionics systems as part of built-in tests that must execute at certain times to test the control response of actuators in various flight models of operation. In an independent environment, without any integration context, there is no problem with the model. All states and transitions are reachable and all conditions can be tested.

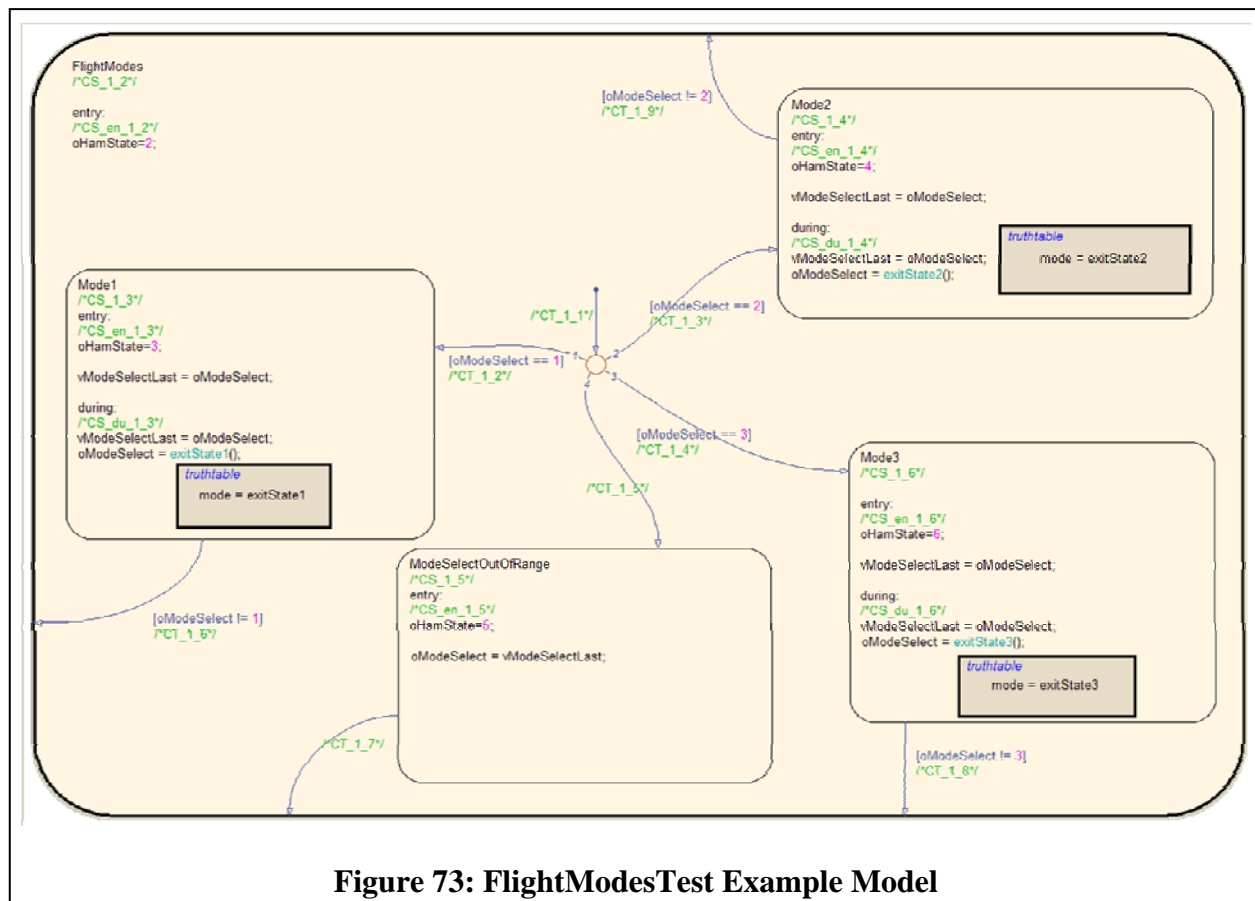
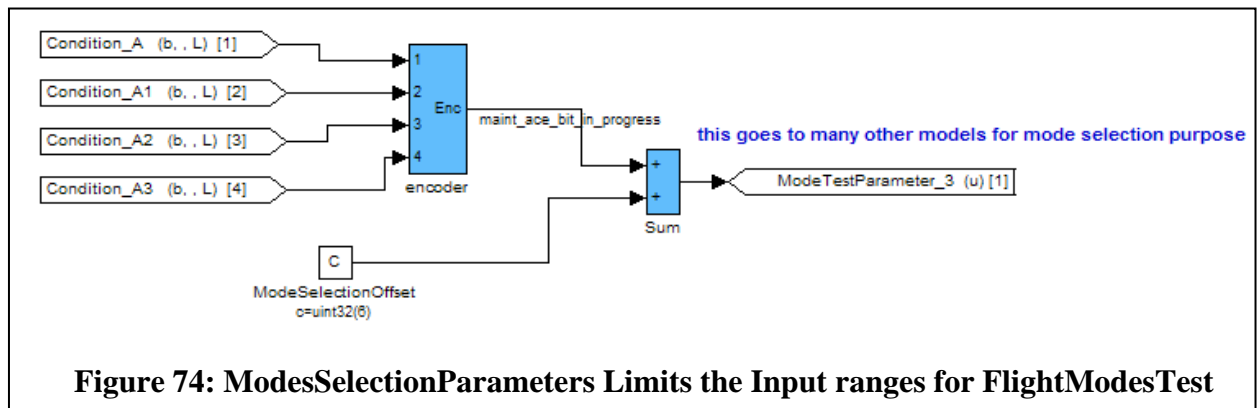


Figure 73: FlightModesTest Example Model

However, the upstream model ModeSelectionParameters limits the inputs to FlightModesTest. In particular, the ModeTestParameter_3 output of ModeSelectionParameters is connected to the iSignal03 input of FlightModesTest. The encoder block will return the value of the highest port with an input of 1 – that is, the maint_ace_bit_in_progress output will be an integral value between zero and four. HiLiTE will propagate this information through the sum block to determine that the range of ModeTestParameter_3 is six to ten.



This is a problem for FlightModesTest. As shown in the exitState2 truth table of Figure 75, the value of iSignal03 must be greater than ten to set mode to three. Since the range is constrained to be less than or equal to ten, oModeSelect will not be set to 3 in Mode2. Likewise, in exitState1 and when oModeSelect is initialized, iSignal03 must be greater than ten to set oModeSelect to 3 to enter Mode3.

Condition Table					
	Description	Condition	D1	D2	
1		iSignal03 > 10	T	-	
	Actions: Specify a row from the Action Table		1	2	
Action Table					
#	Description	Action			
1	ANCHOR: a1;	mode = 3;			
2	ANCHOR: a2;	mode = vModeSelectLast;			

Figure 75: Value of Mode will be 3 only if the Value of iSignal03 is Greater than 2

When HiLiTE runs on ModeSelectionParameters, it outputs an intermediate ranges file that reflects the computed range for ModeTestParameter_3. When this ranges file is supplied to HiLiTE for a run on FlightModesTest, HiLiTE reports that Mode3 is an unreachable state.

6.4 Conclusions

The META approach to reusability of library components can only be realized if detailed design properties of integrated components are automatically verified as parameters are selected and fine-tuned for specific vehicle configurations. Providing automated verification tools that give feedback to the designer by tracing causes of violations across components is key to success. The integrated CyPhy—HiLiTE tool provides such capability.

Checking safety and stability properties of designs early in the process is important so as to avoid proceeding with designs that do not satisfy these crucial requirements. The integrated CyPhy—HybridSAL capability allows checking such requirements during the design phase.

Industry observation has shown that 40-50% of all defects are injected in model design phases. Other studies have shown that problems found at code testing can cause much more effort than if caught in early design. Thus, integrating verification tools into the design phase is a solution to reduce time and cost of software development because errors can be detected early.

A feature which is critical in decreasing time schedules is the capability to automate the management of labor intensive tasks, such as the static analysis of large scale embedded control software, and seamlessly integrating the verification results with the architecture models and the design space exploration process.

7. REFERENCES

1. Sriram Sankaranarayanan and Ashish Tiwari, "Relational abstractions for continuous and hybrid systems", in *CAV 2011*: 686-702.
2. Ashish Tiwari, "Approximate reachability for linear systems", in *Proceedings of Hybrid Systems: Computation and Control*, HSCC 2003.
3. <http://www.csl.sri.com/users/tiwari/existsforall/>
4. Thomas Sturm and Ashish Tiwari, "Verification and synthesis using real quantifier elimination", in *ISSAC 2011*.
5. Ashish Tiwari, "Compositionally analyzing a PI controller family", in *CDC 2011*. To appear.
6. Matt Richardson, Pedro Domingos. Markov logic networks. *Machine Learning* 62(1-2), 107–136 (2006).
7. Shalini Ghosh, Natarajan Shankar, Sam Owre, "Machine Reading Using Markov Logic Networks for Collective Probabilistic Inference", Appearing in the Proceedings of the European Conference on Machine Learning (ECML) Workshop on Collective Learning and Inference from structured data (CoLISD), 2011.
8. Hoifung Poon, Pedro Domingos. Joint inference in information extraction. In: *AAAI* (2007)
9. Alfons Geser and Paul Miner. A New On-line Diagnosis Protocol for the SPIDER Family of Byzantine Fault Tolerant Architectures. NASA TM-212432, December 2004.
10. Marc Chérèque, David Powell, Philippe Reynier, Jean-luc Richier, Jacques Voiron, "Active Replication in Delta-4", Symposium on Fault-Tolerant Computing - FTCS, pp. 28-37, 1992
11. Y.C. (Bob) Yeh, "Triple-Triple Redundant 777 Primary Flight Computer", 1996 IEEE Aerospace Applications Conference, pp. 293-307.
12. Roger M. Kieckhafer, Chris J. Walter, Alan M. Finn, Philip M. Thambidurai, *The MAFT Architecture for Distributed Fault*, IEEE Transactions on Computers - TC, vol. 37, no. 4, pp. 398-405.

13. Robert C. Hammett, Philip S. Babcock, "Achieving 10^{-9} Dependability with Drive-by-Wire Systems", The Charles Stark Draper Lab, SAE 2003 World Congress & Exhibition, March 2003, SP-1783
14. Hermann Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications, 2nd Edition, 2011, XVIII, 376 p.
15. Stefan Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, The Springer International Series in Engineering and Computer Science, 1996
16. Maria Sorea and Wilfried Steiner, "Classification and Analysis of Failure Modes for Time-Triggered Systems", 7th IFAC International Conference on Fieldbuses and Networks in Industrial and Embedded Systems (2007), Fieldbuses and Networks in Industrial and Embedded Systems, Volume# 7, Part# 1.
17. 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, March 2001. doi: 10.1016/S0951-8320(00)00076-4.
18. M.J. O'Donnell, *Equational Logic as a Programming Language*, Massachusetts Institute of Technology, Cambridge, MA, USA (1985).
19. J. Meseguer, "Conditional rewriting logic as a unified model of concurrency", *Theoretical Computer Science* 96 (1992) 73–155.
20. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott, eds.: "All about Maude - A high-performance logical framework: How to specify, program and verify systems in rewriting logic", Vol. 4350 of *Lecture Notes in Computer Science*, Springer (2007).
21. M. Clavel and J. Meseguer, "Reflection and strategies in rewriting logic", in Rewriting Logic Workshop 96, Number 4 in *Electronic Notes in Theoretical Computer Science*, Elsevier (1996). <http://www.elsevier.nl/locate/entcs/volume4.html>.
22. M. Wirsing, "Algebraic specification", in van Leeuwen, J., ed.: *Handbook of Theoretical Computer Science*, Vol. B., North-Holland (1990) 675–788
23. http://en.wikipedia.org/wiki/Probability_space
24. N.J. Nilsson, 1986, "Probabilistic logic", *Artificial Intelligence* 28(1): 71-87.
25. <http://www.youtube.com/watch?v=PnSLzCPTdts>, ASIIST Schedulability Analysis Demo
26. <http://www.youtube.com/watch?v=NrXtwK8aowE&feature=related>, ASIIST Bus Delay Analysis Demo

27. Aditya Agrawal, Gyula Simon, and Gabor Karsai, “Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations”, *Electronic Notes in Theoretical Computer Science*, Vol. 109, 14 December 2004, pages 43-56.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

APU	auxiliary power unit
ASIIST	application specific i/o integration support
BAG	bandwidth allocation gap
BBW	brake by wire
BE	best effort
BIU	bus interface unit
BUS	physical media/link representing PCI bus
CAC	cabin air compressor
CAN	controller area network
COTS	common off-the-shelf
CRC	cyclic redundancy check
CPS	cyber-physical system
CyPhy	Cyber-Physical (language)
DA	destination address
DE	discrete event
DESERT	the design space exploration tool
DoD	Department of Defense
ECS	environmental control system
EL	equational logic
ES	end system
EsMOL	embedded system modeling language
FCS	frame check sequence
FCR	fault containment region
GUI	graphical user interface

HI	high integrity
HiLiTE	Honeywell lifecycle tools & environment
IFG	inter frame gap
KB	knowledge base
kW	kilowatt
FF	feed forward
FSM	finite state machine
ID	identifier
IMA	integrated modular avionics
LRU	line-replaceable unit
MCMC	Markov chain Monte Carlo
MC-SAT	Monte Carlo Satisfiability
MLN	Markov-logic network
NASA	National Aeronautics and Space Administration
ODE	ordinary differential equations
OMH	one more hop
OMH-FTP	one more hop file transfer protocol
OS	operating system
PCE	probabilistic consistency engine
PE	processing elements
PI	proportional integral
PHY	physical link
PVS	prototype verification system
PROMISE	probabilistic, compositional, multi-dimension model-based verification
PTMS	power and thermal management system
RC	rate constrained

RL	rewriting logic
RMU	redundancy management unit
ROBUS	reliable optical bus
RST	reset
Rx	receive
SA	source address
SAL	symbolic analysis laboratory
SAT	satisfiability
SC	supervisory control
SFD	static frame delimiter
SI	standard integrity
SMT	satisfiability modulo theory
SN	sequence number
SOS	sums of squares
SPEED_FF	speed feed forward
SPIDER	scalable processor-independent design for extended reliability
TDMA	time-division multiple access
TMR	triple modular redundancy
TT	time triggered
TTE	TT Ethernet
TTGbE	time-triggered gigabit Ethernet
Tx	transmit
V&V	verification and validation
VL	virtual link
VLID	virtual link identifier

APPENDIX 1

The architectures A2, A3, A4, A5 the corresponding Markov-Logic Networks (MLNs) (two-cac1-complete.mln, three-cac1-complete.mln, voting-three-cac-complete.mln, and three-cac-hl-complete.mln) and the corresponding trace outputs on running MCSAT on each MLN (two-cac1-complete.mln.output, three-cac1-complete.mln.output, voting-three-cac-complete.mln.output, and three-cac-hl-complete.mln.output) are provided below.

two-cac1-complete.mln

```
## Copyright: SRI International, 2010.
##
## This MLN simulates failures in a dual-CAC system
## by using a coarse system model. This version tries to
## test whether you can regenerate the prior on failSystem
## from the System rules

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;

# System
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 2 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;

### Rules

# System
add (failCac(c1) and failCac(c2)) iff failSystem(s);

# Component
add failCac(c1) => highLoadCac(c2);
add failCac(c2) => highLoadCac(c1);

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);
```

```
# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1;
```

```
### Priors
add [cac_id] failMotor(cac_id) -4;
add [cac_id] failMotorController(cac_id) -4;
add [cac_id] failCompressor(cac_id) -4;
add [cac_id] failCanBus(cac_id) -4;
```

```
### Query
mcsat_params 10000000, 0.01, 20.0, 0.01, 100;
mcsat; dumptables atom;
```

two-cac1-complete.mln.output

```
Loading two-cac1-complete.mln
Input from file two-cac1-complete.mln
Setting MCSAT parameters:
  max_samples was 100, now 10000000
  sa_probability was 0.500000, now 0.010000
  samp_temperature was 5.000000, now 20.000000
  rvar_probability was 0.050000, now 0.010000
  max_flips was 100, now 100
```

```
Calling MCSAT with parameters (set using mcsat_params):
  max_samples = 10000000
  sa_probability = 0.010000
  samp_temperature = 20.000000
  rvar_probability = 0.010000
  max_flips = 100
  max_extra_flips = 5
  timeout = 0
```

i	probability	atom
0	1.0144e-03	failSystem(s)
1	3.1614e-02	failCac(c1)
2	4.9182e-01	highLoadCac(c1)
3	8.1695e-03	failMotor(c1)
4	8.1082e-03	failMotorController(c1)
5	8.1208e-03	failCompressor(c1)
6	8.1054e-03	failCanBus(c1)
7	3.1046e-02	failCac(c2)
8	4.9219e-01	highLoadCac(c2)
9	8.1084e-03	failMotor(c2)
10	7.9976e-03	failMotorController(c2)
11	7.9406e-03	failCompressor(c2)
12	7.8608e-03	failCanBus(c2)

three-cac1-complete.mln

```
## Copyright: SRI International, 2010.
##
## This MLN simulates failures in a triple-CAC system
## by using a coarse system model.

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;

# System
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 3 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;
const c3: cac_id;

### Rules

# System
add (failCac(c1) and failCac(c2) and failCac(c3)) iff failSystem(s);

# Component
add failCac(c1) => highLoadCac(c2);
add failCac(c1) => highLoadCac(c3);
add failCac(c2) => highLoadCac(c1);
add failCac(c2) => highLoadCac(c3);
add failCac(c3) => highLoadCac(c1);
add failCac(c3) => highLoadCac(c2);

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);

# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1;

### Priors
add [cac_id] failMotor(cac_id) -4;
```

```

add [cac_id] failMotorController(cac_id) -4;
add [cac_id] failCompressor(cac_id) -4;
add [cac_id] failCanBus(cac_id) -4;

```

```

### Query
mcsat_params 10000000, 0.01, 20.0, 0.01, 100;
mcsat;
dumptables atom;

```

three-cac1-complete.mln.output

```

Loading three-cac1-complete.mln
Input from file three-cac1-complete.mln
Setting MCSAT parameters:
  max_samples was 100, now 10000000
  sa_probability was 0.500000, now 0.010000
  samp_temperature was 5.000000, now 20.000000
  rvar_probability was 0.050000, now 0.010000
  max_flips was 100, now 100

```

```

Calling MCSAT with parameters (set using mcsat_params):
  max_samples = 10000000
  sa_probability = 0.010000
  samp_temperature = 20.000000
  rvar_probability = 0.010000
  max_flips = 100
  max_extra_flips = 5
  timeout = 0

```

i	probability	atom
0	5.9600e-05	failSystem(s)
1	1.7452e-02	failCac(c1)
2	4.9239e-01	highLoadCac(c1)
3	4.5388e-03	failMotor(c1)
4	4.4547e-03	failMotorController(c1)
5	4.4539e-03	failCompressor(c1)
6	4.4972e-03	failCanBus(c1)
7	1.6874e-02	failCac(c2)
8	4.9249e-01	highLoadCac(c2)
9	4.4293e-03	failMotor(c2)
10	4.2980e-03	failMotorController(c2)
11	4.3343e-03	failCompressor(c2)
12	4.2942e-03	failCanBus(c2)
13	1.7171e-02	failCac(c3)
14	4.9246e-01	highLoadCac(c3)
15	4.4812e-03	failMotor(c3)
16	4.4125e-03	failMotorController(c3)
17	4.3850e-03	failCompressor(c3)
18	4.3950e-03	failCanBus(c3)

voting-three-cac-complete.mln

```
## Copyright: SRI International, 2010.
##
## This MLN simulates failures in a triple-CAC system that votes the
## cac outputs, by using a coarse system model.

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;

# System
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 3 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;
const c3: cac_id;

### Rules

# System
add [c,d,e,s] ((failCac(c) and failCac(d)) or (failCac(c) and failCac(d) and
  failCac(e))) and ((c ~= d) and (d ~= e) and (c ~= e)) iff failSystem(s);

# Component
add [c, d] failCac(c) and ~failCac(d) and (c ~= d) => highLoadCac(d);

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);

# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1;

### Priors
add [cac_id] failMotor(cac_id) -4;
add [cac_id] failMotorController(cac_id) -4;
add [cac_id] failCompressor(cac_id) -4;
add [cac_id] failCanBus(cac_id) -4;
```

```

### Query
mcsat_params 10000000, 0.01, 20.0, 0.01, 100;
mcsat;
dumptables atom;

```

voting-three-cac-complete.mln.output

```

Loading voting-three-cac-complete.mln
Input from file voting-three-cac-complete.mln
Setting MCSAT parameters:

```

```

    max_samples was 100, now 10000000
    sa_probability was 0.500000, now 0.010000
    samp_temperature was 5.000000, now 20.000000
    rvar_probability was 0.050000, now 0.010000
    max_flips was 100, now 100

```

```

Calling MCSAT with parameters (set using mcsat_params):

```

```

    max_samples = 10000000
    sa_probability = 0.010000
    samp_temperature = 20.000000
    rvar_probability = 0.010000
    max_flips = 100
    max_extra_flips = 5
    timeout = 0

```

i	probability	atom
0	2.7610e-04	failSystem(s)
1	1.7014e-02	failCac(c1)
2	4.9184e-01	highLoadCac(c1)
3	4.2967e-03	failMotor(c1)
4	4.4401e-03	failMotorController(c1)
5	4.4544e-03	failCompressor(c1)
6	4.3056e-03	failCanBus(c1)
7	1.6410e-02	failCac(c2)
8	4.9181e-01	highLoadCac(c2)
9	4.2073e-03	failMotor(c2)
10	4.2543e-03	failMotorController(c2)
11	4.2419e-03	failCompressor(c2)
12	4.1764e-03	failCanBus(c2)
13	1.6163e-02	failCac(c3)
14	4.9228e-01	highLoadCac(c3)
15	4.1988e-03	failMotor(c3)
16	4.1342e-03	failMotorController(c3)
17	4.2161e-03	failCompressor(c3)
18	4.0727e-03	failCanBus(c3)
19	0.0000e+00	failSystem(c1)
20	0.0000e+00	failSystem(c2)
21	0.0000e+00	failSystem(c3)

three-cac-h1-complete.mln

```
## Copyright: SRI International, 2010.
##
## This MLN simulates failures in a triple-CAC system that votes the
## cac outputs, by using a coarse system model.

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;

# System
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 3 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;
const c3: cac_id;

### Rules

# System
add [c,d,e,s] (failCac(c) and failCac(d) and failCac(e)) and ((c ~= d) and (d
  ~= e) and (c ~= e)) iff failSystem(s);

# Component
add [c, d] failCac(c) and ~failCac(d) and (c ~= d) => highLoadCac(d) -1.1;
# prob = 0.25
add [c, d, e] (failCac(c) and failCac(d) and ~failCac(e)) and ((c ~= d) and
  (d ~= e) and (c ~= e)) => highLoadCac(e) 1.1; # p = 0.75

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);

# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1; # prob = 0.525

### Priors
add [cac_id] failMotor(cac_id) -4;
```



```
add [cac_id] failMotorController(cac_id) -4;
add [cac_id] failCompressor(cac_id) -4;
add [cac_id] failCanBus(cac_id) -4;
```

```
### Query
mcsat_params 1000000, 0.01, 20.0, 0.01, 100;
mcsat;
dumptables atom;
```

three-cac-hl-complete.mln.output

```
Loading three-cac-hl-complete.mln
Input from file three-cac-hl-complete.mln
Setting MCSAT parameters:
  max_samples was 100, now 1000000
  sa_probability was 0.500000, now 0.010000
  samp_temperature was 5.000000, now 20.000000
  rvar_probability was 0.050000, now 0.010000
  max_flips was 100, now 100
```

Calling MCSAT with parameters (set using mcsat_params):

```
max_samples = 1000000
sa_probability = 0.010000
samp_temperature = 20.000000
rvar_probability = 0.010000
max_flips = 100
max_extra_flips = 5
timeout = 0
```

i	probability	atom
0	3.4100e-04	failSystem(s)
1	1.6839e-01	failCac(c1)
2	4.0064e-01	highLoadCac(c1)
3	4.2501e-02	failMotor(c1)
4	4.3303e-02	failMotorController(c1)
5	4.3514e-02	failCompressor(c1)
6	4.3637e-02	failCanBus(c1)
7	1.7034e-01	failCac(c2)
8	4.0236e-01	highLoadCac(c2)
9	4.3206e-02	failMotor(c2)
10	4.4964e-02	failMotorController(c2)
11	4.2501e-02	failCompressor(c2)
12	4.4192e-02	failCanBus(c2)
13	1.6813e-01	failCac(c3)
14	4.0173e-01	highLoadCac(c3)
15	4.3781e-02	failMotor(c3)
16	4.2395e-02	failMotorController(c3)
17	4.2719e-02	failCompressor(c3)
18	4.3864e-02	failCanBus(c3)
19	0.0000e+00	failSystem(c1)

20	0.0000e+00	failSystem(c2)
21	0.0000e+00	failSystem(c3)

three-cac-hl-complete.output.1fail

Loading three-cac-hl-complete.mln
Input from file three-cac-hl-complete.mln
Setting MCSAT parameters:
max_samples was 100, now 1000000
sa_probability was 0.500000, now 0.010000
samp_temperature was 5.000000, now 20.000000
rvar_probability was 0.050000, now 0.010000
max_flips was 100, now 100

Calling MCSAT with parameters (set using mcsat_params):
max_samples = 1000000
sa_probability = 0.010000
samp_temperature = 20.000000
rvar_probability = 0.010000
max_flips = 100
max_extra_flips = 5
timeout = 0

i	probability	atom
0	1.5070e-03	failSystem(s)
1	1.0000e+00	failCac(c1)
2	5.0075e-01	highLoadCac(c1)
3	1.0000e+00	failMotor(c1)
4	1.7684e-02	failMotorController(c1)
5	1.8135e-02	failCompressor(c1)
6	1.8351e-02	failCanBus(c1)
7	3.3665e-02	failCac(c2)
8	2.3985e-01	highLoadCac(c2)
9	8.6550e-03	failMotor(c2)
10	8.6920e-03	failMotorController(c2)
11	8.7630e-03	failCompressor(c2)
12	8.4750e-03	failCanBus(c2)
13	3.3986e-02	failCac(c3)
14	2.4090e-01	highLoadCac(c3)
15	8.5650e-03	failMotor(c3)
16	8.3860e-03	failMotorController(c3)
17	9.2010e-03	failCompressor(c3)
18	8.7350e-03	failCanBus(c3)
19	0.0000e+00	failSystem(c1)
20	0.0000e+00	failSystem(c2)
21	0.0000e+00	failSystem(c3)

three-cac-hl-complete.output.2fail

```

Loading three-cac-hl-complete.mln
Input from file three-cac-hl-complete.mln
Setting MCSAT parameters:
  max_samples was 100, now 1000000
  sa_probability was 0.500000, now 0.010000
  samp_temperature was 5.000000, now 20.000000
  rvar_probability was 0.050000, now 0.010000
  max_flips was 100, now 100

```

```

Calling MCSAT with parameters (set using mcsat_params):
  max_samples = 1000000
  sa_probability = 0.010000
  samp_temperature = 20.000000
  rvar_probability = 0.010000
  max_flips = 100
  max_extra_flips = 5
  timeout = 0

```

i	probability	atom
0	3.3156e-02	failSystem(s)
1	1.0000e+00	failCac(c1)
2	5.0034e-01	highLoadCac(c1)
3	1.0000e+00	failMotor(c1)
4	1.7894e-02	failMotorController(c1)
5	1.7725e-02	failCompressor(c1)
6	1.7774e-02	failCanBus(c1)
7	1.0000e+00	failCac(c2)
8	5.0000e-01	highLoadCac(c2)
9	1.0000e+00	failMotor(c2)
10	1.7968e-02	failMotorController(c2)
11	1.8006e-02	failCompressor(c2)
12	1.8393e-02	failCanBus(c2)
13	3.3156e-02	failCac(c3)
14	2.3946e-01	highLoadCac(c3)
15	8.8460e-03	failMotor(c3)
16	8.3050e-03	failMotorController(c3)
17	8.3490e-03	failCompressor(c3)
18	8.4980e-03	failCanBus(c3)
19	0.0000e+00	failSystem(c1)
20	0.0000e+00	failSystem(c2)
21	0.0000e+00	failSystem(c3)

three-cac-hl-complete.output.3fail

```

Loading three-cac-hl-complete.mln
Input from file three-cac-hl-complete.mln
Setting MCSAT parameters:
  max_samples was 100, now 1000000
  sa_probability was 0.500000, now 0.010000

```

```
samp_temperature was 5.000000, now 20.000000
rvar_probability was 0.050000, now 0.010000
max_flips was 100, now 100
```

Calling MCSAT with parameters (set using mcsat_params):

```
max_samples = 1000000
sa_probability = 0.010000
samp_temperature = 20.000000
rvar_probability = 0.010000
max_flips = 100
max_extra_flips = 5
timeout = 0
```

i	probability	atom
0	1.0000e+00	failSystem(s)
1	1.0000e+00	failCac(c1)
2	4.9957e-01	highLoadCac(c1)
3	1.0000e+00	failMotor(c1)
4	1.8033e-02	failMotorController(c1)
5	1.8015e-02	failCompressor(c1)
6	1.8095e-02	failCanBus(c1)
7	1.0000e+00	failCac(c2)
8	5.0133e-01	highLoadCac(c2)
9	1.0000e+00	failMotor(c2)
10	1.7969e-02	failMotorController(c2)
11	1.7875e-02	failCompressor(c2)
12	1.7890e-02	failCanBus(c2)
13	1.0000e+00	failCac(c3)
14	4.9966e-01	highLoadCac(c3)
15	1.0000e+00	failMotor(c3)
16	1.8090e-02	failMotorController(c3)
17	1.7762e-02	failCompressor(c3)
18	1.7829e-02	failCanBus(c3)
19	0.0000e+00	failSystem(c1)
20	0.0000e+00	failSystem(c2)
21	0.0000e+00	failSystem(c3)

cac-model1-demo.pcein

```
## Copyright: SRI International, 2010.
##
## This MLN simulates failures in a dual-CAC system
## by using a coarse system model. This version tries to
## test whether you can regenerate the prior on failSystem
## from the System rules

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s
```

```

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;

# System
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 2 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;

### Rules

# System
add (failCac(c1) and failCac(c2)) iff failSystem(s);

# Component
add failCac(c1) => highLoadCac(c2);
add failCac(c2) => highLoadCac(c1);

# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1;

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);

### Priors
add [cac_id] failMotor(cac_id) -8; # p = 5 x 10-4
add [cac_id] failMotorController(cac_id) -8; # p = 5 x 10-4
add [cac_id] failCompressor(cac_id) -8;
add [cac_id] failCanBus(cac_id) -8;

### Query
mcsat_params 10000000, 0.01, 20.0, 0.01, 100;

ask[c] failMotor(c);
ask[c] failCanBus(c);
ask[c] failCac(c);
ask[s] failSystem(s);

```

cac-model2-demo.pcein

```

## Copyright: SRI International, 2010.
##

```

```

## This MLN simulates failures in a dual-CAC system
## using a more detailed failure model.

### Definitions

sort cac_id; # can be only c1 or c2 now
sort sys_id; # can be only s

# Component
predicate failCac(cac_id) indirect;
predicate highLoadCac(cac_id) indirect;
predicate failMotor(cac_id) indirect;
predicate failMotorController(cac_id) indirect;
predicate failCompressor(cac_id) indirect;
predicate failCanBus(cac_id) indirect;
predicate unstableMotorController(cac_id) indirect;
predicate maxpowerMotor(cac_id) indirect;
predicate loosebladesMotor(cac_id) indirect;
predicate overspeedMotor(cac_id) indirect;
predicate overheatMotor(cac_id) indirect;

# System
predicate failCooling(sys_id) indirect;
predicate failSystem(sys_id) indirect;

# We will only model 1 system with 2 cacs
const s: sys_id;
const c1: cac_id;
const c2: cac_id;

### Rules

# System
add (failCac(c1) and failCac(c2)) iff failSystem(s);

# Component
add failCac(c1) => highLoadCac(c2);
add failCac(c2) => highLoadCac(c1);

# Cac
add [cac_id] (failMotor(cac_id) or failMotorController(cac_id) or
  failCompressor(cac_id) or failCanBus(cac_id)) iff failCac(cac_id);

# High Load => Cac failure
add [cac_id] highLoadCac(cac_id) => failCac(cac_id) 0.1;

# Motor and Heating
add [cac_id] unstableMotorController(cac_id) iff maxpowerMotor(cac_id);
add [cac_id] maxpowerMotor(cac_id) iff overspeedMotor(cac_id);
add [cac_id] overspeedMotor(cac_id) iff loosebladesMotor(cac_id);
add [cac_id] loosebladesMotor(cac_id) iff failMotor(cac_id);
add [cac_id] overspeedMotor(cac_id) iff overheatMotor(cac_id);
add [sys_id, cac_id] (failCooling(sys_id) and overheatMotor(cac_id)) iff

```

```

    failMotor(cac_id);
#add [cac_id, sys_id] loosebladesMotor(cac_id) iff failSystem(sys_id);

### Priors
add [cac_id] failMotor(cac_id) -8; # p = 5 x 10-4
add [cac_id] failMotorController(cac_id) -8; # p = 5 x 10-4
add [cac_id] failCompressor(cac_id) -8;
add [cac_id] failCanBus(cac_id) -8;

### Facts
add unstableMotorController(c1);
add overspeedMotor(c2);

### Query
mcsat_params 10000000, 0.01, 20.0, 0.01, 100;
ask[c] failMotor(c);
ask[c] failCac(c);
ask[s] failSystem(s);

```