SAFETY ASSURANCE TECHNIQUES FOR AUTONOMOUS CYBER PHYSICAL SYSTEMS

By

Charles A. Hartsell

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
in

Electrical Engineering

September 30, 2021

Nashville, Tennessee

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Gabor Karsai, for his guidance and support throughout the Ph.D. program as well as Nag Mahadevan, Professor Abhishek Dubey, and Professor Ted Bapty for their regular guidance and feedback which influenced the direction of my research. I would also like to thank my many other Institute for Software Integrated Systems colleagues and the members of my doctoral committee, Professors Bharat Bhuva, Richard Alan Peters, and Jules White.

I would like to acknowledge several of my graduate student peers for their friendship and research collaboration including Ibrahim Ahmed, Dimitrios Boursinos, Nate Hamilton, Tim Krentz, Diego Manzanas Lopez, Patrick Musau, Dr. Brad Potteiger, Ayana Wild, and Ben Yett among many others.

Finally I would like to thank my family and friends for their continued support and encouragement, including my parents Bill and Janice Hartsell, sister Emily Hartsell, and fiancée Julia Brock.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AADL	Architecture Analysis and Design Language
ABD	Assurance Based Development
AdvoCATE	Assurance Case Automation Toolset
AEBS	Automatic Emergency Braking System
ALARP	As Low As Reasonably Practicable
ALC	Assurance-based Learning-enabled CPS
ASARP	As Safe As Reasonably Practicable
AV	Autonomous Vehicle
BBN	Bayesian Belief Network
BTD	Bow-Tie Diagram
CBD	Component Based Development
CNN	Convolutional Neural Network
CPN	Colored Petri Net
CPS	Cyber Physical System
DAG	Directed Acyclic Graph
DCCA	Deductive Cause-Consequence Analysis
DDS	Data Distribution Service
DSML	Domain Specific Modeling Language
ENTRUST	Engineering of Trustworthy Self-adaptive Software
FHA	Functional Hazard Analysis
FMEA	Failure Modes and Effects Analysis
FMI	Functional Mockup Interface
FSS	Failure-Sensitive Specification
FTA	Fault Tree Analysis
GPS	Global Positioning System
GSN	Goal Structuring Notation
IMU	Inertial Measurement Unit
LEC	Learning Enabled Component
LES	Learning Enabled System
MAPE	Monitor-Analyse-Plan-Execute
MBE	Model Based Engineering
ML	Machine Learning

MQTT	MQ Telemetry Transport
MTBF	Mean Time Before Failure
NASA	National Aeronautics and Space Administration
OOD	Out-Of-Distribution
RAF	Royal Air Force
RBD	Reliability Block Diagram
ReSonAte	Runtime Safety Evaluation in Autonomous Systems
RIAPS	Resilient Information Architecture Platform for Smart Grid
RISC	Risk Informed Safety Case
ROS	Robot Operating System
SDL	Scenario Description Language
SRM	Safety Risk Management
TFPG	Timed Failure Propagation Graph
UAS	Unmanned Aerial System
UUV	Unmanned Underwater Vehicle
WAM	Wide-Area Multilateration
XMPP	eXtensible Messaging and Presence Protocol

CHAPTER 1

Introduction

We interact with a variety of Cyber Physical Systems (CPSs) in our everyday lives directly through consumer goods such as automobiles and smart homes as well as indirectly through examples including energy infrastructure and manufacturing systems. While these systems are already pervasive in society, they have further potential in a variety of applications with commonly identified grand challenge problems including advanced/smart power grids, autonomous transportation, and improved biomedical and healthcare systems (Rajkumar et al., 2010), (Baheti and Gill, 2011). However, as these systems are expected to perform additional tasks, often with increasing levels of autonomy, the required level of complexity also increases and poses new design and development challenges as identified by various authors - e.g. (Lee, 2008), (Mosterman and Zander, 2016), (Fitzgerald et al., 2014).

A defining characteristic of CPSs is a tight interconnection between the computational and physical aspects of the system which cannot be easily decoupled. Instead, effective CPS design and analysis requires an understanding of both aspects and typically requires expertise across multiple engineering domains. Well established techniques such as Component Based Development (CBD) and Model Based Engineering (MBE) emphasize the *separation of concerns* concept which allows for efficient management of the complexity and difficulty involved in developing modern CPSs by subdividing the design process into specific aspects which can be analyzed, understood, and addressed separately from other aspects. To address each aspect, the various engineering disciplines often rely on their own specialized modeling languages and methods, typically supported by numerous software tools (eg. simulators, analysis tools, CAD software, etc.). However, the domain interdependence commonly seen in CPSs challenges this separation of concerns concept and adequately understanding their behavior requires tighter integration between the models and methods of the various engineering domains.

There is a constant demand for higher levels of autonomy across the CPS domain, and development of these autonomous systems poses new challenges. Such systems must operate in highly uncertain environments and react appropriately to constantly changing conditions. Many of the functions required for autonomy - e.g., object perception from camera images - are not well suited to traditional, analytically-derived solutions. Instead, designers are increasingly using data-driven methods such as machine learning to overcome these challenges, leading to the introduction of LECs in CPSs. These LECs have demonstrated good performance for a variety of traditionally difficult tasks such as object detection and tracking (Held et al., 2016), robot path planning in urban environments (Sombolestan et al., 2018), and attack detection in smart power grids (Ozay et al., 2016). Systems which use LECs, referred to as a Learning Enabled System (LES), can enable higher levels of autonomy than previously possible, but also introduce unique challenges in each stage of the development life cycle.

Many potential applications for autonomous CPSs involve safety-critical tasks or otherwise have enormous costs associated with system failure - e.g., autonomous passenger vehicles and smart power grids. Before any such system can be operationally deployed, strong assurance that the system will operate safely and correctly in the intended environments is required. While safety assurance is not a new problem, the ever increasing scale and societal impact of CPSs demands higher levels of assurance than previously required. Additionally, existing assurance techniques, particularly for safety-critical systems, are costly and rarely updated once a system is operationally deployed. Autonomous CPS bring new challenges such as continued machine learning from operational data and cyber security vulnerabilities which must be addressed through periodic software updates. Each change after a system has been operationally deployed requires a corresponding revision of system safety assurance. Therefore, new safety assurance techniques are needed which enable iterative revision and improvement and reduce the associated costs.

While numerous formal methods exist that can provide guarantees for certain system properties, these methods are limited in scope and require various assumptions to be made about the system and environment. While these methods are useful, the complexity of real systems and environments prevents any formal guarantees regarding overall safety. In practice, safety assurance involves constructing a well-reasoned, compelling argument that the system is fit for the intended purpose and using this argument to convince stakeholders - e.g., system developers, regulatory bodies, society at large, etc. - that the system will operate safely at all times.

There are a wide variety of techniques for providing evidence in support of the safety assurance argument including thorough field testing, formal analysis of system models, and system simulation among others. However, these techniques are often insufficient for highly autonomous CPS, particularly those utilizing LECs. Additionally, safety is a property of a composed system but the various analysis techniques often produce heterogeneous evidence artifacts in support of limited, component-specific safety properties. Composing the available evidence into a comprehensive and compelling argument for overall system safety is a non-trivial task.

System development processes have traditionally relied heavily on existing standards and regulations to provide guidance for what safety assurance techniques to use and the proper methods to apply them. However, standards have struggled to keep pace with technological advancement and the increasingly difficult tasks modern systems are expected to perform. The one-size-fits-all approach offered by most standards is often too restrictive and inefficient for the highly variable demands of autonomous CPSs. There has been a shift from these strict, prescriptive approaches to more flexible, goal-setting approaches which enable system developers to determine the best methods of achieving and demonstrating safety.

Clear, precise argumentation becomes particularly critical for understanding and communicating how a system achieves safety when using these highly flexible approaches. Informal or implicit arguments are prone to misinterpretation which can ultimately lead to potentially severe accidents. Assurance cases (Rhodes et al., 2010) are one accepted approach for clearly and precisely presenting arguments through the use of a hierarchical tree structure. Root nodes of this tree correspond to a particular system property in question while leaf nodes represent the individual evidence artifacts. The interior structure of the tree explains the logical argument used to connect the various pieces of evidence in support of the system property. Assurance cases are a reoccurring topic throughout this dissertation and are introduced in more detail in Section 2.1.

1.1 Research Contributions

The research contributions of this dissertation are presented as a series of publications where each publication offers a solution to a specific challenge in the field. These publications are presented in Chapters 3 through 5 and a brief foreword is provided at the beginning of each chapter which contextualizes the specific research challenge and summarizes the contribution. These contributions are:

- A CPN based formal analysis technique for real-time, component-based software systems such as autonomous CPSs. The primary focus of this technique is analysis and verification of timing requirements, but various other formal analyses are also supported. The technique is applied to an autonomous Unmanned Aerial System (UAS) to produce best and worst case bounds on various stimulus-to-response timings.

- A platform for development of CPSs which utilize LECs known as the Assurance-based Learning-enabled CPS (ALC) Toolchain. This platform integrates safety assurance processes into the model-based development cycle for CPSs with special consideration for the demands of data-driven software development. An illustrative example is provided where the ALC Toolchain is used to develop an autonomous Unmanned Underwater Vehicle (UUV). This platform also serves as a prototyping tool for the assurance case construction and evaluation contributions.

- A technique for automated construction of assurance cases based on the instantiation and composition of existing argument patterns. This method automates the collection and organization of necessary information by extracting it directly from existing system design models. The method is used to generate a partial assurance case for a UUV example system designed using the aforementioned ALC Toolchain.

- A methodology for modeling potential hazard escalation paths and dynamic estimation of the risk posed by these hazards known as ReSonAte. These risk estimates are used to periodically reevaluate assurance arguments as a system operates under changing internal and environmental conditions. ReSonAte is applied to an AV example system and the estimated risk levels are validated against simulation data.

1.2 Organization

The remainder of this proposal is organized as follows: Chapter 2 provides additional introduction to safety assurance and examines related work in this field, Chapters 3 through 6 present each of the research contributions summarized in the previous section, and Chapter 7 ends with concluding remarks.

CHAPTER 2

Background and Related Work

2.1 Safety Assurance

While several definitions for safety are available, this dissertation uses the definition presented in the NASA System Safety Handbook which states "Safety is freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment and property, or damage to the environment" (Dezfuli et al., 2011). However, achieving perfect safety is an impossible goal for any engineered system. Instead, safety engineering tries show that systems meet an acceptable level of safety and adhere to safety engineering principles such as the As Safe As Reasonably Practicable (ASARP) principle. The same NASA Handbook provides the following definition for this principle: "The system is ASARP if an incremental improvement in safety would require a disproportionate deterioration of system performance in other areas" (Dezfuli et al., 2011). In this same handbook, Dezfuli et al. present a system safety framework with key guidelines including:

- Clear safety objectives for the system should be established at the outset of the project.
- Safety analysis should be started as early as possible in the concept phase. Safety concerns should influence system design choices, not simply rationalize them after system design is complete.
- The plan for achieving an acceptable level of system safety should be further developed and updated at every key decision point throughout the system life cycle. This is not limited to the development stage, but also includes operations, maintenance, closeout, etc.

A key point of this approach, which has been echoed by other authors (e.g., (Leveson, 2011b), (Rinehart et al., 2015)), is that safety assurance should be an integral aspect of the system development process which evolves alongside the system design regardless of the specific safety approach used. In current practice, safety is too often treated with a "checking-the-box" mentality where assurance tasks may only be done once near the end of system development in order to satisfy regulatory requirements. This often leads safety assurance to be an exercise in rationalizing an existing system design instead of a guiding consideration in the process of developing that design. Instead, safety must be treated with a mentality of constant improvement and integrated as a fundamental part of the system development process.

Traditional safety assurance techniques have relied heavily on compliance with existing standards and regulations (eg. DO-178C (RTCA, 2011) for aviation software or ISO 26262 (ISO, 2018) for automotive domain) which typically prescribe specific development processes and goals which must be met such as requirements traceability, test case coverage, and independent verification of results among many others. However, these techniques are found to be lacking when applied to modern CPS development. While these standards are valuable parts of the safety assurance process which have undoubtedly improved system safety (Rinehart et al., 2015), various authors (e.g., (Leveson, 2011a), (Rinehart et al., 2015)) have identified several deficiencies in the traditional, standards-based approach to safety when applied to modern systems including:

- Inability to match the fast pace of technological change
- Does not ensure system will meet assurance goals
- Inflexible, one size fits all approach to safety
- Reduced ability to learn from experience
- Changing nature of accidents
- New types of hazards
- Increasing complexity and coupling
- Decreasing tolerance for single accidents
- Difficulty in selecting priorities and making safety tradeoffs
- More complex relationships between humans and automation
- Changing regulatory and public views of safety

The introduction of LECs to CPS development has exacerbated this issue since data-driven components are fundamentally different from traditional, analytically-derived solutions and are not well covered by existing standards. Recent reports (EASA, 2020), (Alves et al., 2018) have examined the applicability of existing standards and regulations to highly autonomous systems, including those that utilize machine learning. They find that some assurance processes outlined in these documents can be adopted or extended for machine learning and most are still applicable to higher-level system design. However, they also identify several specific challenges which are not well addressed with current safety assurance techniques:

- Design emphasis is shifted to data preparation, Machine Learning (ML) architecture, and ML algorithm selection.
- Difficultly in clearly defining and maintaining traceability of low-level requirements.
- Lack of predictability and explainability.

- Intended behavior of the system is defined by the training data instead of with explicit functional requirements.
- Inherent risk and uncertainty from LECs is likely to propagate to the system-level. Architectural solutions including conflict-resolution, fault-isolation, and fail-safe mechanisms are needed
- Data-driven components require periodic updates, but frequent system re-validation is infeasible with current techniques.

As the cost, scale, and societal impact of engineered systems continues to increase, safety assurance methods must adapt to fit these systems. Several authors (e.g. (Rinehart et al., 2015), (Leveson, 2011b)) have noted a general trend towards more goal-oriented approaches which allow safety goals and requirements to be tailored specifically to the system in question. This type of approach is better suited to complex systems where the one-size-fits-all techniques used by prescriptive regulation often fail to address all potential safety flaws. A strong, convincing argument structure in support of system safety goals is particularly important for goal-oriented approaches since there is no prescribed technique to guide the safety assurance process. These highly flexible approaches are only recently seeing industry adoption with examples like Eurocontrol's Wide-Area Multilateration (WAM) preliminary safety case (McFarlane, 2007) and NASA's Risk Informed Safety Case (RISC) (Dezfuli et al., 2011).

2.1.1 Assurance Cases

The necessary components of safety assurance arguments can be generally divided into two primary categories: *evidence* which are various artifacts representing knowledge about the system created or collected through analysis techniques, and the *argument structure* which is the logical explanation of how the available evidence provides support for system safety requirements and goals. Traditionally, safety arguments have been presented in unstructured written language which can obscure the structure of the argument. Similarly, many of the traditional safety standards and regulations used as a basis for assurance prescribe required processes and artifacts, but the underlying argument for assurance is implicit. Alternatively, *assurance cases* allow arguments to be presented explicitly through use of a structured argument which explains how certain evidence supports the stated conclusion.

An assurance case can be defined as "A reasoned and compelling argument, supported by a body of evidence, that a system, service or organisation will operate as intended for a defined application in a defined environment" (The Assurance Case Working Group, 2018). Assurance cases are often represented graphically using an appropriate graphical modeling language such as GSN (Kelly, 1999). Arguments in GSN are represented as a hierarchical tree structure where root nodes of the tree correspond to a particular system

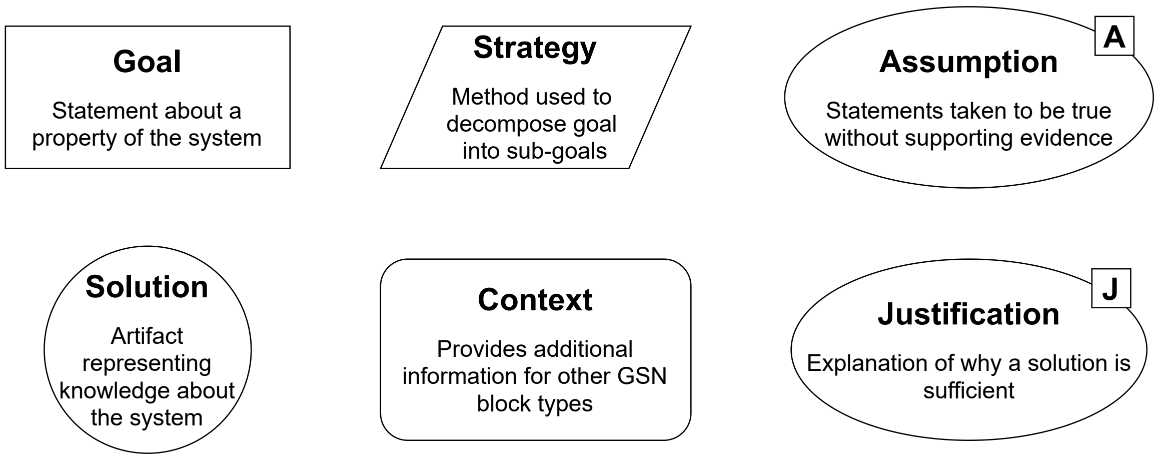

Goal
Statement about a property of the system
Strategy
Method used to decompose goal into sub-goals
Assumption
A
Statements taken to be true without supporting evidence
Solution
Artifact representing knowledge about the system
Context
Provides additional information for other GSN block types
Justification
J
Explanation of why a solution is sufficient

Figure 2.1: GSN modeling elements.

property in question while leaf nodes represent the individual evidence artifacts. The interior structure of the tree explains the logical argument used to connect the various pieces of evidence in support of the root system property.

Arguments in GSN are constructed using six modeling elements, shown in Fig. 2.1, including:

- *Goal*: A statement about a property of the system under consideration.
- *Strategy*: Method used to decompose a goal into sub-goals.
- *Context*: Provides additional information for other GSN block types. For example, may provide necessary definitions.
- *Assumption*: Statements taken to be true without supporting evidence or explanation.
- *Justification*: Explanation of why a solution is sufficient to satisfy a goal.
- *Solution*: Artifact representing knowledge about the system, often collected through analysis or testing.

Top-level goals typically deal with system-level requirements, but the available evidence often provides support for component-level properties. Arguments used to link low-level evidence to high-level goals often involve multiple logical steps and cannot be accurately represented by a single-level assurance case. Instead, *goals* and *claims* can be decomposed into smaller *sub-goals* and *sub-claims*. This decomposition process is repeated as necessary until each step in the supporting argument corresponds to one level in the assurance case. Once a sub-goal is properly supported by evidence, this sub-goal can itself be used as evidence for a higher-level goal or sub-goal.

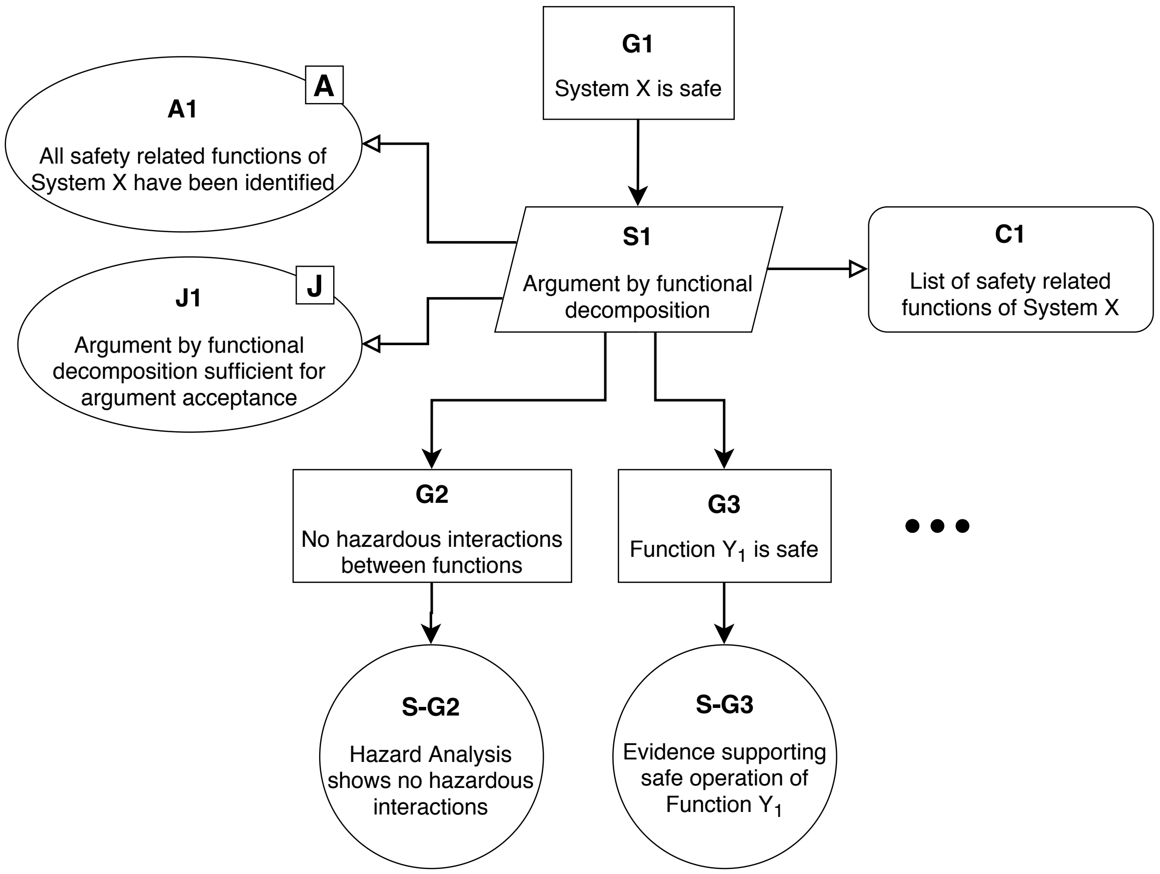

G1
System X is safe
A1
A
All safety related functions of System X have been identified
S1
Argument by functional decomposition
C1
List of safety related functions of System X
J1
J
Argument by functional decomposition sufficient for argument acceptance
G2
No hazardous interactions between functions
G3
Function Y1 is safe
S-G2
Hazard Analysis shows no hazardous interactions
S-G3
Evidence supporting safe operation of Function Y1

Figure 2.2: Example GSN argument tree showing decomposition of a safety goal through functional decomposition.

Figure 2.2 shows an example of how GSN blocks can be used to decompose goals into sub-goals. In this example, the top-level goal "System X is safe" is broken down by a functional decomposition argument. Each of the GSN blocks described above is used in this argument, but not all blocks are restricted to being used in this manner. For instance, the assumption block **A1** in this example is linked to the strategy block **S1**. In general, assumption blocks are not limited to strategy blocks but may also be linked to other block types such as goals and solutions.

By forcing arguments to be constructed in an explicit, structured manner, assurance cases help bring many benefits of MBE to the engineering argumentation domain. While the detailed benefits of MBE are numerous and vary between application domains (eg. Automotive (Broy et al., 2012), Avionics (Le Sergent et al., 2016)), several general benefits are universal to most MBE techniques including assurance cases. These universal benefits include:

- Improved ability to handle complexity by dividing problem into smaller, distinct problems.
- Improved ability to analyze and performing reasoning on models due to their structured nature, usually

with well-defined model semantics. Analysis may be manual or automated.

- Improved re-usability of design artifacts (ie. models) through generalized templates and patterns.
- Earlier identification of potential problems.

Assurance cases may be constructed to support system requirements for a variety of concerns including performance, reliability, and security. In particular, assurance cases are increasingly being used in support of safety-related properties and are called *safety cases* when used in this manner. Rinehart et al. claim seven primary benefits of safety cases over more traditional safety assurance techniques identified through a combination of a literature survey and interviews with field practitioners (Rinehart et al., 2017). The first of these benefits is more fundamental than the others where the authors claim that "Assurance cases are successful where suitable". They explain that the boundary for defining when assurance cases are "suitable" to a particular application is not yet well-defined but is constantly being refined. Instead, the authors support this claim by examining several successful and unsuccessful applications of assurance cases in industry and identifying the general consensus of the related literature. The six remaining claims identify specific benefits of assurance cases over conventional methods and norms including:

- More comprehensive
- Improve allocation of responsibility
- Organize information more effectively
- Address modern certification challenges
- Offer a more efficient path to certification
- Provide a practical, robust way to establish due diligence

While the details relevant to an assurance case are different for each system considered, common patterns emerge in the structure of the arguments themselves. It is useful to document and reuse these common sections with assurance case *patterns* or *templates*. Kelly introduced both this assurance case pattern concept and provided extensions to the GSN modeling language for describing these patterns (Kelly, 1999). The additional modeling objects are used for *entity abstraction*, where a distinction is made between a *class* of objects and an *instance* of that object class, as well as *structural abstraction*, which generalizes relationships between two or more objects. The structural abstraction concepts can be further divided into groups for *multiplicity*, which specifies the number of relationships between objects, and *optionality*, which denotes possible alternatives for satisfying a relationship.

2.1.2 Safety Analysis Techniques

Regardless of the strength of an argument's logical structure, it must always be based on firm evidence in order to be sound. Both system-level and component-level evidence artifacts are typically used in assurance cases, and there are a wide variety of component-level analysis techniques available. These techniques provide evidence that a particular component satisfies its design requirements and will behave as expected, but are often specific to the technology used to implement the component. In this document we are primarily concerned with system-level analysis techniques which provide evidence for systems composed of multiple components.

There are a wide variety of safety analysis techniques used to create these necessary evidence artifacts, and the various techniques can be divided into two categories: *qualitative* and *quantitative*. *Qualitative* techniques, such as Failure Modes and Effects Analysis (FMEA) (Stamatis, 2003), try to identify how a system can fail. *Quantitative* techniques, such as Fault Tree Analysis (FTA) (Lee et al., 1985), extend this by estimating numeric failure rates for any identified failure modes. Rouvroye et al. (Rouvroye and van den Bliek, 2002) identify and compare several generic safety analysis techniques which are applicable to nearly any engineered system. Their analysis includes two qualitative techniques, FMEA and expert analysis, as well as several quantitative techniques including: Parts count analysis (U.S. Department of Defense, 1991), Reliability Block Diagram (RBD) (IEC, 2016), RBD-FTA Hybrid techniques, FTA, Markov Analysis (Lewis, 1995), and Enhanced Markov Analysis (Rouvroye, 2001). The authors find that each approach, in the order listed, offers increasing modeling power at the cost of increasing analysis complexity. They note that the less powerful techniques are likely useful in the early stages of system development, while the more powerful techniques should be applied as the system design matures and more detailed information is available. This is in line with the general consensus that safety assurance should be a continuous, evolutionary process which is started as early in the design cycle as possible.

Among the issues identified by Leveson (Leveson, 2011a), increasing system complexity and the changing nature of accidents are not well handled by existing analysis techniques. These techniques (eg. FMEA, FTA, etc.) often rely on human experts to be able to identify possible component failure modes and understand how these failures will propagate through the system architecture. However, the validity of expert judgement of safety risk has been questioned, particularly when making quantitative estimates instead of qualitative statements (Rae and Alexander, 2017). The heavy reliance on human expertise is no longer reasonable as the complex interactions between components in modern systems may lead to emergent system behavior that is often non-intuitive and unexpected. As Leveson notes, this type of emergent behavior is often a primary accident cause but is rarely identified and addressed by existing analysis techniques. New techniques for adequately understanding these relationships and reducing the burden on system analysts are needed.

Formal analysis techniques can alleviate some of this reliance on human expertise. Haider et al. examine several safety analysis methods including informal, semi-formal, and formal techniques (Haider and Nadeem, 2013). In particular, they examine two formal techniques: Deductive Cause-Consequence Analysis (DCCA) (Ortmeier et al., 2005) which is a formal generalization of FMEA and FTA, and Failure-Sensitive Specification (FSS) (Ortmeier and Reif, 2006) which is a method for constructing system failure modes alongside its specification. Haider et al. argue that formal techniques like these offer several advantages including rigor and completeness. For example, Daskaya et al. apply DCCA to two level crossing controllers used for a railway automation system (Daskaya et al., 2011). They are able to formally verify a number of safety properties, but note severe complexity problems even with their medium-sized designs. Haider et al. note that informal techniques have certain advantages as well, and that both types of analysis techniques should be combined for best results.

Model checking is one class of formal techniques which is particularly useful for ensuring software correctness. These techniques, which have been described by several authors (e.g. (Alur et al., 1990), (Baier and Katoen, 2008)), start with the creation of two models: one for the system to be analyzed and another for the operating environment. These models must capture all relevant properties while abstracting away unnecessary details. The models must be defined using an appropriate formalism with precisely defined execution semantics capable of handling the highly non-deterministic nature of the environment as well as any potential non-determinism present in the system itself. The two models can then be composed into a larger model describing how the system interacts with its environment, and this composed model can be used to generate a state-space which contains all possible states the system may reach within a set time horizon. Analysts may then search this state space to verify any number of properties such as worst-case stimulus-to-response times, the absence of race conditions between components, or that message queues always remain below their set maximum size. Several such techniques have been developed using a variety of modeling languages (e.g. Timed Automata (Alur and Dill, 1994) and CPN (Jensen and Kristensen, 2009b)) with varying levels of automated tool support (e.g. UPPAAL (Bengtsson et al., 1995) and recent CPN-based techniques (Kumar and Karsai, 2015)).

In a 1992 paper, Barroca et al. examine the relevance of formal techniques to safety-critical system development (Barroca and McDermid, 1992). While formal techniques have evolved significantly since this review was published, many of the conclusions about the relevance of these techniques remain valid. Overall, the authors advocate for the use of formal techniques throughout the software life-cycle, but note that there is no unified methodology for the entire development cycle. Instead, each method must be individually evaluated based on expected cost and potential benefit. Since formal techniques often require significant effort and expertise to apply correctly, they should be applied strategically to achieve maximum benefit for minimum cost. Additionally, formal techniques rely on various assumptions and mathematical system abstractions. It is critical that these are adequately validated as any formal guarantees are nullified if the assumptions they are based on become false.

2.2 Assurance Case Construction

As several authors have identified, safety assurance should be an integral part of the system development cycle. Graydon et al. proposed an Assurance Based Development (ABD) method which helps developers make informed design decisions while considering the impact those decisions will have on system safety, functionality, dependability, etc. (Graydon et al., 2007). Under their method, both the system and its assurance case(s) are developed in parallel, and the assurance case is used as a driver for evaluating design decisions. It is an iterative method which can be summarized with the following four steps:

1. The developer examines unsatisfied goals in the assurance case and selects one for examination. The top-level goal of this assurance case should be one or more of the system requirements which must be fulfilled.

2. Next the developer makes a system development choice on how to address the unsatisfied goal. Examples of a development choice are a software architecture selection, choice of programming language, method of assuring component performance, etc.

3. Based on the system development choice, the developer expands the unsatisfied goal with an acceptable assurance argument or direct evidence. This step may result in multiple new, unsatisfied sub-goals.

4. Repeat steps 1 through 3 until all goals are satisfied.

The ABD method also provides seven criteria each development choice should be evaluated against: functionality, restrictions on later choices, evidence of dependability, cost, feasibility, applicable standards, and non-functional requirements. The authors note that this method cannot guarantee optimal design choices, but it helps developers make informed designs choices while considering their impact on system assurance. Using this method to drive the design process also helps identify potential assurance problems early while they are much cheaper and easier to address. The ABD method can also help avoid the common pitfall where system assurance is treated as a singular activity to be performed once at the end of system development. However, exploring a large variety of potential system development choices for every unsatisfied assurance goal is a time consuming process. Appropriate formalisms of the development choices may allow much of the ABD process to be automated as part of a larger architectural design space exploration framework.

Assurance case patterns are a useful way of documenting *successful* argument structures. Exactly what constitutes a successful argument is context specific, but generally includes arguments which are convincing, comprehensive, easily understood, certified by regulators, etc. Kelly and McDermid (Kelly and McDermid, 1998) point out that argument pattern reuse predates the concept of a structured assurance case, but was previously done in an ad-hoc, copy-and-paste manner with textual safety documents. Formalizing this practice with assurance case patterns helps avoid inappropriate reuse of patterns and improves traceability, consistency, and knowledge transfer of the safety assurance process. Additionally, the authors point out opportunities for both *horizontal* and *vertical* pattern reuse, meaning reuse within a specific domain and across multiple domains respectively.

As discussed in Section 2.1, the current most common approach to safety assurance relies on conformance to prescriptive standards and regulations. Holloway (Holloway, 2013) pointed out that these standards often provide detailed development guidance, but do not explain how each particular objective contributes to achieving the overall goal of the standard (eg. system safety). In these cases the assurance argument itself is implicit, but it can be valuable to extract this underlying argument and document it in the form of a structured assurance case template. Holloway describes an on-going effort to apply this to the DO-178C standard for aviation software development, and other similar work has been done for the ISO 26262 standard regarding safety of automotive vehicles (Chowdhury et al., 2017). Holloway identifies three expected benefits from this and similar efforts:

- Enable effective analysis of the adequacy of the reasoning underpinning DO-178C.
- Facilitate intelligent conversations about the relative efficacy of DO-178C and related standards.
- Improve mutual understanding between supporters of prescriptive style and goal-based style standards.

2.2.1 Common Mistakes

Safety cases are a useful engineering tool, but simply using a safety case to present an assurance argument does not guarantee system safety. The Royal Air Force (RAF) Nimrod aircraft accident is a common example of an ineffective safety case which was undermined by a number of safety fallacies. The Nimrod aircraft had been in service with the RAF since 1969, and a safety case for the aircraft had been completed in 2005 after 30+ years of successful operation. However, in September of 2006, a catastrophic in-flight fire on the XV230 Nimrod aircraft resulted in the deaths of all 14 crew members on board. A subsequent report from the U.K. government, "The Nimrod Review" by Charles Haddon-Cave (Haddon-Cave, 2009), identified numerous flaws in the safety processes and the resulting safety case constructed for the Nimrod aircraft. Haddon-Cave provides many valuable future recommendations over a range of safety-related topics, and a few of the key failures and recommendations identified are summarized as follows:

- Widespread assumption that the Nimrod was safe due to its operational history. This caused the safety case construction to be treated as a documentary exercise instead of a true safety analysis.
- The safety case development was under-resourced and several shortcuts were taken to stay on schedule and within budget. This led to many of the identified potential safety hazards being left in a "Unclassified" (not yet classified based on expected risk) or "Open" (analysis work incomplete) states in the final safety case.
- The safety case regime has lost its way, leading to a culture of *paper safety* where the safety case is treated as a means to an end at the expense of real safety.

Haddon-Cave notes that the lessons learned from the Nimrod XV230 are not new and compares the causes to those identified in the investigation of the National Aeronautics and Space Administration (NASA) Space Shuttle Columbia accident (Gehman, 2003) among other similar accidents. However, the Nimrod is distinct from Columbia in that a safety case had been completed for safety assurance. This serves as a reminder that developers must be aware of common pitfalls when constructing safety cases and actively take steps to avoid or mitigate the impact of these fallacies. In particular, the "checking the box" mentality where safety assurance is treated simply as a requirement that must be satisfied for certification should be avoided in favor of an ongoing cycle of analysis and improvement. The following paragraphs discuss several more common pitfalls and methods to avoid them in more detail.

Koopman et al. examined common safety argumentation approaches, how they apply to autonomous systems, and identified common faults and anti-patterns present in these arguments with an emphasis on autonomous ground vehicles (Koopman et al., 2019). In particular, the authors examine five common approaches for arguing safety: conformance to standards, "proven in use" arguments based on operational history, brute force testing, data collection through simulation, and formal proofs of correctness. Several potential pitfalls are explained for each of these approaches, and a complete listing is omitted here for brevity. However, some of the key issues are summarized below, particularly those which resonate with similar findings of other authors in the autonomous system safety literature:

- *Human filter pitfall*: Systems are often designed to be robust against common human failures, but autonomous components do not necessarily share these common failure modes.

- *Insufficient testing pitfall*: Brute force testing needs to be several times longer than the acceptable Mean Time Before Failure (MTBF) to be statistically significant, which is often an infeasible amount of testing.

- *Fly-Fix-Fly anti-pattern*: Improving system dependability by addressing problems as they appear during testing is a common approach. The pitfall here is that developers tend to take credit for all previous field testing after each fix is applied. However, the validity of previous testing is suspect since there is no guarantee that each new fix does not introduce new problems. Regression testing can provide some confidence that previous test results are still valid, but the coverage and real-world applicability of these tests must closely examined. For example, common software testing may not fully address all sources of non-determinism present in real environments.

- *Assumption validity pitfall*: Assumptions used in formal verification techniques, or in the wider development process, must be valid in the deployed system. If they are not guaranteed to be valid, then they must be actively monitored in the operational system.

Additionally, Koopman et al. note that other authors have proposed publishing a catalog of accepted argumentation patterns (eg. (Graydon et al., 2012), (Szczygielska and Jarzębowicz, 2017)). They extend this idea by proposing a catalog of commonly attempted but invalid argumentation patterns.

Leveson pointed out that there is often more value to be gained in arguing that a system is unsafe rather than trying to prove that a system is safe (Leveson, 2011b). As part of the future recommendations from the Nimrod Review, Haddon-Cave goes as far as to recommend safety cases be renamed "risk cases" with the goal of "demonstrate that the major hazards of the installation and the risk to personnel therein have been identified and appropriate controls provided" (Haddon-Cave, 2009). Similarly, Leveson argues that safety cases are prone to confirmation bias, particularly when attempting to show that system safety properties are satisfied (Leveson, 2011b). Leveson agrees with Haddon-Cave to an extent by concluding that assurance cases should focus on showing a system is unsafe. Additionally, Leveson notes that independent safety assessment can be helpful in avoiding the problem of confirmation bias.

2.2.2 Automated Construction

Construction of assurance cases is a valuable safety task, but manual construction is a costly and time-consuming process. Safety analysis and assurance tasks are often delayed due to the associated costs, and may be treated as one-time activities instead of an ongoing process. Techniques such as argument instantiation from existing, accepted patterns can help reduce these costs, but are still highly manual processes. Several methods for further automating the assurance case construction process have been proposed and are discussed in the remainder of this section.

Gacek et al. propose a framework for assurance case generation known as *Resolute* (Gacek et al., 2014). Resolute assurance cases combine information about the system derived from two different sources: 1) System architecture models specified in Architecture Analysis and Design Language (AADL) (Feiler et al., 2006), and 2) a set of user-defined set of claims and logical rules for satisfying those claims specified in Resolute's own Domain Specific Modeling Language (DSML). Based on this information, an assurance case argument is generated which shows how the various safety claims can be proven from sub-claims based on the given system architecture. Resolute can also incorporate the results of external formal analysis which have been performed on the AADL model to determine whether or not these sub-claims have been satisified. Since the assurance case is directly tied to the AADL architecture model, changes to the system architecture will automatically result in an updated assurance case argument. Additionally, the authors provide an implementation of the Resolute framework in the Open Source AADL Tool Environment (OSATE) (Feiler, 2019) plug-in for the Eclipse IDE.

Calinescu et al. present another method known as Engineering of Trustworthy Self-adaptive Software (ENTRUST) (Calinescu et al., 2018). Similar to Resolute, ENTRUST attempts to prove or disprove various system safety properties stated in an appropriate formal language based information available in system models. However, ENTRUST requires the system models themselves to be formally verifiable as well. The ENTRUST framework is also targeted at self-adaptive systems which may reconfigure themselves during runtime operation. To address this, ENTRUST includes a high-level Monitor-Analyse-Plan-Execute (MAPE) control loop for updating the system assurance case as the system evolves. During the analysis and planning steps, models are re-verified based on the updated system configuration to generate *adaptation assurance evidence*. This evidence is then used to update the assurance case and re-examine the validity of the system safety properties in the current configuration. Dynamically constructing an updated assurance case after each system reconfiguration allows this technique to scale to systems with a large space of potential adaptations where explicit consideration of every possible adaptation during system design would be infeasible.

2.3 Assurance Case Evaluation

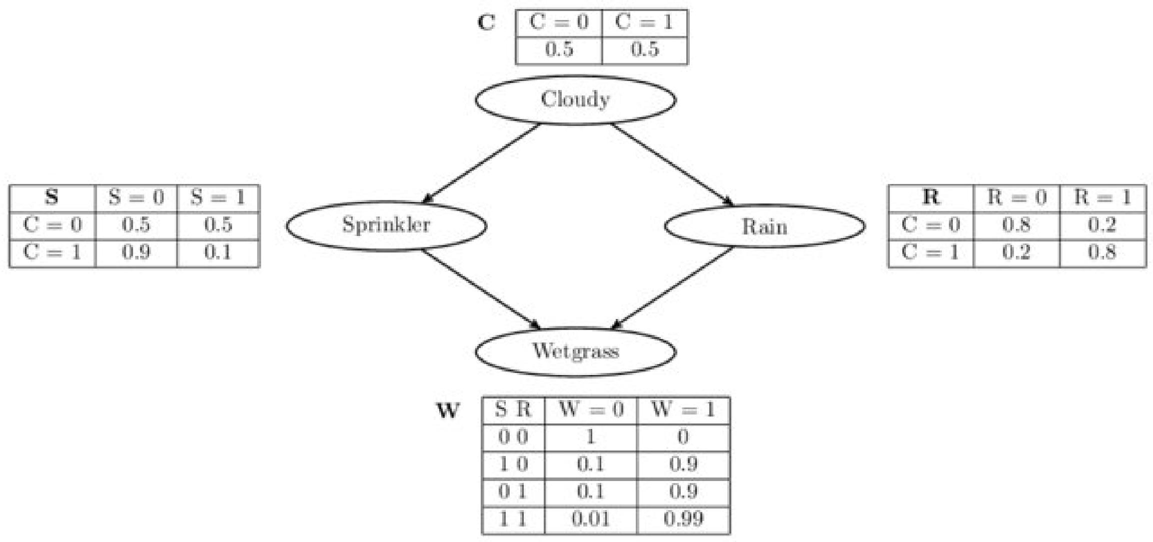
The process of constructing an assurance case can be a valuable assurance exercise by itself since it forces developers to critically analyze system safety properties. However, thorough evaluation of the argument after it has been constructed is critical for safety assurance. Simple examination by an independent assessor is beneficial, and several structured evaluation methods have been proposed. A variety of such techniques are discussed in the remainder of this section.

When evaluating an argument in any form - structured, unstructured, textual, graphical, etc. - it is important to consider whether the argument is *deductive* or *inductive*. *Deductive* arguments are those where the truth of the premises necessarily implies the truth of the conclusion. The argument for Socrates' mortality is a classic example of a deductive argument stated as two premises and a conclusion: (P1) Socrates is a man and (P2) all men are mortal, therefore (C) Socrates is mortal. In this argument, if premises P1 and P2 are true, then the conclusion C is necessarily true.

It is often impossible to adequately support the strong premises in a deductive argument when applied to real systems. *Inductive* arguments, where the truth of the premises strongly supports but does not guarantee the truth of the conclusion, may be used instead. Graydon presents an example of an inductive engineering argument based on identification and mitigation of hazards, a common technique for safety assurance, summarized as follows (Graydon, 2015). (P1) Potential hazards have been adequately identified. (P2) All identified hazards have been sufficiently mitigated. (C) All hazards are sufficiently mitigated. This argument is inductive since it provides strong support for the conclusion C, but cannot guarantee that it is true since premise P1 does not claim that all hazards have been identified. This argument could be made *deductive* by replacing premise P1 with a stronger claim such as: (P1) All potential hazards have been identified. However, such a strong claim is impossible to support in a real system since it is always possible another potential hazard exists which has not been identified.

As Graydon points out, some level of induction is unavoidable in arguments for real systems and forming the argument in a deductive way only relocates the induction from the argument structure to the task of supporting the argument premises. However, the question of where this induction should reside for the argument to be most effective still remains. Rushby argues that every logical step in an assurance case can be divided into one of two categories: *reasoning* or *evidential* (Rushby, 2015). *Evidential* steps describe what is known about the system (ie. provide evidence for the assurance case), while *reasoning* steps describe logical connections between the available evidences and the conclusion they support. In his view, reasoning steps should be purely deductive and all inductive reasoning is restricted to the evidential steps. This allows the assurance case to be analyzed in a modular fashion by avoiding the hidden dependencies that may occur with inductive reasoning. Rushby also extends the definition of soundness and considers an argument to be *sound* if the reasoning steps are deductively valid and all evidential steps cross the established acceptance thresholds.

Since engineering arguments cannot be made entirely deductive, methods are needed for evaluating the strength of inductive arguments. Goodenough et al. propose a framework for justifying confidence in assurance case claims based on the concepts of *defeasible reasoning* and *eliminative induction* (Goodenough et al., 2012). *Defeasible reasoning* is a type of inductive reasoning where the exceptions that may invalidate



C = 0	C = 1
0.5	0.5

S	S = 0	S = 1
C = 0	0.5	0.5
C = 1	0.9	0.1

R	R = 0	R = 1
C = 0	0.8	0.2
C = 1	0.2	0.8

S R	W = 0	W = 1
0 0	1	0
1 0	0.1	0.9
0 1	0.1	0.9
1 1	0.01	0.99

Figure 2.3: Example Bayesian Belief Network. Reproduced from (Han et al., 2008).

the conclusion are listed explicitly and known as *defeaters*. For example, a defeasible argument for hazard identification may look as follows. (P1) Potential hazards have been adequately identified and (P2) all identified hazards have been sufficiently mitigated. Therefore, (C) all hazards are sufficiently mitigated unless (D) a previously-unknown type of hazard was not considered. In this argument, the defeater *D* is a potential exception that would invalidate the conclusion *C*. Goodenough et al. also identify and discuss three categories of defeaters: rebutting, undercutting, and undermining.

Eliminative induction is a method for evaluating a hypothesis which says that the confidence in the hypothesis increases as reasons for doubting the truth of the hypothesis are eliminated. If there are n possibilities for doubt and i of these have been eliminated, then the *Baconian probability* that the hypothesis is true is $\frac{i}{n}$. Eliminative induction is an idealized method since it is generally impossible to exhaustively list all possible defeaters for any given argument. However, Goodenough et al. argue that this technique is still useful in practical applications since the primary sources of uncertainty can be identified and addressed, while other minor sources are considered negligible. The author's proposed framework includes all identified defeaters directly in the assurance argument, usually stated as GSN sub-goals, and uses Baconian probability as a metric of confidence in the argument. This concept is closely related to Rushby's observation that inductive arguments can often be made deductive by explicitly stating the absence of any conditions which may invalidate the argument as assumptions (Rushby, 2015).

Eliminative induction combined with Baconian probability provides one technique for quantifying the level of confidence provided by a particular assurance case. However, this is by no means the only quantitative assurance case evaluation method. Several approaches have been developed on the basis of a Bayesian Belief

Network (BBN) (Jensen et al., 1996). BBNs are a type of Directed Acyclic Graph (DAG) where nodes in the graph represent probabilistic variables and the edges between nodes denote conditional dependency between nodes. The probability values for each node are computed based on Bayesian statistics (Lee, 1989). The example BBN in Figure 2.3 shows the probability that a field of grass is currently wet conditioned on two other variables: whether or not a sprinkler system is on and it is raining. These two variables are themselves dependent on yet another variable: if the weather is cloudy or clear.

Denney et al. propose one such evaluation method where leaf nodes in the BBNs represent possible sources of uncertainty in the argument claims (Denney et al., 2011). Each source of uncertainty is rated based on the analyst's confidence in that source using a five-point scale ranging from *Very Low* to *Very High*. Similarly, Hobbs et al. propose another method where assurance arguments are directly represented as BBNs (Hobbs and Lloyd, 2012). In this way, each leaf node represents evidence and the structure of the BBN represents the argument. Additionally, the authors claim that *noisy-or* and *noisy-and* logical combinations may more accurately represent argument reasoning when compared to the standard probability *and* and *or* operators. While these techniques are useful, it is often difficult to determine conditional probability values and even more difficult to verify the accuracy of these values. Bayesian statistics is prone to large conditional probability tables due to the size of the Cartesian product quickly increasing when nodes are dependent on more than one variable.

Other quantitative methods based on Dempster-Shafer theory (Sentz et al., 2002) try to reason about both the level of *belief* in a particular claim as well as the *plausibility* of the claim. Ayoub et al. propose one such method which requires an analyst to independently assess the sufficiency of each evidence node (Ayoub et al., 2013). Then, combination rules can be used to assess claims which are directly supported by this evidence. The same technique is then applied again for the next higher level of claims and this process repeats until all claims have been examined. Other techniques, such as the method proposed by Duan et al. (Duan et al., 2014), are based on Jøsang's opinion triangle (Jøsang, 2001) which extends Dempster-Shafer theory into three variables: *belief*, *disbelief*, and *uncertainty*.

While quantitative evaluation methods can be useful for estimating probabilities, some authors have criticised the validity of these estimates. Graydon et al. examine 12 unique methods and identify counter examples which produce implausible results for each method (Graydon and Holloway, 2017). Each technique examined uses an illustrative example to demonstrate the validity of the probability estimates produced. However, Graydon et al. are critical of this approach and argue that each technique should be evaluated in a more thorough review process and supported with direct evidence. Any new techniques developed or extensions of existing techniques should keep these criticisms in mind as they are very relevant for accurate assurance case evaluation.

2.4 Tool Support for Safety Assurance

Several authors have identified safety as an emergent property of a complete system, not something that can be determined for individual components in isolation (Leveson, 1995), (Dekker et al., 2011), (Yang et al., 2017). Safety should be an integral consideration in all stages of development in order to adequately understand the complex interactions which may lead to unsafe conditions and address all potential concerns. As discussed in the previous sections, safety is too often treated with a "check-the-box" mentality where tasks are done only once for the purpose of certification. Additionally, safety related tasks are often delayed until later stages of the development cycle due to their associated costs. Appropriate tool support and automation of mechanical safety-related tasks is necessary to efficiently and effectively integrate safety into the development process.

Construction and evaluation of small assurance cases with sizes in the tens of nodes can be done well enough with ad-hoc processes and notation tools. However, assurances cases for complete systems are typically on the scale of hundreds or thousands of nodes and quickly become unmanageable without appropriate tool support. Assurance case editors fill this need and, at a minimum, provide an environment for manual construction and examination of assurance cases. Many editors provide significantly more functionality than this minimum, with one prime example being the Assurance Case Automation Toolset (AdvoCATE) (Denney and Pai, 2018). Some of the additional functionalities provided by AdvoCATE include: instantiation of argument patterns, hierarchical and modular abstraction, support for user-defined queries and generation of custom views based on these queries, integration of formal methods, and verification of argument structure properties. Additionally, AdvoCATE automates these functions as much as possible, significantly reducing developer time spent on highly mechanical tasks. AdvoCATE provides the highest automation level of the existing tools identified, but several other editors are available (eg. (Adelard, LLP, 2011), (Matsuno, 2011), (Barry, 2011), (Steele et al., 2011), (Aiello et al., 2014)) with varying features and levels of automation. Maksimov et al. have also completed a more detailed survey and evaluation of the various assurance case tools available (Maksimov et al., 2018).

Tools focused solely on safety assurance activities such as assurance case editors are useful, but do not fully address the problem of integrating safety assurance into all stages of design cycle. Sztipanovits et al. identify technical barriers in model and component based design for CPS which have prevented higher levels of productivity gains seen in other domain (Sztipanovits et al., 2014). In particular, heterogeneity of CPS models, design tools, and constraints throughout the system life-cycle pose significant challenges to realizing the full benefits of model-based design methods. *Separation of concerns* is a common strategy for managing this heterogeneity where the overall design problem is decomposed according to physical phenomena, level of abstraction, or engineering discipline. This principle has led current CPS design toolchains to be vertically integrated within specific disciplines. However, isolating aspects of the problem in this way leads to decreased predictability of system properties. The authors introduce the OpenMETA tool suite which provides horizontal integration layers and allows for tighter integration between the various design aspects. While OpenMETA is not focused on safety, the horizontal integration concepts developed are useful for integrating safety-related modeling and analysis as a primary consideration throughout the development cycle. The lessons learned from this effort are further explored in a later work (Sztipanovits et al., 2015) and should be considered in the development of future CPS design tools.

Larsen et al. propose another tool chain known as INTO-CPS which addresses the multidisciplinary challenges associated with CPS development (Larsen et al., 2016). INTO-CPS integrates many heterogeneous models with a focus on co-simulation of these models using the Functional Mockup Interface (FMI) (Blochwitz et al., 2012). Similar to the semantic backplane of the OpenMETA tool suite discussed above, INTO-CPS builds on formal foundations to integrate heterogeneous models in a precise manner. INTO-CPS also ensures traceability of various artifacts throughout the development cycle, including refinement of requirements all the way to component implementations. However, the LECs commonly used in CPSs challenge the typical requirement refinement process as discussed in Section 2.1. INTO-CPS also addresses the need to assure system safety, but their approach is limited to showing compliance with existing standards such as ISO 26262.

While various integrated toolchains have been developed for the specific challenges encountered in CPS development, few of these tools adequately consider safety aspects. As discussed in Section 2.1, existing safety assurance techniques are insufficient for autonomous CPSs. A more thorough approach is needed which treats system safety as a primary engineering discipline to be integrated in the development process. Many of the lessons learned from the development of existing toolchains, particularly those which integrate many heterogeneous disciplines and modeling languages, should be applied to the development of any CPS toolchain focused on safety assurance.

CHAPTER 3

CPN-Based Timing Analysis

Foreword

Component based design is a powerful tool for software development which allows complex problems to be decomposed into several smaller tasks which can be solved individually. Components must be able to communicate with one another which is most commonly achieved with some form of message passing. Several different message passing protocols are available such as ZeroMQ (Hintjens, 2013), eXtensible Messaging and Presence Protocol (XMPP) (Saint-Andre et al., 2004), and MQ Telemetry Transport (MQTT) (Banks and Gupta, 2014). Most messaging protocols support multiple messaging patterns such as publish-subscribe, client-server, and push-pull among others.

Nearly all component based systems rely on a set of core services which allow for effective interaction between components. Typical services may include a message passing interface, time synchronization between components, and management of shared system parameters among others. These are often not provided by the chosen operating system, leading many systems to rely on some form of *middleware*. A middleware is a software framework which operates between the operating system and the system components and provides these services. Component based systems will often use some form of middleware which may be a custom development, or one of many pre-existing solutions such as the Robot Operating System (ROS) (Quigley et al., 2009), Resilient Information Architecture Platform for Smart Grid (RIAPS) (Eisele et al., 2017), or Data Distribution Service (DDS) (Pardo-Castellote, 2003).

While component based design is effective for handling complexity, it also introduces non-trivial interactions between components which may lead to new errors in the system design. These interactions may cause a system to fail even when all components individually are behaving as expected, a problem Leveson identifies as *interactive complexity* (Leveson, 2011a). This type of complexity can lead to many non-intuitive problems such as race conditions between messages, disagreement on expected behavior between components, or deadlock situations due to problems such as priority inversion. Additionally, understanding the limitations of the hardware where components will execute such as execution times, sizes of message queues/buffers, and message propagation delay is critical to assuring safe system operation.

The following publication presents a CPN based analysis technique for component based software systems which utilize a message-passing middleware, a common architecture for many autonomous CPSs. The technique extends previous work by Kumar et al. (Kumar and Karsai, 2015) with additions including: (1) an improved generic application model which supports more fine-grained task modeling, preemption by other tasks, and variable task execution time, (2) a model of an autonomy plan execution engine which allows for detailed analysis of mission operation, and (3) an environment model where external events may occur in a non-deterministic way and stimulate a response from the system model. As an example, the technique is used to model the software of a UAS and calculate the worst-case delay between arrival of various stimuli and corresponding responses from the system.

Timing Analysis for UAS Application Software

Charles Hartsell and Gabor Karsai
Institute for Software-Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
Email:{charles.a.hartsell, gabor.karsai}@vanderbilt.edu

Michael Lowry
NASA ARC, Moffett Field, CA 94035, USA
Email: Michael.R.Lowry@nasa.gov

3.1 Introduction

As Unmanned Aerial Systems (UAS) are becoming ubiquitous software development for such autonomous vehicles is facing significant challenges. Classic software development standards for airborne systems, DO-178 in particular, do not meet the needs of systems that are programmed on a per-mission basis, often under severe project schedule constraints. We envision that UAS will be highly customizable, and will support a large number of missions - where the mission software is often built on a 'when-needed' basis.

When such flight software is developed, the timely verification of the system is of great importance. Safety and mission reliability necessitate a comprehensive verification that is supported by solid engineering tools. The verification process should check the entire software application suite providing the autonomy, including flight control, vehicle management, mission management, and their interactions. Some of these applications may have been pre-verified (e.g. the autopilot), but typical mission-specific software (e.g. the software implementing the mission plan) is custom for each mission. Additionally, some of the mission-specific software could be implemented in declarative languages where the time-deterministic execution is not ensured. We consider component-based flight software systems in which software is built from reusable components with well-defined interfaces. Components are composed using a suitable composition platform - a middleware that facilitates component interactions. Systems composed of such building blocks, whose individual behavior is known, need to be verified at the integrated system level.

To support the above verification goals, we have a developed an approach to the rapid timing verification of UAS software suite. The specific verification goal was stated as follows: We need to verify that the software system (potentially consisting of multiple, concurrent applications running on a software platform) will respond to all stimuli within a bounded amount of time, under all foreseeable scenarios. Furthermore, we restrict our approach to one particular middleware and component framework: the Core Flight Executive/Core Flight System (cFE/cFS) from NASA (Wilmot, 2005), explained in detail below. The choice of framework determined the component execution semantics, i.e. how they were scheduled on the processor, what methods were available for implementing interactions among the components, etc.

The rest of this paper is organized as follows. Section 3.2 presents a brief review and comparison of related research. Section 3.3 describes the cFE/cFS middleware, the Plan Execution Interchange Language (PLEXIL), and Colored Petri Nets (CPN). Section 3.4 discusses how the relevant components were modeled within CPN. Section 3.5 describes the state space reduction and analysis techniques used followed by a discussion of the obtained results in Section 3.6. Section 3.7 proposes possible extensions and improvements to this work followed by a conclusion in Section 3.8.

3.2 Related Research

Component-based system verification requires significant knowledge about the system in question commonly derived from the system design model, but these design models often fail to explicitly model the relevant real-time properties of a system. To address this shortcoming, real-time properties and composition semantics of a system have been explicitly described using model descriptors (Lopez et al., 2006). The real-time description can then be converted into a transactional model suitable for use with the MAST modeling and analysis framework (Harbour et al., 2001), (Pasaje et al., 2001). This allows for calculation and analysis of response times, blocking times, slack times, and statistical estimation of average system performance.

High-level Petri nets offer a foundation for a powerful and versatile set of modeling formalisms for concurrent and component-based systems. (Renault et al., 2009b) performed qualitative analysis of general-purpose Architecture Analysis and Design Language (AADL) (Feiler et al., 2005) models by translation to Symmetric nets, then to Timed Petri Nets for analysis of real-time properties (Renault et al., 2009a).

Several analysis approaches exploit and enhance existing verification techniques by presenting tool-aided methodologies based on these techniques. The UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) (Object Management Group, 2009) has been used to convert high-level design models into MAST models for schedulability analysis (Medina and Cuesta, 2011). AADL models have been translated into the real-time process algebra ACSR (Lee et al., 1994), allowing for schedulability analysis by searching the state space using the ACSR-based tool VERSA (Sokolsky et al., 2006). TIMES (Amnell et al., 2004) performs symbolic schedulability analysis by translating task sets into a network of timed automata describing task arrival patterns, then uses UPPAAL (Bengtsson et al., 1996) model checking to calculate worst-case response times.

Analysis of hierarchical component-based systems can be simplified by composing component interfaces that abstract the timing requirements of a component. (Easwaran et al., 2006) presents an algorithm for abstracting the resource requirements of underlying components in the form of periodic resource models with different periods. An operational period which minimizes the resource demands of the system can then be selected using these interfaces, thereby minimizing waste of system resources.

CPNs have been used to perform schedulability analysis of Airborne Mission Systems (Kristensen et al., 2002). This approach used a parametric CPN model allowing for different task sets to be configured and analyzed without major model changes, and assumes a cyclic executive which schedules tasks in a fixed major cycles subdivided into minor frames. The authors also assume that a schedule which satisfies a single major cycle can be repeated in all major cycles. Our approach improves upon this by creating a generic parameter-based application model which allows for entirely new apps to be analyzed without model modification. Additionally, we do not assume that all major cycles are identical and consider a more dynamic system capable of handling (1) high-level discrete control algorithms, (2) variable component execution times, and (3) environmental events.

3.3 Background

3.3.1 Core Flight System

The Core Flight System (cFS) (Wilmot, 2005) is a layered, modular software framework and is the architecture targeted for timing analysis. cFS was originally intended to reduce development time and cost of spacecraft flight software, but has been used on a variety of other embedded systems as well. cFS contains a core application environment and set of flight software services known as the core Flight Executive (cFE) as well as a set of reusable, flight-qualified applications (McComas, 2012) to promote code reuse. cFS allows developers to write applications, hereafter called "apps", which can be enabled and configured independent of other apps on the system. The cFE Software Bus (cFE SB) service and the cFS Scheduler application were of particular importance for determining system stimulus-to-response timing. The cFE SB provides an inter-application messaging service following a publish/subscribe model, while the scheduler application allows for periodic wake-up messages to be configured and sent to other applications running in the cFE framework. Additionally, the Operating System Abstraction Layer (OSAL) provides an OS independent API.

A diagram of the cFS architecture is shown in Figure 3.1 along with a particular app set considered. Apps shown as diamonds can be well represented with a generic model of a cFS application, discussed in Section 3.4, while apps shown as ovals were modeled individually. The generic apps FLT, HEAT, and CAM corresond to the Flight, Heater, and Camera controllers respectively while TRAC and BATT refer to state tracking and battery monitoring apps respectively. The functionality of each generic app is discussed in the System Configuration portion of Section 3.6. The apps modeled individually are the PLEXIL Executive (PLEXIL EXEC) and the cFS Scheduler (cFS SCH).

3.3.2 Plan Execution Interchange Language

The Plan Execution Interchange Language (PLEXIL) (Verma et al., 2006), (USRA, 2014) is a general plan execution language intended for use in a variety of autonomous applications. PLEXIL plans are executed within the PLEXIL Executive, which follows a set of formally defined execution semantics and can be run either standalone or integrated into a larger application framework such as cFS. The purpose of the PLEXIL

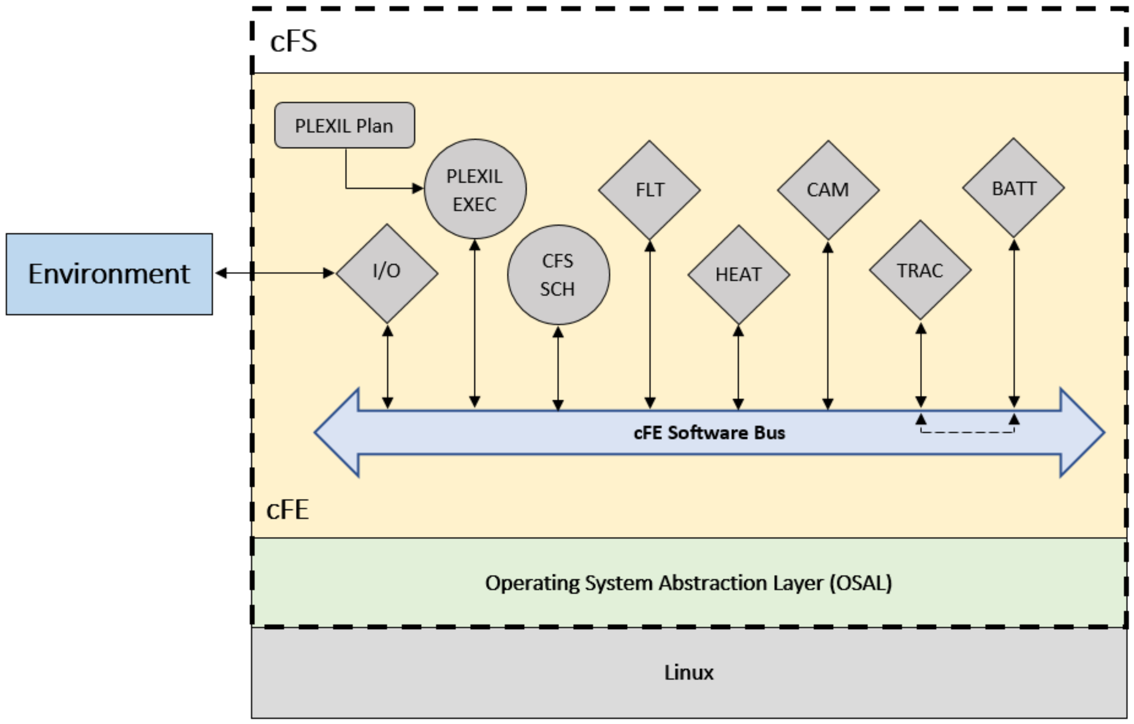


Figure 3.1: Diagram of cFS architecture.

Executive is to exercise overall control over the execution of applications. It can be considered as a high-level, script-based supervisory controller that reacts to external events and orchestrates the execution of other apps. A semantic analysis framework for the PLEXIL executive has been developed (Dowek et al., 2007), (Dowek et al., 2008) using the Prototype Verification System (PVS) (Owre et al., 1992) which allows for formal verification of different semantic variants of the PLEXIL Executive.

PLEXIL allows developers to write high-level control plans which consist of a hierarchical set of *nodes*. A node specifies some *action* to be performed. Each node is always in one and only one of the possible *states*, such as Waiting, Executing, or Finished. Every node has a set of *conditions* that govern when and how a node transitions between states. When a node meets the specified conditions and transitions to the Executing state, the corresponding actions are performed.

3.3.3 Colored Petri Nets

Petri Nets (Murata, 1989) are a graphical modeling language for modeling discrete, dynamic systems. They consist of *places*, *transitions*, *arcs*, and *tokens*. Places are containers which can hold a discrete number of units called *tokens*. Tokens contain no data beyond simply being present, and are often used to represent resources or available information. Arcs are a directional link between places and transitions. Transitions model state changes in the system and can legally fire when all of the corresponding input places contain the required number of tokens, defined by the arc *weight*. If multiple transitions are enabled at once, the firing order is chosen nondeterministically. Mathematically, a Petri net is a five-tuple (P, T, A, W, M_0) where P is the set of places, T is the set of transitions, A is the set of arcs, W is a function defining weights, and M_0 is the initial marking of the net.

Colored Petri Nets (CPN) (Jensen and Kristensen, 2009a) are a backwards compatible extension to Petri Nets which add the concept of *colors*. Colored tokens can contain data instead of simply being present or absent. What type of data a token contains is defined by an associated data structure, similar to C-style structs, called a *color set*. Transitions can read and modify the data stored in a token, and may define guard conditions based on the content of a token. Additionally, CPN allows for hierarchical description of nets making large, complex models more feasible.

Allowing tokens to contain arbitrarily complex data structures is a powerful modeling concept, and can be used to encode several properties of a particular system or component into a single token. This makes the resulting model very concise - an advantage of CPN over other high-level Petri Nets - and is one of our primary reasons for choosing CPN. Additionally, CPN Tools (Ratzer et al., 2003) provides software tools for design, simulation, and analysis of CPNs. However, use of arbitrary data structures often result in complex system models which are difficult to configure and maintain as the desired system configuration changes. Our model overcomes this limitation with a generic model of the cFS/cFE middleware that does not need to be reconstructed to suit each new configuration. Instead, the necessary system information is encoded into tokens which are loaded as the initial markings of several places within the CPN. Thus, each system configuration only requires a new set of initial markings, which are generated from high-level models of the apps and the particular PLEXIL plan(s) under consideration.

As an example, a simplified version of the CPN model used for generic cFS applications is shown in Figure 3.2. This model is configured as a hierarchical component (a "page") within the overall system model. Note that many of the transition and guard condition functions used are defined elsewhere for readability. The "CMD_PIPE" and "DATA_PIPE" places (top of figure) receive input from other components in the model, while the "App_Output" place (bottom of figure) serves as the output of the page. The desired apps are encoded into a data structure and loaded into the CPN model as the initial marking of the "APPS" place. Any messages an app is subscribed to are placed into the "CMD_PIPE" and "DATA_PIPE" places by the cFE software bus when they are published. Once the appropriate wakeup message is placed into the command pipe, the guard condition for transition "PROCESS_APP" evaluates to true and the transition becomes enabled. This transition will then fire, using the queued messages, system time, and parameterized app data as input. The transition evaluates the inputs, increments the system time in the "Clock" place by the app execution

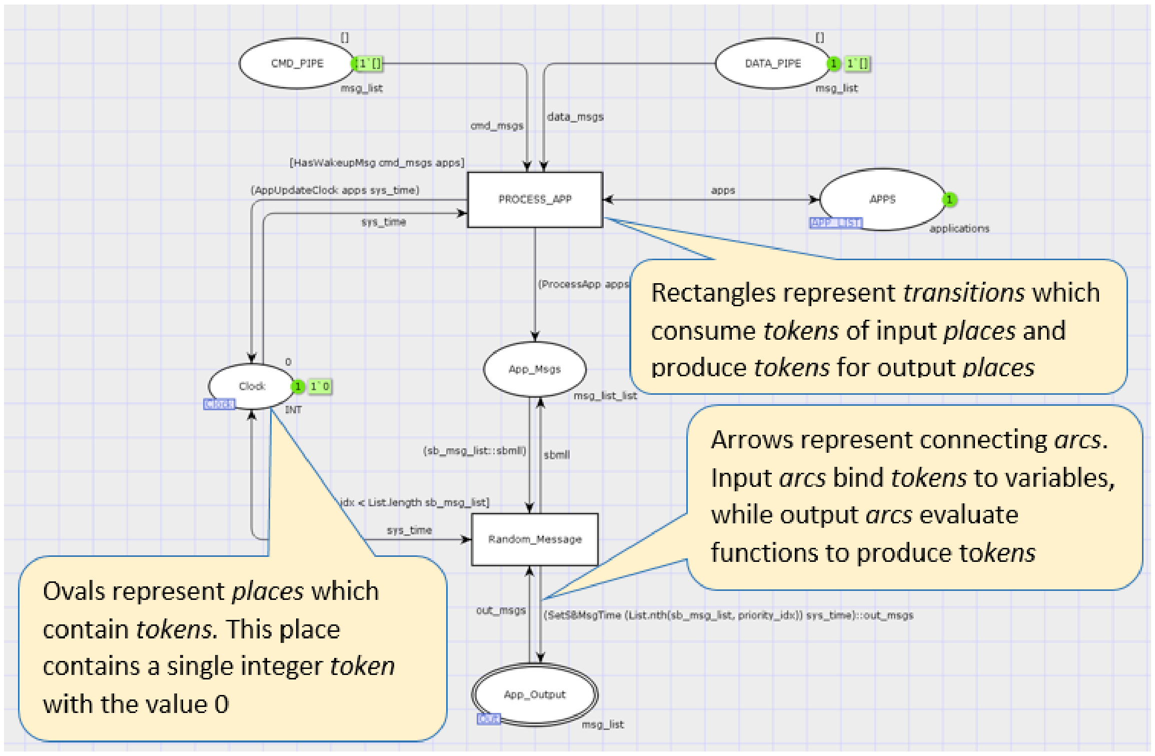

[]
CMD_PIPE
1`[]
msg_list
[]
DATA_PIPE
1
1`[]
msg_list
cmd_msgs
data_msgs
[HasWakeupMsg cmd_msgs apps]
(AppUpdateClock apps sys_time)
sys_time
PROCESS_APP
apps
APPS
1
applications
(ProcessApp apps
Rectangles represent *transitions* which consume *tokens* of input *places* and produce *tokens* for output *places*
0
Clock
1
1`0
INT
App_Msgs
msg_list_list
(sb_msg_list::sbmll)
sbmll
Arrows represent connecting *arcs*. Input *arcs* bind *tokens* to variables, while output *arcs* evaluate functions to produce *tokens*
idx < List.length sb_msg_list]
sys_time
Random_Message
out_msgs
(SetSBMsgTime (List.nth(sb_msg_list, priority_idx)) sys_time)::out_msgs
Ovals represent *places* which contain *tokens*. This place contains a single integer *token* with the value 0
App_Output
msg_list

Figure 3.2: Example CPN. Simplified model of generic cFS app.

time, and produces a set of possible output messages in the "App_Msgs" place. The "Random_Message" transition then makes a non-deterministic choice among these possible outputs and places messages into the "App_Output" place to be consumed by other components in the top-level model. Additionally, this nondeterministic choice of messages causes a split in the generated state space, discussed in more detail in Section 3.5.

3.4 Modeling

The cFE software bus, cFS scheduler app, generic cFS apps, and PLEXIL are the primary components of the system. The model assumes real-time scheduling semantics where the highest priority task ready executes until it either blocks or is preempted by a higher priority task. A diagram of the top-level CPN model is given in Figure 3.3. Each oval corresponds to a place in the model where parameterized system data is entered as an initial marking (with the exception of the cFE SB place). Each rectangle corresponds to a sub-page which models one or more components in overall system. All entities within the dashed rectangle are part of the cFS middleware, with the environment being the only external component. Each component is described in more detail below.

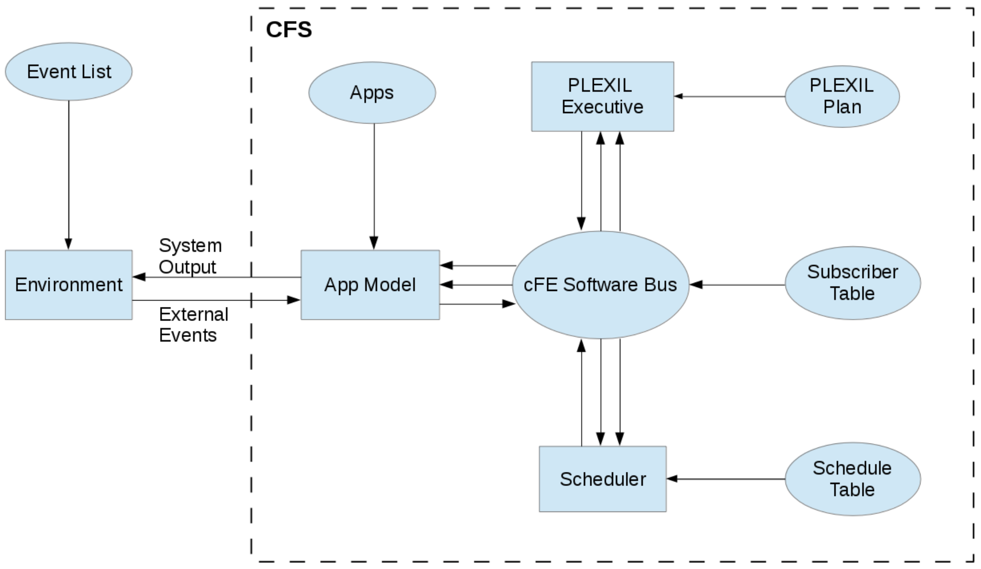


Figure 3.3: Top-level diagram of CPN Model. Double arrows from cFE software bus to each app indicate multiple message pipes.

3.4.1 cFE Software Bus

Inter-app communication between cFS apps occurs primarily through message passing via the cFE software bus (cFE SB) following a publish/subscribe model. Messages on the software bus are identified by a message ID - unique to each type of message - that indicates the structure of the data contained in the message. Apps can subscribe to a particular message type using the associated message ID, and must direct the message to one or more FIFO message queues called *pipes*. The number of messages a pipe can hold is configurable, and any message that arrives after a pipe is full will be dropped.

A model of the cFE SB was created to match the above description. Messages are encoded in a generic data type with fields for message ID, publisher, system time, and a content list of unspecified length. Apps can publish messages by constructing an appropriate message token and placing it in a central software bus place of the CPN model. Message subscriptions are stored in a subscription table data type loaded as an initial marking of the CPN model. When the software bus place contains messages, a receiving transition evaluates both the list of messages and the subscription table, then outputs each message into the appropriate pipes. While cFS supports other methods of inter-application communication (such as shared memory), the cFE SB is by far the most common and is currently the only method supported by the CPN model. However, other methods of communication within cFS can be modeled using similar techniques and are a possible area of future work.

3.4.2 cFS Scheduler

The cFS Scheduler app is one of the flight-qualified reusable applications provided in the cFS suite responsible for providing wake-up messages to other apps at fixed frequencies. The scheduler operates with configurable major frame and minor frame timing windows, with the major frame synchronized to a cFE system wide 1 Hz message. Apps can register an entry in the scheduler table to request a wakeup message be sent at a specified frequency. Note that the scheduler app can provide periodic messages used to unblock a pending app, but the underlying OS is responsible for the scheduling of thread execution on the processor.

The scheduler app is modeled as a single transition in CPN and cannot be preempted by other apps. This is reasonable for most cFS based systems, as the scheduler is typically configured as the highest priority app, preventing preemption by any other apps. However, the scheduler is not required to be the highest priority app, and if a it were configured at a lower priority preemption would need to be added to the scheduler model. Wake up message frequency for each app is encoded into a schedule table data structure and loaded into the model as an initial marking. The scheduler transition takes input from this schedule table as well as the system clock and produces wakeup messages at the specified frequency.

3.4.3 Generic cFS Applications

Typical cFS apps follow general execution guidelines and can be represented by a generic model. However, in general cFS apps are not strictly constrained to follow this execution structure, and may be outside of the scope of a generic model. A normal execution cycle begins when an app receives a wake-up message from the cFS scheduler. The app then processes all messages which have arrived in its message pipes since the last execution cycle ended, with each subscribed message type having an associated message handler. An app may also perform other actions before or after processing the message queue, and can produce messages either periodically or when triggered by one of the message handlers. Once the app finishes processing, it blocks on a wake-up message and the execution cycle is done. Additionally, apps are not restricted to be awoken only by the scheduler app, but can be triggered by other events (such as a message from another app) as well. A generic CPN model of a cFS app was created based on the typical execution cycle with the following assumptions:

1. Each app has a fixed OS-level priority.
2. Apps produce output only at the end of a cycle.
3. Each app executes within its own single execution thread.
4. All inter-app communication occurs via messages through the cFE SB.

5. Apps are awoken by wake-up messages on the cFE SB. These messages may be both time and event triggered.

6. Each app has a constant baseline Worst Case Execution Time (WCET). Each message handler and periodic message also has a constant WCET.

7. Deterministic control logic for handling messages is abstracted away. Modeled by a non-deterministic choice between possible responses.

8. Apps can generate periodic messages at a fixed rate. Message content determined by non-deterministic choice, as before.

9. Each app has two message pipes - command and data.

Apps which conform to these assumptions can be modeled using the current generic app model. Otherwise, either a unique model of the app must be created on a case-by-case basis, or the generic app model must be expanded. The current generic model is relatively simplistic and whether or not an app can be modeled generically is largely dependent on the level of fidelity desired. For instance, the cFS suite includes a limit checker app which monitors when telemetry data violates predefined thresholds. Basic interactions between the limit checker app and other generic apps could be modeled generically. However, if the given PLEXIL autonomy plan made decisions based on outputs from the limit checker app, a custom model would likely be needed for sufficient fidelity.

Non real-time applications which run during periods of inactivity in the system are currently not modeled. Since these apps only execute when no other app is ready, they do not affect the timing analysis results of the real-time apps. However, the model could be expanded for analysis of such best-effort apps if desired.

Several of the apps we have considered are responsible for executing commands issued by PLEXIL. These apps typically make a nondeterministic choice between command success and failure. This is a simple way of simulating commands that may fail to execute for reasons that cannot be known at design-time (ex. faulty actuator/sensor).

3.4.4 PLEXIL Executive

The PLEXIL executive is used as the autonomy engine responsible for reacting to external events and supervising system behavior. PLEXIL responds to external stimuli by issuing commands, often executed by other cFS apps, as well as modifying its internal state as appropriate. Unlike other generic cFS apps, the PLEXIL executive contains complex decision logic crucial to its supervisor responsibilities which cannot be abstracted away as nondeterministic choices. Additionally, PLEXIL plans may change on a per-mission basis, so generic plans need to be able to be simulated without reconstructing the model. Similar to the parameterized apps, generic plans can be encoded into a complex data structure and loaded into the CPN model as an initial marking. Since the PLEXIL executive does meet the assumptions placed on generic cFS apps, a custom model was constructed which implements the core functionality of the PLEXIL executive. While PLEXIL plans are deterministic given an identical sequence of events, the constructed model also implements a resource arbitration function when a command is issued. PLEXIL allows for resource requirements to be specified in a plan, and may refuse to issue a command if the associated resource is not available. This is modeled as a nondeterministic choice between issuing or denying a command, and results in a branch in the state space whenever a command is issued.

3.4.5 Environment

An environment model was created to produce external events for the system. The environment is represented by a sub-page within the CPN model which reads from a predetermined list of events. Each event consists of a list of messages (typically containing events such as sensor updates) and an associated trigger time. When the system clock reaches the trigger time of an event, transitions within the environment page become enabled and the event is sent to the system as well as stored in a log of input events. In particular, output from the environment model is sent to a dedicated external input pipe within the generic cFS app model, and any of the cFS apps are free to read from this pipe. Additionally, any app can send output messages to the environment model by placing messages into the external output pipe. The environment will then consume these messages and store the relevant data into an output event log. In the configurations we have considered all interaction with the environment occurs through a dedicated external I/O manager app, which is itself modeled as a generic cFS app. The list of triggering events (i.e. stimuli) is currently generated by hand with assistance from an external tool. However, another CPN model that simulates more dynamic (possibly nondeterministic) sensor behaviors could be constructed for this purpose, and such a model is breifly discussed in Section 3.7.

3.5 Analysis Techniques

System analysis using CPN models consists of generating a state space from a model, then searching this state space for events of interest. The state space is a type of directed graph data structure containing a tree of all reachable states of the model with connecting arcs for all possible transitions. Following a particular path through the graph represents one possible execution sequence of the system, and branches in the tree indicate nondeterministic execution. The complete state space of a system is infinite since time may increment indefinitely, so a maximum time is imposed on the CPN model to obtain a finite state space. For system verification, this maximum time should be sufficient to allow all relevant message propagation between apps to occur. For example, consider a system where 2 applications exchange a sequence of messages, then may produce some stimulus at the end of the exchange. The maximum time for this system should at minimum be the amount of time required for this exchange to complete and the resulting stimulus to be handled. We discuss results from such a system later in this section.

The size of a state space grows exponentially with the number of branches, leading to the known problem of state space explosion. This effectively limits the amount of non-determinism which can be allowed in the system model without requiring prohibitive generation and analysis times. In order for state space analysis to be feasible, steps must be taken to minimize state space size and avoid this explosion. We have applied methods of handling time and model structure reduction which have been shown to significantly reduce state space size (Kumar and Karsai, 2015). These methods are described briefly below.

3.5.1 State Space Reduction Techniques

3.5.1.1 Handling Time

System time was modeled explicitly as an integer value with its own "Clock" place in the CPN model. This approach offers some advantages over implicit techniques such as Timed Petri Nets where each transition has an associated execution time. First, explicitly modeled time can be more precisely controlled. With Timed Petri Nets, each transition has an associated delay which causes time to be advanced by a fixed amount when a transition fires. This is insufficient for cases such as preemption where a task, modeled as a transition, may not finish execution before being interrupted by another task. Second, explicit time can be progressed in a dynamic way such that the system skips past idle time, essentially fast-forwarding to relevant events. When no tasks are ready to run, the model checks the scheduler table and event list for the earliest time another event will occur, then jumps to that time. This time jump is safe under the assumption that no app can execute until a message triggering app wake-up is received. Dynamically advancing time in this manner greatly reduces state space size, which in turn reduces the time required for generation and analysis.

3.5.1.2 Structural Reduction

CPN modeling approaches often include a unique token for each individual item in a given CPN place (Kumar et al., 2014). These tokens are unordered and a particular sequence is chosen in a nondeterministic manner when the associated transition fires. This results in a branch in the state space, and causes exponential growth with the number of independent tokens in a place. To resolve this issue, all tokens in a place are grouped into a single ordered list element. When a new token is added to the place, it is either prepended or appended - depending on the desired semantics - to this list. Using this method, all elements in a place can be processed in the desired order. Additionally, all elements can be processed in a single step instead of one step per element. This is especially useful when finding one particular event in a set, as is the case when searching the scheduler table for the next app to wake-up. This reduction technique is used extensively throughout our CPN model.

3.5.2 State Space Analysis

Analysis is performed by defining query functions written in SML used to search the generated state space. Users may specify custom search functions to check domain-specific properties of a system as well. Stimulus to response timing is the primary metric used to evaluate the system and search functions have been developed for this purpose. This timing is found by searching the state space for all nodes where a stimulus is initially sent to the system as well as all nodes where a corresponding response is output by the system. Each node in the set of stimuli states is checked for reachability to every node in the set of response states. For all reachable combinations, the elapsed system time is calculated and compared to find the best and worst case stimulus to response timings. This process is repeated for all stimulus to response combinations of interest.

A second similar check is performed over the final system states to identify any execution sequence where the desired response never occured. Since the apps which handle commands are modeled with a nondeterministic choice of command success or failure, cases where a command is successfully issued but fails to execute are common. Unlike the stimulus to response queries, this query terminates as soon as one sequence without the desired response is found. For many of these cases, it may be necessary to construct an error handler PLEXIL node which performs some alternative action if the primary command fails to execute. An example of this is discussed in Section 3.6 below.

Due to the safety-critical nature of many real-time systems, timing estimates must be conservative. To achieve this, the model uses worst-case timing values for all components. Under the assumption that the provided WCETs are conservative for all components, the resulting best and worst case response timings will be pessimistic estimates suitable for system verification.

3.6 Results

3.6.1 System Configuration

As an example, we consider a cFS configuration with 8 apps. These apps are the cFS scheduler, PLEXIL, and 6 apps that conform to the generic application model. The first 3 generic apps - the flight, heater, and camera controllers - act primarily as command handlers responsible for executing commands issued by PLEXIL. Each of these apps handles between 1 and 3 commands with a nondeterministic choice between command success or failure. One generic app is an external I/O manager which acts as the interface between the cFE SB and the Environment. The last 2 generic apps - a system state tracker and a battery monitor - exchange

Table 3.1: Best and worst case stimulus to response timings.

Stimulus	Response	Worst Case Time (μs)	Best Case Time (μs)	Unhandled Sequences Exist
Clearance Received	Takeoff	406,330	655,100	Yes
Waypoint Reached	Fly to Next Waypoint	506,410	755,200	Yes
Low Battery	Prepare to Land	999,840	1,251,200	Yes
Target in View	Take Pancam	953,300	955,200	Yes
Low Temperature	Activate Heater (Primary)	753,300	755,100	Yes
Low Temperature	Decrease Altitude (Backup)	1,005,540	1,255,100	Yes

messages with one another and produce inputs for PLEXIL which may include a low battery warning. This represents interconnected apps with decision logic that depends on output from other apps. The particular contents and timing of the input to PLEXIL depends on several nondeterministic choices during message exchanges.

The chosen PLEXIL plan implements a simple autonomous drone and contains 16 nodes. This plan is loosely based off the "DriveToTarget" example plan included in the standard PLEXIL distribution (USRA, 2014). While many different sets of external events have been tested, the results presented here consider a set containing 6 independent events, each with a fixed firing time. These external events are the primary stimuli considered during analysis, along with some cFS internal events.

3.6.2 Timing Results

A state space was generated from initial loading of the PLEXIL plan at simulation time t = 0 until time t = 4,000 ms, with the highest frequency app running with a period of 250 ms for a total of 16 cycles. The resulting state space contained 443,685 states with 478,245 connecting arcs, and generation took approximately 26 minutes on a standard desktop-class computer. We expect a generation time of 26 minutes would be acceptable for a rapid verification tool. However, more realistic systems would likely be significantly more complex and require longer time horizons resulting in longer generation times. Further improvements will be required to keep state space generation and analysis times at a reasonable level, and some such performance improvements are discussed in Section 3.7.

The results presented in Table 3.1 show the best and worst case delays between arrival of a stimulus and the corresponding response output. The final column indicates if there exists any sequence where no acceptable response occurred. All of the listed stimuli are handled by 1 or more PLEXIL nodes, which then issue a command to be executed by other apps. The Clearance Received, Waypoint Reached, Target in View, and Low Temperature events are all generated by the environment, while the low battery stimulus is triggered by the 2 message exchanging apps within cFS. Note that the Low Temperature stimulus has both a primary

(Activate Heater) and backup (Decrease Altitude) response, where the backup command is only executed if the primary command fails.

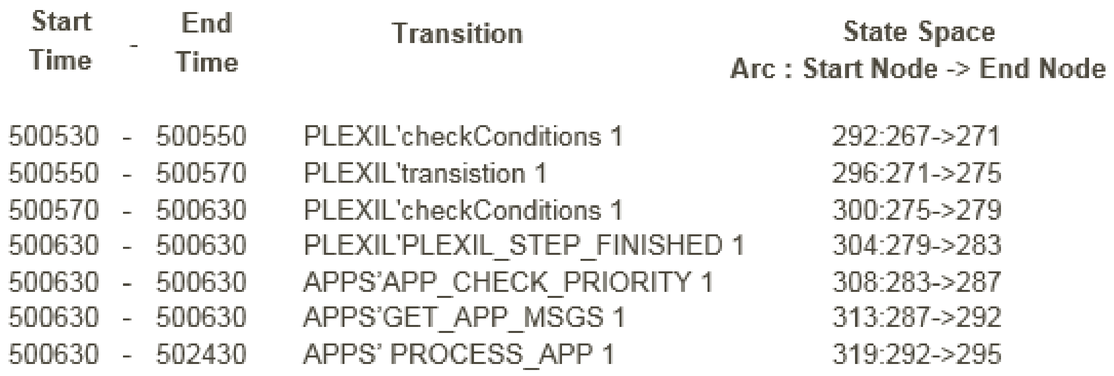
Stimulus to response time is taken from the time a stimulus is generated by the environment (or appears on the cFE software bus for internal events) until a corresponding response is output by the external I/O manager app. This means events must wait to be handled by the I/O manager at least once and most often twice. Since the I/O manager runs at a fixed frequency, messages are often delayed in the message pipe waiting for the next execution cycle. This has the effect of masking much of the internal nondeterminism, thereby reducing the amount of timing variation seen. However, this reduced variation comes at the cost of increased average response time. The Target in View and Low Temperature stimuli most clearly show this reduced variation. The Low Battery event only waits on the I/O manager when exiting the system, so the determinism in when the event occurs is not masked causing increased variation. The variance of the remaining responses is due to the priority of the app responsible for executing these commands being equal to the priority of the I/O manager. This leads to a branch in the state space where the order of execution of these two apps is nondeterministic, causing a large timing variation.

3.6.3 Unhandled Sequences

Most of the command handler apps were configured to make a nondeterministic choice between success or failure when executing a command. Therefore, unhandled execution sequences where the desired response failed to execute were expected for most stimuli. The backup action (Decrease Altitude) for the Low Temperature event was set to always succeed if PLEXIL issued the command, and it was expected that when the Low Temperature event occurred either the primary or backup action would always succeed. However, as indicated in Table 3.1, this was not the case. One of the PLEXIL nodes was responsible for issuing the backup command in case of a primary command failure, but this node did not consider the case where either the primary or backup command was denied by the PLEXIL resource arbitrator. In such a case, the command would never be issued to the system and would register a "denied" outcome instead of a "failed" outcome. This behavior was first observed during simulation of the CPN model, then confirmed by comparison of an unhandled sequence execution trace against the best-case execution trace. The execution traces generated by the tool are discussed in more detail below.

3.6.4 Execution Trace

An excerpt from the generated execution trace file is shown in Figure 3.4 with labels identifying each column. The start and end times correspond to the system clock before and after the corresponding transition fired. The exact location, name, and instance of the fired transition is specified in the following column. CPN Tools



Start Time	-	End Time	Transition	State Space Arc : Start Node -> End Node
500530	-	500550	PLEXIL'checkConditions 1	292:267->271
500550	-	500570	PLEXIL'transistion 1	296:271->275
500570	-	500630	PLEXIL'checkConditions 1	300:275->279
500630	-	500630	PLEXIL'PLEXIL_STEP_FINISHED 1	304:279->283
500630	-	500630	APPS'APP_CHECK_PRIORITY 1	308:283->287
500630	-	500630	APPS'GET_APP_MSGS 1	313:287->292
500630	-	502430	APPS' PROCESS_APP 1	319:292->295

Figure 3.4: Excerpt from generated execution sequence trace.

allows for entire pages (independent component of a model) to be replicated, thus requiring the instance of a transition to be specified. The final column indicates the identifier of the initial state (node), final state, and transition arc in the generated state space which corresponds to the fired transition. For example, the first entry in Figure 3.4 describes the transition "checkConditions" on the first instance of the PLEXIL page. This transition started in state 267 at time 500,530 and completed in state 271 at time 500,550 via arc 292. The trace file also includes a full listing of the content of all tokens involved during the firing of each transition, however this has been omitted for brevity.

When analysis results indicate a timing requirement has been violated, the trace provides a detailed counter example. Similarly, if results indicate that an execution sequence exists where an appropriate response was never received, another trace detailing one such execution sequence will be created. CPN Tools provides functionality to recreate a particular state space node within the model simulator, allowing the user to view the system state that led to a timing violation as well as step through the sequence one transition at a time. The trace is particularly useful for determining the cause of response timing variation, but currently this requires user inspection on a case-by-case basis. The root cause of a timing violation and potentially hazardous system states can often be identified by comparison between the best and worst case traces.

An external tool to assist this trace analysis by generating a PlantUML (Rosques, 2017) compatible UML State Machine Diagram visualization of the execution sequence has been developed. This tool clusters related steps in the execution trace into a single step with overall start and end time. For example, the top two entities (PLEXIL and APPS) in the visualization shown in Figure 3.5 correspond to the trace log file in Figure 3.4. The first 4 entries in the log file are all transitions within the PLEXIL component, and as such can be grouped into a single PLEXIL transition for visualization. The tool also parses the full bindings included in the log file to find any events of particular interest such as which apps were executed and which external events occurred.

These events are denoted to the right of their associated transition.

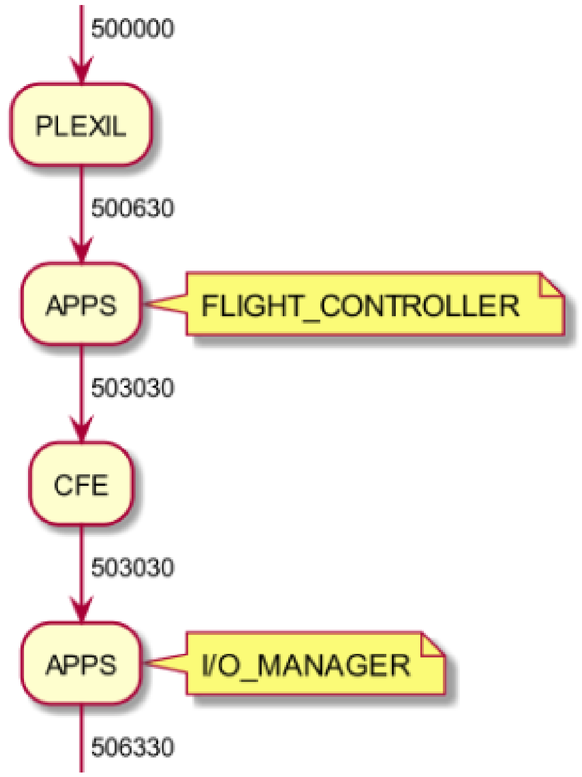


Figure 3.5: UML State Machine Diagram visualization of execution sequence trace.

3.6.5 Response Delay vs Event Timing

To examine the effect of variable event timing on the stimulus to response delay, we consider a simplified version of the described system configuration with fewer external events and a reduced maximum system time. The "Low Temperature" and "Activate Heater" stimulus/response pair was considered with time of occurrence varied between 0 and 900 ms. Response delay vs. event trigger time is shown in Figure 3.6, along with vertical markers indicating when the I/O manager and heater controller app (responsible for executing "Activate Heater" command) received wakeup messages. Note that the I/O manager is configured to run at twice the frequency of the heater controller app.

After an external event occurs, it must wait to be handled by the I/O manager on its next execution cycle before being placed onto the cFE SB. Therefore, the amount of time an event arrives before the start of the next I/O manager cycle is expected to cause a direct increase in both the best and worst case response delay, resulting in the saw-tooth pattern seen in Figure 3.6.

The difference between best and worst case delay is very small for most event timings. However, there is a large variation when an event occurs shortly after the I/O manager app becomes ready to run. This is due

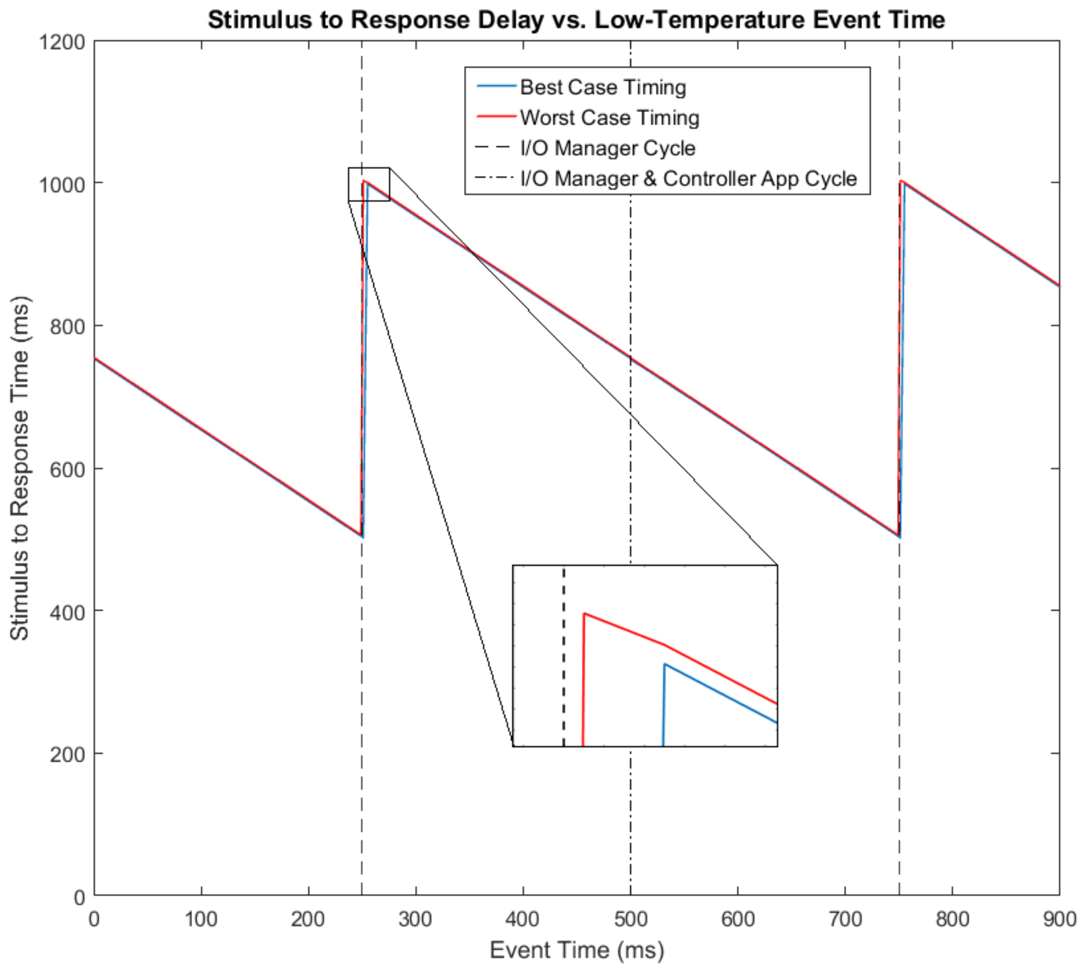


Figure 3.6: Best and Worst case response times for Low Temperature stimulus as a function of event occurrence time.

to the equal priority of the I/O Manager and one of the command handler apps. If the I/O manager is chosen to run first, the event will not be in the external data pipe when the app executes. In this case, the event must wait until the next cycle to be processed, resulting in a large worst case delay. If the command handler app is chosen to run first, then the event enters the external data pipe before the I/O manager executes leading to a much shorter best case delay.

3.7 Future Work

An appropriate CPN data structure for a given PLEXIL plan can be generated entirely automatically from the compiled plan files with the use of an external Python program. However, the same is not true for the cFS apps, which currently must be generated from architectural models by hand. We are working on a tool to automate this generation in order to further reduce the time required to reconfigure the CPN model for analysis of different system compositions.

Real systems often operate in highly variable environments where a wide range of events may occur at uncertain times and in uncertain quantities. Our current environment model does not adequately capture this behavior, and instead relies on a set of predefined events with well defined contents and times of occurrence. We are working on an improved environment model where bounds are placed on the contents and frequency of an event, then sent to the system at nondeterministic times. This would help determine the worst case stimulus to response timing under different external scenarios. However, fully nondeterministic events would quickly lead to state space explosion, making analysis infeasible. A nondeterministic environment model must choose optimum event times likely based on the boundary times around important transitions within the CPN model, similar to the saw-tooth pattern considered in Figure 3.6.

State space analysis currently takes far longer than necessary, and often takes more time than generation of the state space itself. This is primarily due to the large number of reachability checks performed in determining stimulus to response timings, as described in Section 3.5. Each reachability check requires a significant exploration of the state space, and the number of checks required grows proportional to the product of the sizes of the stimulus and response state sets. Since both of these sets tend to increase proportionally with the total size of the state space, the number of reachability checks required grows quadradically with the size of the state space. The current search algorithm is a naive approach which performs reachability checks on all combinations of stimulus and response states before determining the best and worst case timings, meaning that the average and worst cases require the same number of costly reachability checks. The number of checks and amount of time required in the average case could be greatly reduced by improvements to this algorithm. For instance, the optimal best and worst case timing values could be found without performing any reachability checks. These optimal cases could then be checked for reachability, and the search could end immediately if the check succeeded. Otherwise, the next most-optimal cases could be checked until a reachable pair is found. We have observed that at least one pairing in the most optimal set of best and worst case timings is often reachable, and therefore expect that this algorithm improvement could reduce the required number of reachability checks in the average case to scale linearly with the size of the state space. However, the worst case number of checks required would still scale quadratically.

The CPN Tools software used is currently single-threaded and could be parallelized to reduce state space generation and analysis time. Generation of the state space should scale well with parallelization since any state in state space contains all necessary information to generate all following states. Whenever a branch is enountered in the state space, each of the branched states can be tasked to a particular core and the remainder of that branch may be calculated independently of the other branches. State spaces for even relatively simple systems tend to contain a large number of branches and would likely see a significant, sub-linear speedup in generation time with increasing number of processor cores. State space analysis would likely see a similar speedup since the required rechability checks could also be done in parallel. Additionally, utilizing more advanced state space reduction and analysis techniques, such as those supported by the ASAP (Westergaard et al., 2009) analysis tool, could further reduce the time required.

3.8 Conclusions

In this paper we have shown how the behavior of a flight software system built in a component-based manner can be modeled using the CPN formalism. We considered a system with several components - a high-level controller application (PLEXIL), a suite of prototypical reactive applications, and the environment - composed using a specific middleware (cFE/cFS). The system models were constructed in a generic, parameterized fashion to allow for different component compositions to be considered without requiring model reconstruction. Timing analysis was performed using the coupled set of models to determine worst-case response timings of the integrated system. This analysis required knowledge of (1) the autonomy plan executed by PLEXIL, (2) the worst-case response times of the cFS apps, and (3) the set of environmental stimuli under consideration. Timing analysis consisted of generating the state-space of the system model, then executing queries against that computed state-space. The analysis lends itself to complete automation which, combined with the ability to consider new system configurations without reconstruction, indicates the suitability of the model for rapid verification of UAS flight software. Additionally, a number of potential improvements were identified including but not limited to (1) fully automated generation of the system model from an architectural model, (2) improving the environment model, and (3) further reducing the time taken for the state-space generation and analysis.

CHAPTER 4

ALC Toolchain

Foreword

A critical observation from many authors, as discussed in Chapter 2, is that effective safety assurance requires system safety to be a primary concern throughout all stages of the system life-cycle. However, the required effort for thorough safety analysis often results in these tasks being treated as a one-time activity near the end of the development cycle. Appropriate tool support with automation where possible is a necessity to enable this regular safety analysis without excessively increasing the required developer effort. There has been significant effort in this kind of tool support for general CPS development as discussed in Section 2.4. However, safety assurance is given relatively little consideration in the development of these tools. Additionally, while LECs have received significant research focus, their use in safety-critical CPS applications is only beginning to be explored. Safety assurance is a major challenge for this type of application and should be a central consideration in any supporting tool development.

Most of these tool suites make heavy use of MBE techniques, including the use of one or more DSMLs. Each engineering discipline utilizes their own DSMLs such as system architecture models for systems engineering, dynamics models for control engineering, or component models for software engineering. DSMLs often include associated tooling which automates common development tasks using information available in the models. However, CPS development challenges this traditional *separation of concerns* approach due to the high level of inter-dependency between the various domains. CPS methods and tools often fuse information from several heterogeneous modeling languages. This is particularly true for safety assurance since safety is an emergent property of a composed system and not something that can be fully analyzed for components in isolation. Integration and interoperability of many heterogeneous DSMLs must be a central consideration of any development environment for autonomous CPS.

As LECs are increasingly being used in these autonomous CPSs, special consideration should be given to LEC-specific design considerations. LECs shift much of the design emphasis from traditional specification and requirements refinement onto the data and ML algorithms used for training. In particular, maintaining *data provenance*, a complete history of data including its origin and transformations over time, is critical and even required by most emerging regulations and guidelines for these systems. Maintaining this provenance is a highly mechanical task which is prone to human error if done manually. Instead, development environments should automate this provenance tracking to the maximum extent possible.

The publication in this chapter presents the ALC Toolchain, an integrated development environment for autonomous CPSs with particular focus on safety assurance concerns and LEC-specific design challenges. A model-based methodology for design and development of LECs is presented for both *supervised* and *reinforcement* machine learning approaches. Special consideration is given to version control of system models and provenance tracking of all generated data, trained LECs, and verification artifacts. The integration of heterogeneous models and data objects within one environment allows for direct traceability from safety assurance arguments to relevant artifacts. Two example systems, a UUV and an AV, are used to demonstrate component development through the ALC Toolchain. Additionally, the toolchain serves as a prototyping platform for the assurance techniques described in Chapters 5 and 6.

Model-Based Design for CPS with Learning-Enabled Components

Charles Hartsell, Nagabhushan Mahadevan, Shreyas Ramakrishna,
Abhishek Dubey, Ted Bapty, Taylor Johnson,
Xenofon Koutsoukos, Janos Sztipanovits, and Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
charles.a.hartsell@vanderbilt.edu

4.1 Introduction

CPSs are often required to operate in highly uncertain environments, with significant degree of autonomy. For such systems, it is typically infeasible to explicitly design for all possible situations within the environment. CPS designers are increasingly using data-driven methods such as machine learning to overcome this limitation. LECs have demonstrated good performance for a variety of traditionally difficult tasks such as object detection and tracking (Held et al., 2016), robot path planning in urban environments (Sombolestan et al., 2018), and attack detection in smart power grids (Ozay et al., 2016). Some systems have even used end-to-end learning components for complex CPS tasks, such as the NVIDIA DAVE-2 (Bojarski et al., 2016) which used a Convolutional Neural Network (CNN) to map images from a front-facing camera directly to steering commands for an autonomous vehicle. However, component-based design where only selected functionality is implemented with LECs remains the more widely used approach.

Development of CPSs requires strong coordination between multiple engineering disciplines, challenging the traditional principle of "separation of concerns". Each discipline relies on their own specialized modeling languages and methods, typically supported by numerous software tools (e.g. simulators, analysis tools, CAD software, etc) with little to no support for the methods of other disciplines. Previous work with the OpenMETA tool suite (Sztipanovits et al., 2018) addressed this problem with the introduction of the CyPhyML model-integration language and supporting model and tool integration platforms. However, the platform focused on CPSs using conventional components based on analytical understanding of the domain. Such techniques do not account for epistemic uncertainties in the models of the physical components as well as their environments. Data-driven techniques, including machine learning methods, are a promising approach to account for these limitations. In this work, we consider how the concepts developed for the OpenMETA platform can be extended to learning-enabled systems and their assurance.

CPSs are commonly used in mission-critical or safety-critical applications which demand high reliability and strong assurance for safety. Assuring safety in these systems requires supporting evidence from testing data, formal verification, expert analysis, etc. Machine learning relies on inferring relationships from data instead of deriving them from analytical models, leading many systems employing LECs to rely almost entirely on testing results as the primary source of evidence. However, test data alone is generally insufficient for assurance of safety-critical systems. Techniques for formal verification of learning-enabled systems are an active area of research (Seshia and Sadigh, 2016)(Xiang et al., 2018a) and will need to be incorporated into the safety assurance of such systems.

Additionally, evidence used for safety assurance should be traceable and reproducible. Manual data management across the complex toolsuites often used for CPS development is a time consuming and error-prone process. This issue is even more pronounced for systems using LECs where training data and the resulting trained models must also be properly managed. Clearly, the task of maintaining traceability and reproducibility in all phases of the development cycle should be automatically handled by an appropriate development environment. In this paper, we introduce a model-driven design methodology for ALC and present the supporting development environment called the ALC Toolchain. Our approach combines multiple DSMLs to support various tasks including architectural modeling, experiment configuration/data generation, LEC training, performance evaluation, and system safety assurance. All generated artifacts - including system models, simulation data, trained networks, etc. - are stored and managed to allow for both traceability and reproducibility. Methods are provided for constructing static, design-time safety assurance arguments as well as dynamic, run-time assurance monitors. Certain artifacts, such as formal verification results and testing evaluation metrics, may be referenced as evidence in static system assurance arguments.

The rest of this paper is organized as follows. First, Section 4.2 discusses various related research efforts. Section 4.3 explains our methodology and the various modeling languages used to support it, followed by a description of the tool chain implementation in Section 4.4. Next, Section 4.5 provides illustrative examples of the methodology applied to multiple CPS platforms. Finally, Sections 4.6 and 4.7 discuss possible directions for future research and concluding remarks respectively.

4.2 Related Research

Various existing architectural description languages provide integration with analysis tools, such as the Architecture Analysis and Design Language (AADL) (Feiler et al., 2006). AADL is a standard for modeling and analyzing real-time embedded systems. Components in AADL may represent hardware, software, or system entities, and connections between components are used to model component interactions. AADL allows for analysis of various system properties including performance, schedulability and reliability. OMG's SysML (OMG, 2017) provides a general-purpose modeling language for systems engineering. SysML can be extended with the Modeling and Analysis of Real-time and Embedded systems (MARTE) profile for UML (Group et al., 2010) to allow for system analysis similar to AADL. These languages and tools offer strong support for general-purpose system development, but are not well equipped for many domain-specific tasks. In particular, they do not consider how data-driven techniques such as machine-learning may be integrated into the development of these systems.

DeepForge (Broll et al., 2018) is a machine learning development environment which aims to reduce the barriers to entry and development times for deep learning models. It provides a DSML with modeling concepts for describing neural network architectures and their training pipelines. Additionally, DeepForge uses a version control system to ensure traceability and reproducibility during the design of a deep learning model. The Google Colaboratory [1] is another interactive environment for machine-learning research, but provides a free-form environment to the user instead of following any predefined methodology. Similarly, OpenAI Gym (Brockman et al., 2016) provides another machine-learning toolset focused on reinforcement-learning techniques. However, these environments only consider the development of machine learning models, and do not address how these models may be incorporated into a larger development framework for CPS. In particular, they do not integrate LEC assurance and verification techniques which may be essential for systems used in safety-critical applications.

While machine learning offers several significant advantages over conventional design techniques, it also poses some unique challenges. In (Sculley et al., 2015), the authors consider the hidden costs of machine learning with the idea of *technical debt*. They identify several ways in which machine learning can incur significant costs for long-term maintenance of a system, and propose development techniques to mitigate these costs. Many of these techniques (eg. using appropriate levels of abstraction and careful management of data dependencies between components) can be enforced with an appropriate methodology and should be automated supporting tools.

Recent work has shown that many machine learning techniques are susceptible to adversarial examples where small input perturbations can cause the model to produce incorrect outputs with high confidence (Goodfellow et al., 2014). This is particularly troublesome for LECs used in safety-critical applications. The authors of (Pei et al., 2017) address this issue by introducing DeepXplore - a whitebox testing framework for deep learning models. DeepXplore provides an algorithm for generating test inputs which exercise the corner cases of a particular learning model. Also, the authors introduce the concept of "neuron coverage", analogous to branch coverage in traditional software testing, for systematically measuring the amount of a learning model exercised by a given test set. This approach significantly improves testing of machine-learning models, providing more confidence that the system will perform as expected. However, testing alone is not sufficient for safety-critical systems and will need to be supplemented with other safety assurance techniques.

The Model-based Demonstrator for Smart and Safe Cyber Physical Systems (MoDeS3) (Vörös et al., 2018) combined several model-based design languages and techniques for the development of a model railway system. The authors specified the system architecture with SysML block diagrams and the component

[1] colab.research.google.com

level behavior with the Gamma Statechart Composition Framework (Molnár et al., 2018). Code generation tools provided by the Gamma framework were used for implementation of the software components. Both design-time and run-time safety assurance techniques were applied including formal verification methods and run-time monitors. The authors successfully demonstrate the effectiveness of Model Based Systems Engineering (MBSE) for CPS. However, MoDeS3 does not consider the integration of machine learning with CPS. Additionally, MoDeS3 does not provide a unifying tool-chain for the MBSE methodology used.

Other projects have developed DSMLs specifically for machine learning applications in CPS. The authors of (Hartmann et al., 2017) advocate for a fine-grained learning approach which operates on small regions of a larger data set, as opposed to typical coarse-grained approaches which learn global behavior patterns from the complete data set. Their approach combines system properties learned from this fine-grained approach with properties derived from domain knowledge of the system, and the authors present a DSML derived from UML class diagrams for modeling this combination.

4.3 Methodology

Our methodology combines multiple DSMLs to support common tasks for CPS development including architectural modeling, experiment configuration/data generation, performance evaluation, and system safety assurance. Special considerations are made for CPS which use LECs with LEC training models for both supervised and reinforcement learning methods. The following sections describe each DSML and how they fit into the development workflow.

4.3.1 System Architecture Model

The CPS development workflow begins with high-level specification of the system architecture, and our approach supports a subset of the block diagram models from the SysML modeling standard for system architecture design. *Components* are abstract building blocks of the system architecture, and may be hierarchically composed into complete systems. Components are first defined in a *block library* before instances of a component can be created in a *system model*. This approach promotes reusability and maintainability of components across many system models. Component interfaces are defined using *ports* which can represent signal, power, or material flows. Software components use directional signal ports to model data flow between components, and each signal port produces or consumes a particular *message type*. Message types are similar to C-style data structures, and must be defined in a *message library*. Physical components use acausal power and/or material ports.

Components may contain one or more concrete *implementation alternatives* which fulfill the component's functional requirements. For example, a software object detection component may include one implemen-

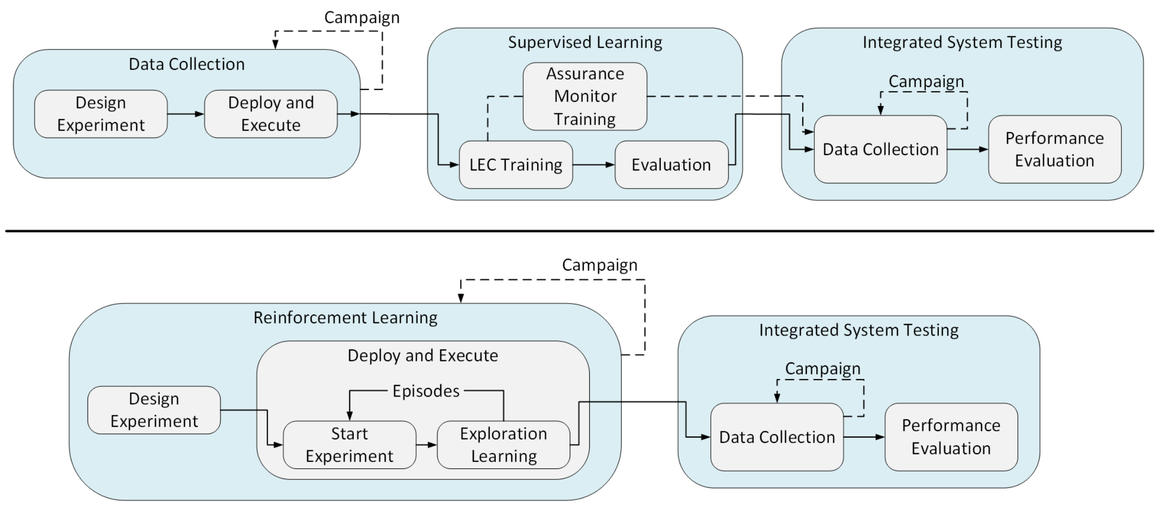


Figure 4.1: LEC Development workflows for supervised learning (Top) and reinforcement learning (Bottom).

tation based on conventional (analytical) methods and another based on machine-learning methods. For software components, implementation models contain the information necessary to load, configure, and initialize the software 'nodes' when the system is deployed as well as the business-logic associated with its runtime implementation.

4.3.2 Development Workflow Model

Once the system architecture is established, the developer executes one of two LEC development workflows – supervised or reinforcement learning - shown in Figure 4.1. Supervised learning begins with the collection of training data using one or more *experiment* or *campaign* models. In a *supervised learning* setup, an LEC model is trained against the collected data set, with the option to train run-time assurance monitors as well. Then, new *experiment* or *campaign* models may deploy the system with the trained LEC for integrated system testing and evaluation.

In a *reinforcement learning* setup, the LEC model is trained while the system or environment is being executed (or simulated). During training, the LEC model is updated based on the environment response (state and reward) to the input action. The training is repeated for a specified number of episodes, with each episode lasting a specified maximum time-limit or step-size. The trained LEC is evaluated by executing the reinforcement learning setup in a non-training mode.

4.3.3 Experiment Model

The experiment model captures all information necessary for configuration and execution of a system. First, the architecture model of the system under test is refined to an *assembly model* where one specific implementation from the available alternatives is selected. This is done for each component that includes multiple implementation alternatives. Execution of the assembly model requires additional parameters divided into three sets: environment, mission, and execution. *Environment parameters* define any environmental variables (terrain, weather conditions, etc.) and provide the files necessary for launching the simulation. *Mission parameters* define the objective for the system (eg. Track and follow a pipeline on the seafloor) and the relevant parameters to set up the experiment. *Execution parameters* define miscellaneous parameters for simulator management, data server configuration, etc.

An experiment model can be enriched with a *Campaign* model which configures iterations of a experiment. Campaigns include a *parameter sweep* block for varying system or environment parameters over a range of possible values. Campaigns are commonly used to tune system parameters for optimal performance, or to gather data in a variety of environments for LEC training.

4.3.4 Training LECs

Integration of model-based design with learning-enabled components is a focus of our methodology. Therefore, LEC training is an essential part of this methodology, and models for supervised (Murphy, 2012) and reinforcement (Sutton and Barto, 1998) learning are provided. Both techniques utilize the *LEC model* concept, which allows for learning architectures to be defined graphically or through code. Currently, the LEC model supports specification of deep neural networks.

The model for *supervised learning* setup includes the LEC model to be trained, the training data set(s), the training code, and any necessary hyper-parameters. The *training data* provides a reference to the desired data sets, which are typically results from previous experiments or campaigns. The *parameter* block captures hyper-parameters relevant to the specific training exercise (e.g. batch size, number of epochs, etc.).

Unlike supervised learning, reinforcement learning involves training the LEC online (i.e. interacting with the environment). Just as in the experiment model setup, the *reinforcement learning* model includes an *assembly model* where the specific component implementation(s) are selected from the possible alternates. The setup definition includes the LEC model to be trained, underlying reinforcement learning algorithm, associated rewards function for agent action given the current state of the system, and any training hyper-parameters.

4.3.5 Safety Assurance

Safety assurance is a critical component of any CPS which operates in mission-critical or safety-critical applications, but certification requirements and techniques vary between different regulating bodies. Safety cases are one such method of system certification which have been accepted by certain industries for years (eg.

UK Ministry of Defense (UK Ministry of Defense, 2007)). Safety cases are essentially structured arguments, often represented graphically, with supporting evidence that a particular system is acceptably safe. They have gained popularity in areas including CPS software development, and more regulating bodies have published guidelines and standards for their use (eg. Appendix D of FAA Unmanned Aircraft Systems Operational Approval document (Administration, 2013)).

Our methodology uses GSN (Kelly and Weaver, 2004) to allow for construction of safety cases. GSN is a language for representing structured arguments in a tree-like graphical form where primary *goals* are decomposed into logical combinations of *subgoals*, often in a hierarchical manner. Goals at the lowest level are represented by leaf nodes in the tree and are supported by *solution* nodes which provide evidence that the goal is satisfied. *Strategy* nodes can be used when composing multiple subgoals into a single, higher-level goal to provide a detailed explanation for the reasoning of the argument. Any assumptions made during the construction of a safety case can be explicitly stated with *Assumption* blocks, which may be attached to any node in an argument. A more comprehensive introduction to hierarchical safety cases and the GSN standard can be found in (Denney et al., 2013).

As part of our development of the Systems Engineering And Assurance Modeling (SEAM[2]) toolsuite, we extended the GSN models and integrated them with system architecture models to provide context to each branch of the assurance case argument. GSN models - integrated with system architecture and fault propagation models - were used to build assurance cases for radiation-reliability of CubeSat payloads with commercial off the shelf parts(Austin et al., 2018). The assurance arguments were grounded in Reliability and Maintainability standards (NASA-STD-8729.1 [3]) established by NASA's Office of Space and Mission Assurance (OSMA) for space flight systems (Groen et al., 2015).

The ALC toolchain builds upon the extensions to GSN from SEAM. GSN goals and solutions often address one particular component or subsystem. *Model reference* blocks provide a reference from a GSN node to a block in the system architecture model to make this relationship explicit. Similarly, solution nodes can provide a link to any supporting data - formal verification results, testing evaluation metrics, etc. - using *evidence source* blocks.

4.3.6 Verification and Run-Time Assurance

Currently, assurance of LEC-based systems is heavily reliant on testing results and such systems often require constant supervision from a human operator for acceptance (U.S. Department of Transportation, 2018). To supplement testing-based assurance, our methodology also supports new formal verification techniques (Xi-

[2]https://modelbasedassurance.org/
[3]https://standards.nasa.gov/standard/nasa/nasa-std-87291

ang et al., 2018b) as well as dynamic assurance monitors (Papadopoulos et al., 2011) and provides modelling concepts for both. However, there is a fundamental problem with LECs: the training set is finite and it may not capture all possible situations the system encounters at operation time. For such unknown situations the LEC may produce incorrect or unacceptable results – and the rest of the system may not even know that. Hence, the safety assurance of CPS with LECs is very problematic. One concept that might help to mitigate this situation is to use continuous monitoring on the LEC to its performance and indicate when the level of confidence in the output of the LEC is low – i.e. if the LEC does not perform as expected. This monitoring process is termed 'assurance monitoring', which oversees the LEC and gives a clear indication of problematic situations. These assurance monitor techniques are an active area of research, and a more detailed explanation can be found in (Papadopoulos, 2008). Once a problematic indication is given, a higher-level control loop – or a 'safety-controller' – may take over and perform a safe action (e.g. slow down the vehicle) to mitigate the lack of performance in the LEC. Note that the 'safety controller' must be independently designed and verified such that an overall safety assurance case can be constructed for the complete system, and executed in a way that allows hot-swapping.

4.4 Implementation

The physical architecture of our toolchain is shown in Figure 4.2, and is centered around the WebGME infrastructure (Maróti et al., 2014): a meta-programmable collaborative modeling environment which provides several advantages. First, the WebGME user interface is a web-based environment that can be accessed from most web browsers and allows for real-time collaboration between multiple users. WebGME supports a flexible meta-modeling paradigm that allows for development of customized DSMLs. Further, WebGME API provides support to write custom code for model visualization and interpretation. While WebGME allows for code to be executed within the browser, it is not intended for execution of computationally intensive tasks such as simulation or LEC training. Instead, these jobs are dispatched to execution servers equipped with appropriate hardware. All generated data resulting from execution of a job is uploaded to a central fileserver, with only the relevant metadata being returned to WebGME and stored in a version-controlled database. Each aspect of this architecture is discussed in more detail in the following sections.

4.4.1 Modeling Language

In WebGME, the meta-model for a DSML can be created from scratch or it can be built on top of a library of existing DSMLs. The ALC DSML borrows upon existing meta-model libraries: SEAM [2], DeepForge [4] and ROSMOD [5]. Our toolchain provides an integrated modeling framework that supports multiple models

[4]http://deepforge.org/
[5]https://cps-vo.org/group/ROSMOD

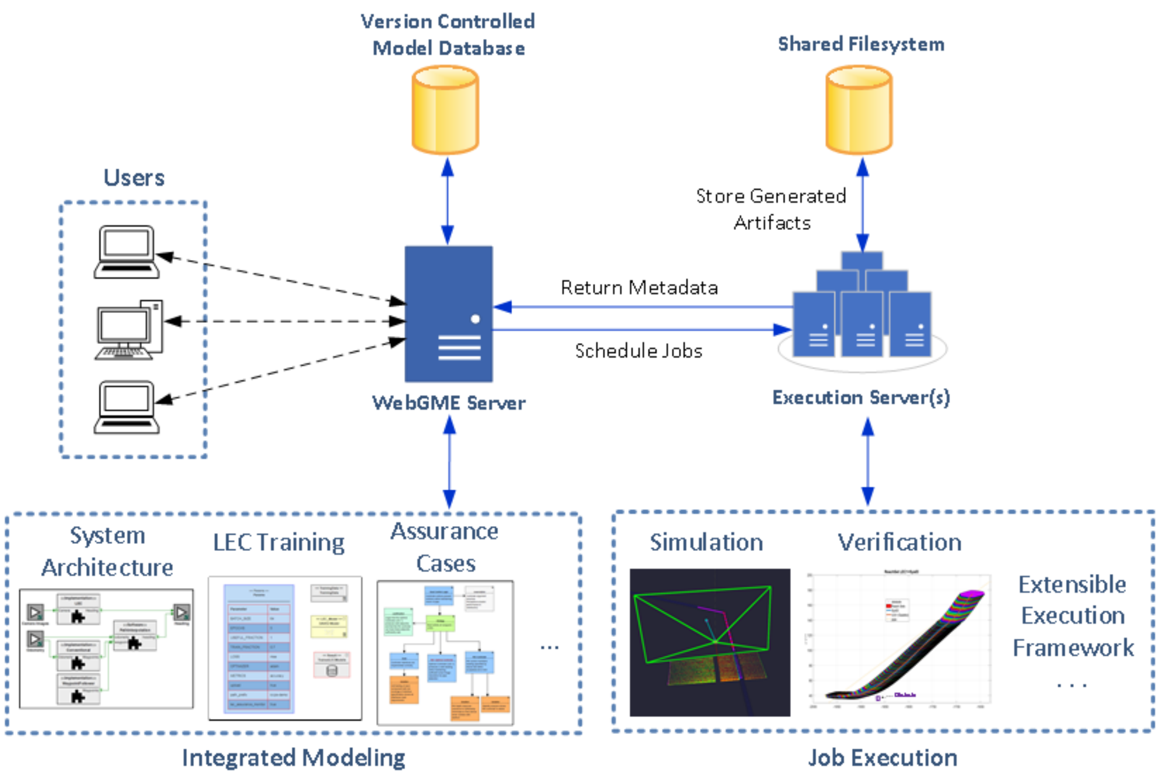


Figure 4.2: ALC Toolchain architecture overview.

including:

- A system architecture model based on SysML Internal Block Diagrams allows the user to describe the system architecture in terms of the underlying components (hierarchical blocks) and their interaction via signal, energy, and material flows.
- An experiment configuration model allows the users to configure execution instances for data collection, training of LECs with supervised or reinforcement learning, deployment and evaluation of trained LECs.
- Assurance case models based on GSN allows the user to create assurance arguments for the safety, performance, and reliability of the system.

The ALC toolchain uses the WebGME visualization and decorator framework to customize the visualization based on the model context. It borrows upon the WebGME CodeEditor to allow the users to develop and edit their code within the context of an ALC model.

4.4.2 Execution

The ALC toolchain uses the WebGME plugin and executor framework to launch execution instances on appropriate server machines (labeled "Execution Servers" in Figure 4.2). These execution instances could be a system execution (or simulation) for data collection or training exercises of LECs. Such simulation and training exercises are often computationally intense, and usually require Graphics Processing Units (GPU) or other forms of hardware acceleration. The ALC toolchain extends upon the DeepForge Pipeline model and its execution to allow remote deployment of computationally intense tasks on appropriately equipped servers. This enables developers of CPS to configure and launch computationally intensive simulation and training exercises on powerful machines from local web browsers, while collaborating with a distributed team of developers.

4.4.2.1 Interactive Execution and Debugging

The toolchain supports embedded Jupyter notebooks within the context of an experiment, training, or evaluation model. The toolchain can configure the code in the Jupyter notebook to execute the model. This allows users to launch their execution instances in an interactive manner and debug their code if required. Additionally, it allows users to write custom code to evaluate the system performance.

4.4.3 Data Management

Simulation environments often generate large amounts of data, which are needed for effective LEC training. In the ALC toolchain, large data sets (eg. simulation data and trained LEC models) are stored in a dedicated file server as shown in Figure 4.2. Currently, the implementation uses SSH File Transfer Protocol (SFTP) (Galbraith and Saarenmaa, 2006) with the standard ext4 Linux filesystem. Once a data set is uploaded, a corresponding metadata file is returned to the WebGME server and stored in the model. The metadata files provide enough information for retrieving a particular data set from the file-sever when needed for other tasks such as LEC training, performance evaluation, or LEC deployment.

When the experiment results are uploaded to the file-server, configuration files and other artifacts used to execute the experiments are stored with the generated data. This allows for the experiment to be repeated and for any generated data to be reproduced as needed. Additionally, this pattern of uploading the data to a dedicated server and only storing the corresponding meta-data in the model frees WebGME from handling large files and improves efficiency as well as model-scalability.

4.4.4 Traceability

Since we consider safety-critical systems, traceability and reproducibility at every step in the development process is a primary focus of our toolchain. WebGME provides a version control scheme similar to Git [6] where model updates are stored in a tree structure and assigned an SHA1 hash. For each update, only the differences between the current state and the previous state of the model are stored in the tree. This allows for the model to be reverted to any previous state in the history by rolling back changes until the hash corresponding to the desired state is reached. For a more detailed explanation of this approach, see (Maróti et al., 2014).

4.4.5 Experiment Gyms

The ALC toolchain is intended for both simulated and real-world experiment environments, referred to as "gyms". Currently, we support three gyms using open-source simulation environments: UUV Simulator, CARLA, and TORCS. UUV Simulator [7] (Manhães et al., 2016) is an extension to the Gazebo [8] (Koenig and Howard, 2004) simulation environment which provides additional plugins for simulation of UUVs. CARLA [9] (Dosovitskiy et al., 2017a) is an automotive simulator intended for the development of autonomous vehicles. The Open Racing Car Simulator (TORCS) [10] (Wymann et al., 2000) is another automotive simulator which has been used in several research projects. Currently, all software components are implemented using the ROS [11] (Quigley et al., 2009) middleware.

Training and execution of LEC models is done using the Keras neural network library (Chollet, 2015) on top of the TensorFlow machine learning framework (Abadi et al., 2015). Additionally, Jupyter notebooks (Kluyver et al., 2016) have been integrated into the WebGME environment which allows users to interactively perform simulations, training, or performance evaluation.

4.5 Examples

4.5.1 Unmanned Underwater Vehicle

4.5.1.1 System Overview

As an example, we consider the design and implementation of a UUV controller tasked with following a pipeline on the seafloor. The controller is built on the ROS middleware, and experiments were performed using the Gazebo simulation environment with the open-source UUV Simulator extension packages. The ECA A9 vehicle provided with UUV Simulator was the chosen UUV. This vehicle is equipped with four control

[6]http://git-scm.com
[7]uuvsimulator.github.io
[8]www.gazebosim.org
[9]www.carla.org
[10]torcs.sourceforge.net
[11]www.ros.org

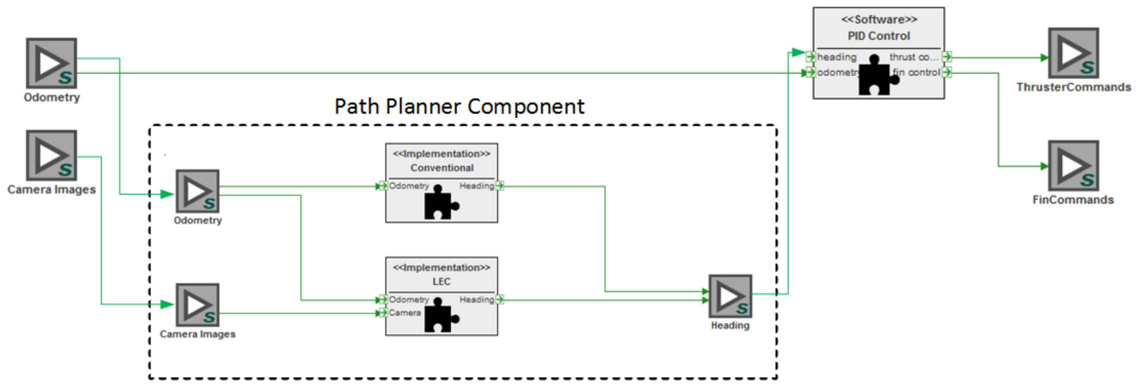


Figure 4.3: System architecture diagram of a UUV controller.

fins, one propeller/thruster, a forward-looking camera, and vehicle speed and position sensors. Additional sensors are available but were not used for this example.

The controller was required to produce all actuator commands necessary to follow the pipeline at a desired separation distance using the image stream from the camera and the vehicle odometry data as input. This task was further divided into two components: a path planner and a lower-level PID controller. The path planner component was responsible for determining a suitable heading for the vehicle to follow based on images from the camera. This heading is sent to the PID controller which then must produce commands for all four fins and the thruster. The desired heading was provided as a 3 dimensional vector with components for both pitch and yaw control of the vehicle. However, the PID controller used fixed setpoints for both depth and speed control of the vehicle, and the pitch component of the desired heading was discarded.

A block library of components was created to model vehicle sensors, actuators, and the controller. The controller block was composed from two component blocks representing the path planner and the PID controller. Each component was assigned an implementation with the ROS launch files necessary for configuring and deploying the component. The path planner component contains two implementations: an analytical planning algorithm and a Neural Network based solution. The analytical planner had access to ground truth information about the pipeline from the Gazebo simulator, and was used to generate training data for the LEC based implementation. Instances of the library blocks were then connected into a system model, and the UUV controller section of this model is shown in Figure 4.3. This figure shows an exploded view of the path planner component with both conventional and LEC implementation blocks.

4.5.1.2 Data Generation and Training

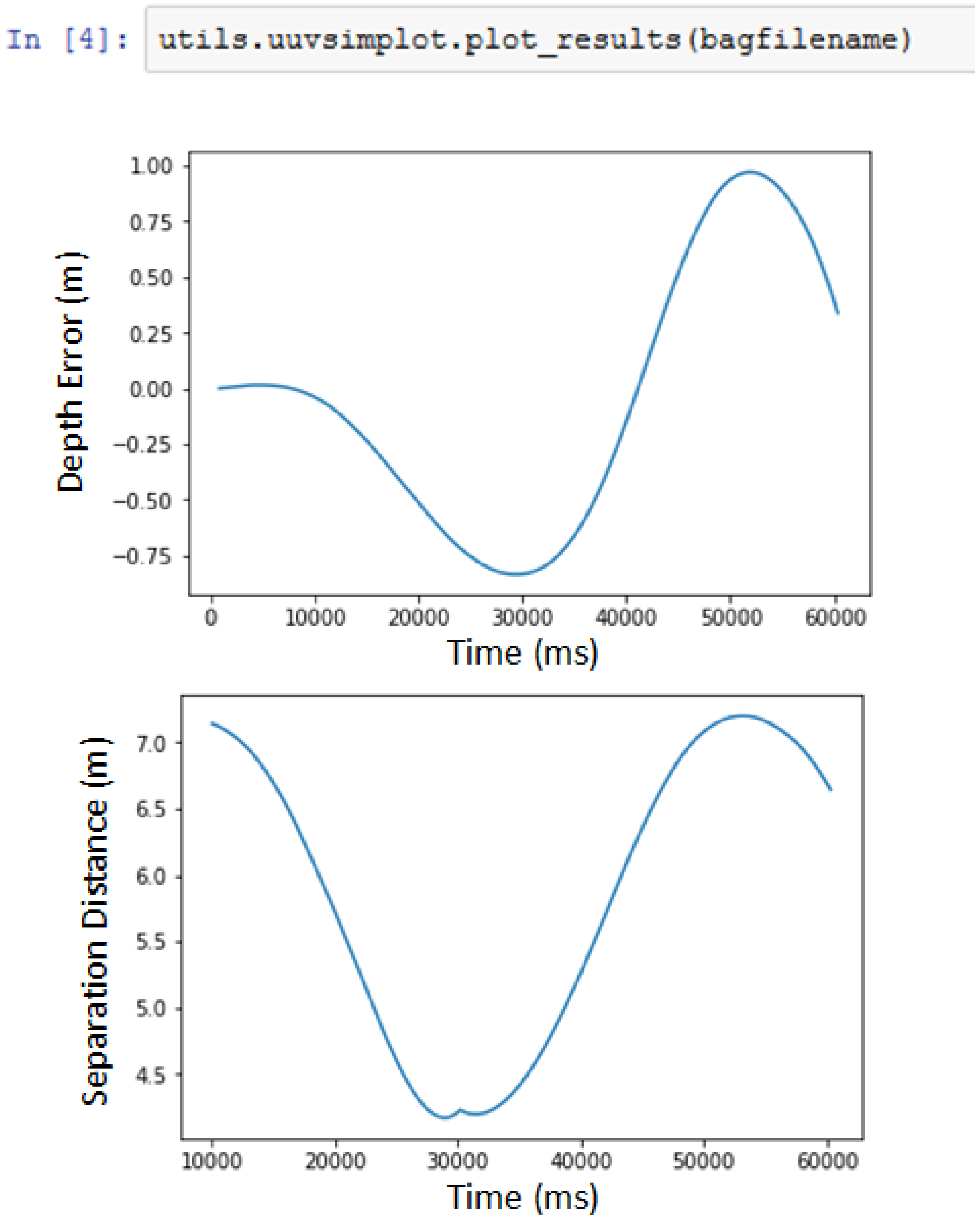
An experiment was configured using the UUV system model with the analytical implementation selected for the path planning component. For the experiments, the environment was set to a world model with a flat seabed and good water visibility. The experiment mission was for the UUV to follow a pipe on the seabed. In order to generate sufficient training data, campaign models were setup to run multiple iterations of the experiment, with varying pipe layouts. The qualities of the pipe (e.g. size or color) were not changed. Additionally, as part of the data generation campaign, each experiment was executed twice: first without disturbing the path of the vehicle, then with random noise added. This was done to gather additional training data for situations where the vehicle is in a non-optimal location relative to the pipe.

A supervised learning model was constructed for the LEC implementation of the path planner, and the data generated from the campaign was referenced for training data. Initially, a simple CNN architecture was chosen and trained. A second campaign model was created using the LEC based path planner for evaluation of the trained CNN. Multiple CNN architectures were trained and evaluated in an iterative process before finally selecting a modified version of the NVIDIA DAVE-2 (Bojarski et al., 2016) model. Additionally, training hyper parameters such as batch size and number of epochs were adjusted for optimal performance. Each CNN was evaluated based on several metrics including: error between the CNN predicted heading and the ideal heading from ground-truth data, how well the system maintained the desired separation distance from the pipe, and if the system successfully kept the pipe in view of the camera at all times. Jupyter notebook integration allows for interactive plotting and evaluation of system performance metrics using either provided utility functions, as shown in Figure 4.4, or custom python functions written by the user. The upper plot in Figure 4.4 shows the error between the desired depth and the actual depth of the UUV against time, while the lower plot shows the separation distance between the UUV and the pipeline during the same time range.

4.5.1.3 System Assurance

The primary safety goal for our UUV is to avoid collision with the submerged pipeline at all times. Additionally, the system should also keep the pipe in view of the camera while progressing at a set minimum speed. With this in mind, a GSN safety case was developed for the complete system which consisted of approximately 100 total blocks constructed hierarchically. However, due to space constraints, a simplified, single-level version of this argument is used as an example here and is shown in Figure 4.5. The primary goal of this argument has not changed, but only the software controller portion of the system is considered. Assumptions made in this argument are listed in the assumption block connected to the top-level goal. The top level goal is broken into two sub-goals: the neural network path planner outputs a safe heading, and possible dangerous behaviors have been identified and appropriately mitigated.

Solution blocks describe the evidence used to support each leaf goal, and may contain direct links to the evidence itself. For instance, the "Statistical analysis of system testing data ..." solution block shown in Figure 4.5 contains a reference to the neural network component in the system architecture model, as



```
In [4]: utils.uuvsimplot.plot_results(bagfilename)
```

Figure 4.4: Interactive plotting and evaluation of system performance metrics through Jupyter notebook. Depth error (upper) and separation distance (lower) plots are shown.

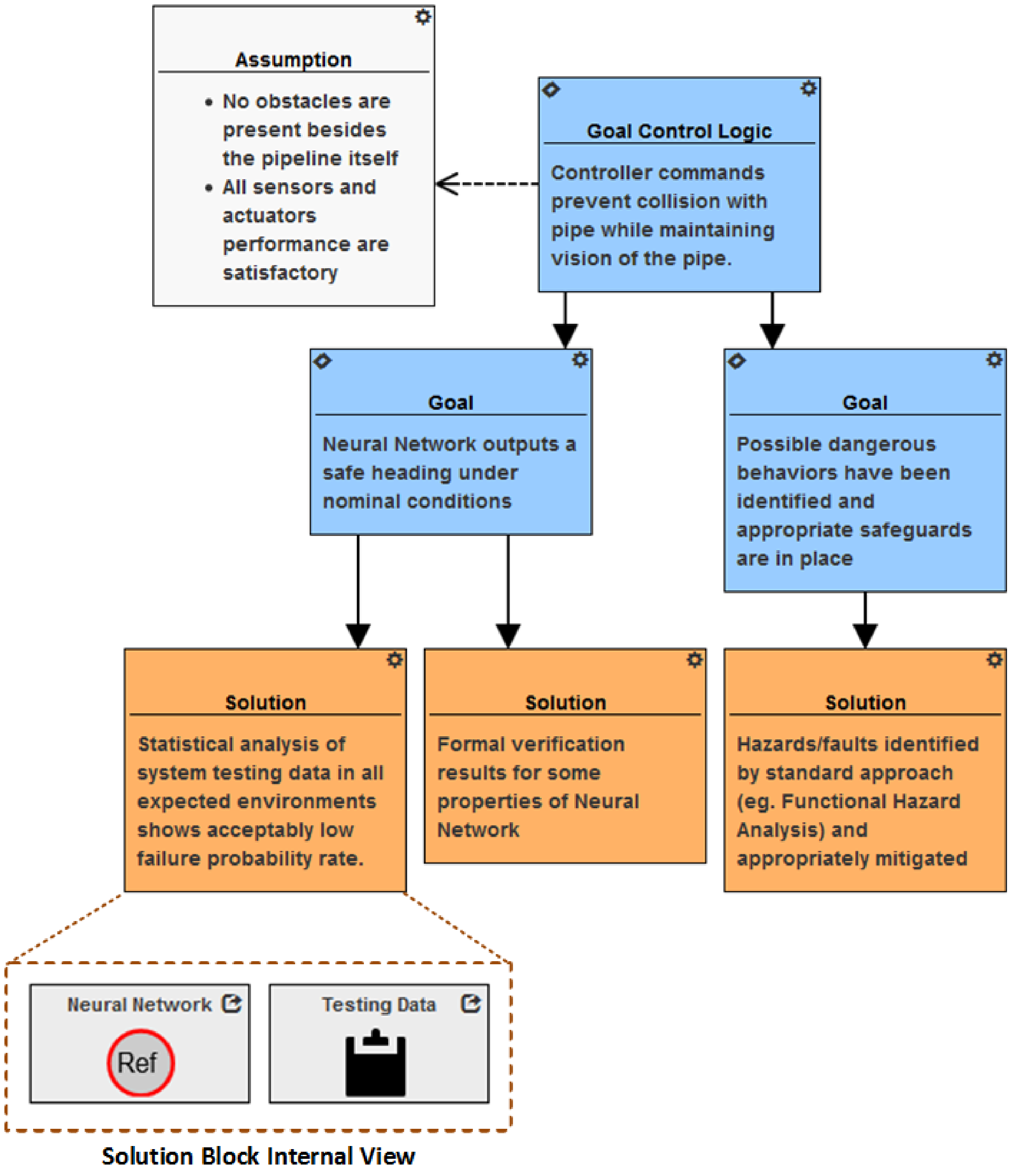

Assumption
No obstacles are present besides the pipeline itself
All sensors and actuators performance are satisfactory
Goal Control Logic
Controller commands prevent collision with pipe while maintaining vision of the pipe.
Goal
Neural Network outputs a safe heading under nominal conditions
Goal
Possible dangerous behaviors have been identified and appropriate safeguards are in place
Solution
Statistical analysis of system testing data in all expected environments shows acceptably low failure probability rate.
Solution
Formal verification results for some properties of Neural Network
Solution
Hazards/faults identified by standard approach (eg. Functional Hazard Analysis) and appropriately mitigated
Neural Network
Ref
Testing Data
Solution Block Internal View

Figure 4.5: Simplified safety case for UUV system with internal view of solution block shown.

well as a reference to the simulated testing data used for statistical analysis. Both of these references are traversable, which allows the user to quickly navigate to the relevant model artifacts. Formal verification techniques for Neural Networks are an active area of research, and the ALC toolchain is currently limited to reachable set estimation methods (Xiang et al., 2018b). While these methods are typically not sufficient for safety assurance of a complete system, they can provide guarantees for some system properties and should be used to supplement other assurance measures including system testing. Additionally, solution nodes may reference formal verification results as evidence, similar to the testing data reference block.

4.5.2 DeepNNCar

This example deals with an environment that includes the hardware in the loop. The testbed shown in Figure 4.6 includes a Traxxas Slash 2WD 1/10 Scale radio controlled car mounted with a Raspberry Pi (RPi) (on-board computing unit) and two sensors - a forward-facing RGB camera (30 FPS, resolution of 320x240) and an IR optocoupler which measures wheel RPM to compute vehicle speed. The overall goal is to drive the car autonomously around a track without violating any safety properties (e.g. do not cross track boundaries).

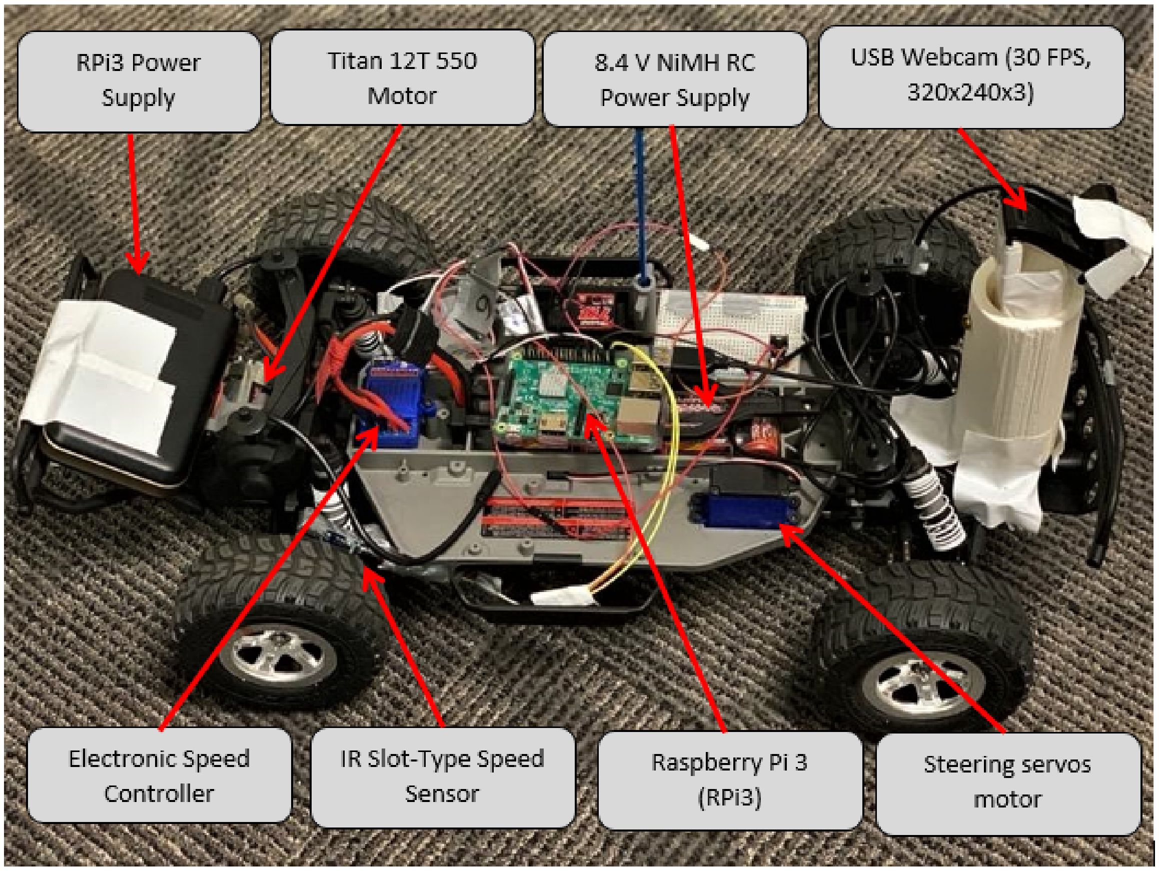

RPi3 Power Supply
Titan 12T 550 Motor
8.4 V NiMH RC Power Supply
USB Webcam (30 FPS, 320x240x3)
Electronic Speed Controller
IR Slot-Type Speed Sensor
Raspberry Pi 3 (RPi3)
Steering servos motor

Figure 4.6: DeepNNCar platform

4.5.2.1 System Architecture

The system architecture includes a LEC controller that uses a modified version of NVIDIA'S DAVE-II CNN model (Bojarski et al., 2016). The control architecture includes a lane-detection based safety controller that serves as a back-up to keep the system within the tracks. These controllers are part of a Simplex Architecture (SA) (Sha, 2001) with a decision manager that decides the actual control command to the vehicle.

4.5.2.2 LEC Training

The LEC controller was trained and validated using the supervised learning setup with the data collected from the testbed. Thereafter, the trained LEC was deployed and its performance evaluated on multiple tracks. A reinforcement learning setup was employed to train and learn the arbitration logic in the decision manager to meet the performance and safety conditions. The training employed the Q learning algorithm to learn the mapping from the states (current speed, weights for each controller output) to actions (change in speed and change in controller weights). In this setup, the reward value was computed based on the distance from the center of the track and the current speed.

4.5.2.3 Assurance

An assurance case for the safety and performance of the system was modeled in GSN. It takes into account the performance of the individual components - the LEC controller, the safety controller, the lane detection algorithm, and the decision manager. The evidence to the GSN arguments is based on the performance of each individual component as well as the integrated system. Additionally, we are looking into how these individual evidences can be combined to predict the safety and performance of the overall system.

4.6 Future Work

Our toolchain is intended to support both simulated and real-world environments. Currently, the toolchain has been integrated with gyms that are simulation environments. We plan to integrate with gyms or testbeds that correspond to physical systems that involve hardware-in-the-loop such as DeepNNCar so that experiment configuration, execution, data collection, training and evaluation could be automated through the tool-chain.

Both formal verification and run-time assurance methods for LECs are active areas of research with new techniques being rapidly developed. These new techniques should be incorporated to our tool-chain as they become available, and may require new modeling concepts to more tightly integrate with the overall system design.

Various techniques for quantitative evaluation of confidence in safety case arguments have been developed (e.g. using Bayesian Networks (Denney et al., 2011) or Dempster–Shafer theory (Cyra and Górski, 2011)) in an attempt to formalize certain aspects of the safety assurance process. Integrating these techniques with our methodology is another direction for future research.

4.7 Conclusion

Modern Cyber Physical Systems demand ever-increasing levels of autonomy while operating in highly uncertain environments. Conventional components are often insufficient for these systems due to the epistemic uncertainties present, leading many CPS developers to utilize Learning Enabled Components. These systems are often used in mission or safety critical applications where strong safety assurance is necessary. In this paper, we introduced a model-driven design methodology for Assurance-based Learning-enabled CPS which combines multiple DSMLs to support various tasks including architectural modeling, experiment configuration/data generation, system safety assurance, and LEC training, evaluation, and verification. A supporting development environment known as the ALC Toolchain allows for collaboration between multiple users while maintaining reproducibility and traceability during all stages of the development cycle. Additionally, the examples considered show how the complete methodology may be applied during the development of CPS using LECs.

CHAPTER 5

Assurance Case Construction

Foreword

Assurance cases have emerged as a useful technique not only for communicating the safety argument to stakeholders, but also as a tool for understanding what is required to achieve an acceptable level of system safety. However, assurance cases are non-trivial and require significant developer effort to construct. This often results in safety assurance tasks being delayed far too late in the development cycle where any problems identified can be very costly to address. Additionally, safety cases constructed after system design is complete often assume that the system is safe and the goal of the safety case is to demonstrate this safety. Ideally safety analysis would be started in the concept phase of system design, and the goal of the safety case should be to identify any problems the system

Reducing the effort required to construct safety cases is critical to encourage system developers to start safety analysis earlier in the system life cycle. Existing assurance case editors typically offer some level of productivity enhancements such as argument instantiation from a library of existing patterns. However, the construction process is still largely manual and dependent on the developer. Integration of assurance case construction with other MBE techniques is a powerful approach which can leverage existing system modeling practices to automate the construction process and significantly reduce required manual effort. However, system design information is often spread across many heterogeneous, interconnected DSMLs and effectively leveraging this information requires understanding the cross-domain relationships between these models.

The publication in this chapter presents a method for assurance case construction based on the instantiation and composition of existing patterns. This method automates the collection and organization of necessary information by extracting it directly from an existing set of interconnected system design models. To support human review and refinement, the generated assurance case maintains (1) traceability from objects in the argument back to the corresponding system models and (2) explainability of choices made during the construction process based on the relationships between model objects. As an example, the method is used to generate a partial assurance case for a UUV designed using the ALC Toolchain from Chapter 4.

Pre-publication copy.

Hartsell, C., Mahadevan, N., Dubey, A., & Karsai, G. (2021). Automated Method for Assurance Case Construction from System Design Models. Accepted for publication in *5th International Conference on System Reliability and Safety*.

Automated Method for Assurance Case Construction from System Design Models

Charles Hartsell, Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
charles.a.hartsell@vanderbilt.edu

5.1 Introduction

Modern CPSs are highly complex and successful design and development of these systems requires expertise across multiple engineering domains. MBE techniques are often used during development to separate the different areas of concern into more manageable pieces. Each engineering discipline often relies on DSMLs which are specialized for concerns of the discipline. This modular "divide-and-conquer" approach is powerful, but CPSs often have many inter-disciplinary interactions which cannot be ignored during system development. Instead, heterogeneous DSMLs are interconnected with appropriate cross-model links to capture these relationships between disciplines. This results in a rich set of models which contain a large amount of information, including safety related information, about the system being developed

Since these systems often operate in safety- or mission-critical applications, strong safety assurance is necessary for the system to be operationally deployed. Developers have traditionally relied heavily on regulatory agencies to provide guidance on the necessary level of safety assurance, particularly for those systems widely used in common society such as aircraft and automobiles. However, these regulations often struggle to keep pace with the rate of technological development, particularly in the realm of highly autonomous systems. This has caused a shift from a highly prescriptive, regulation-based approach to safety to more adaptable, goal-focused approaches. While these more flexible approaches are better able to keep pace with new technologies, they require significant effort from the developer to present a comprehensive and compelling argument for system safety.

Engineering arguments present a logical explanation of how individual pieces of evidence can be composed to provide support for system goals such as safety, security, or performance. Assurance cases have emerged as a useful tool for constructing these arguments and have been adopted in a number of industries including nuclear energy (IAEA, 2019), medical devices (US Food and Drug Administration and others, 2010), and aerospace (Dezfuli et al., 2011) among others. Moreover, assurance case patterns allow for common fragments of successful arguments to be abstracted and reused in later assurance cases. These patterns are a type of design pattern (Gamma et al., 1993) applied to the engineering argumentation domain and capture commonly repeated argument structures by replacing concrete, system-specific details with abstract parameters which can be instantiated later with details from any number of other systems.

Capturing and reusing argument structures through patterns offers several benefits including reducing the effort required to construct new assurance cases and avoiding repetition of mistakes made during the initial pattern development. However, construction of new assurance cases, even when patterns are used, is still a largely manual process which is both costly and error-prone, particularly given the scale and complexity of modern CPS. Manually constructed assurance cases rarely maintain direct traceability to system design artifacts or provide sufficient explanation for these artifacts, making review of the arguments difficult. Additionally, the assurance case should ideally evolve as the system design matures, but the high cost of manual construction and review leads safety assurance to be an afterthought of the design process.

Our Contribution: We introduce an automated process of assurance case construction based on the instantiation and composition of patterns into a larger assurance case. The information required to instantiate each pattern is sourced from existing system models spread across multiple heterogeneous modeling languages with links between the models describing cross-model relationships. This process is driven by three underlying techniques for: (1) traversing the interconnected models to identify model objects which match the requirements for pattern instantiation, (2) building a graph of assigned parameters including their relationships with other parameters, and (3) identifying patterns which can be sequentially composed with unsupported goals in an existing assurance case. The technique provides significant automation, but key decision points remain where the developer must choose from a pruned set of candidate model objects.

Outline: The remainder of this paper is organized as follows. First, Section 5.2 provides additional background on assurance case patterns and model-based development techniques. Next, our assurance case construction process is detailed in Section 5.3 followed by an illustrative example application in Section 5.4. Finally, we examine related research on automated instantiation of patterns and construction of assurance cases in Section 5.5, then conclude in Section 5.6.

5.2 Background

5.2.1 Assurance Case Patterns

An assurance case is a structured, logical argument in support of a goal backed by a body of evidence. Assurance cases are commonly used to argue that a system will operate as intended for a particular application and are often represented with an appropriate graphical notation such as GSN (Group, 2011) where logical steps in the argument are represented as nodes in a hierarchical tree structure. We assume familiarity with the core GSN notation and refer the reader to (Kelly and Weaver, 2004) and (Spriggs, 2012) for a more complete introduction.

While the details of every assurance case differ, common repeated structures of successful argumentation emerge. These successful structures can be captured as assurance case patterns (Kelly, 1999) (Kelly and McDermid, 1997) where details of the specific argument are abstracted as free parameters to be instantiated.

These patterns are able to capture domain knowledge and experience from previously successful arguments, reduce the effort required for construction of new assurance cases, and improve the quality of newly constructed assurance case. GSN (Group, 2011) provides two types of abstraction for describing these patterns: *structural* and *entity*. *Structural* abstractions generalize the relationships between two or more objects in the argument. *Structural* abstractions can be further divided into *multiplicity*, which generalizes *n*-ary relations between objects, and *optionality*, which generalizes *n*-of-*m* choices between objects. *Entity* abstractions include the object attributes *uninstantiated*, where free parameters are replaced with concrete values when the object is instantiated, and *undeveloped*, where the object requires additional support in the argument. Beyond the abstract structure, an assurance case pattern also contains descriptions about the pattern and how it may best be used including: name, intent, motivation, applicability, participants, collaborations, consequences, implementation, and related patterns.

A formal definition for assurance case patterns has been provided in (Denney and Pai, 2013) including the addition of typed parameters along with a generic algorithm for instantiating patterns based on a corresponding table of parameter assignments known as a $\mathcal{P}$-table. Prior to this, pattern instantiation was a largely informal process where abstract elements of the pattern were manually replaced with concrete instances and parameters could be assigned values of any type. However, construction of the $\mathcal{P}$-table is still a largely manual process where appropriate parameter assignments must be determined from outside information about the system.

An example assurance case pattern based on the As Low As Reasonably Practicable (ALARP) pattern (Kelly, 1999) is shown on the left side of Fig. 5.1. This argument pattern argues for system safety through the identification of hazards and reduction of the potential risks posed by those hazards to levels that are as low as reasonably practicable. Accordingly, the root goal *G1* states that a particular system, represented by parameter *A* of type *System*, conforms to the ALARP principle. This goal is decomposed into two sub-goals, *G2* and *G3*, through the argument strategy block *S1*. Goal *G2* claims that no intolerable risks are present in the system and can be further supported either with argument strategy *S2* or *S3*. Goal *G3* states that the risk from each hazard present in the system, represented by parameter *B* of type *Hazard*, has been reduced in accordance with the ALARP principle. The acceptable level of risk for each hazard will depend on the severity class assigned to that hazard, which is provided in context *C2* as the parameter *C* which is an attribute, *Severity*, of a *Hazard* object.

5.2.2 Model Based Engineering

In order to handle the complexity of modern CPSs, many systems are developed using Model Based Engineering techniques where the system is described by a set of system models. Each model captures informa-

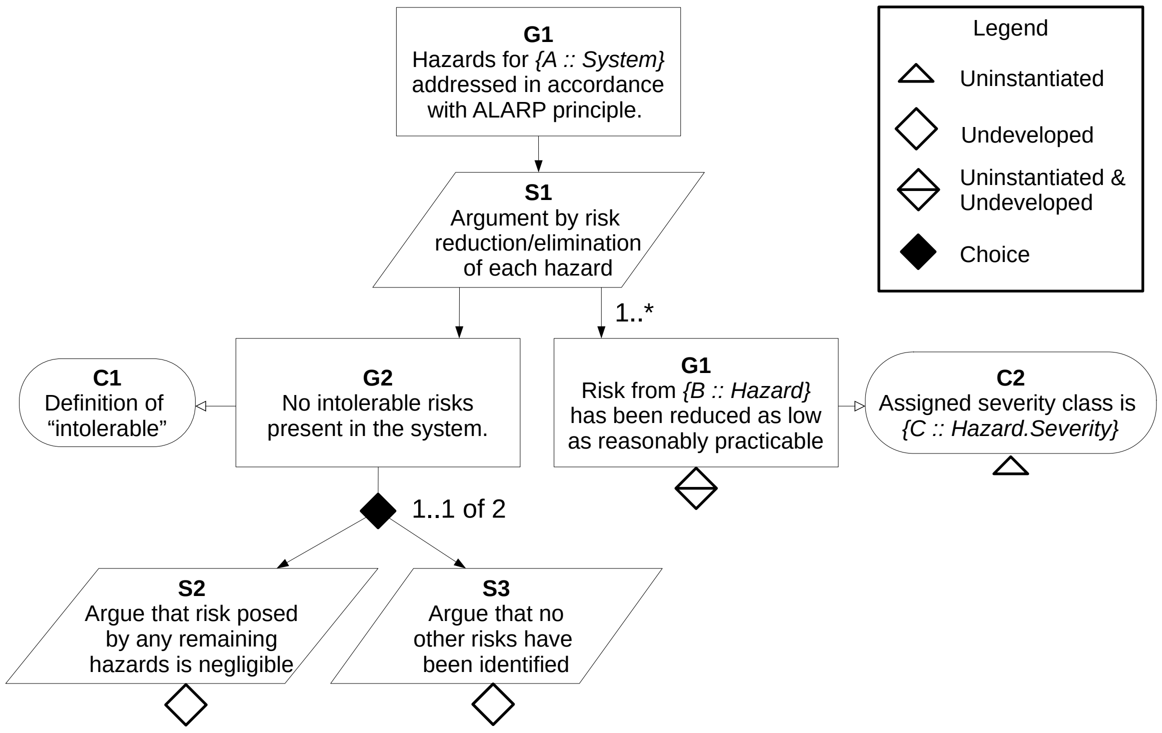

G1
Hazards for {A :: System} addressed in accordance with ALARP principle.
S1
Argument by risk reduction/elimination of each hazard
1..*
C1
Definition of "intolerable"
G2
No intolerable risks present in the system.
G1
Risk from {B :: Hazard} has been reduced as low as reasonably practicable
C2
Assigned severity class is {C :: Hazard.Severity}
1..1 of 2
S2
Argue that risk posed by any remaining hazards is negligible
S3
Argue that no other risks have been identified
Legend
Uninstantiated
Undeveloped
Uninstantiated & Undeveloped
Choice

Figure 5.1: Example assurance case pattern based on the ALARP pattern.

tion about different aspects of the system design such as architectural layout, functional decomposition, and hazard mitigation among others. Each aspect is commonly described using an appropriate DSML such as an internal block diagram (Friedenthal et al., 2014), functional decomposition model, or Bow-Tie Diagram (de Ruijter and Guldenmund, 2016). While each type of model describes a distinct aspect of the system design, they are rarely disconnected from the other aspects. Instead, there are various interdependencies between models which are captured using cross-model relationships.

Various integrated modeling tools (e.g., (Maróti et al., 2014), (White et al., 2007), (IBM, 2021)) exist where many system models described using multiple DSMLs can be contained in a single environment. Such environments allow for the interrelationships between models of different languages to be captured directly in the models themselves. These environments often use a *meta-model* which describes the syntax and semantics for each DSMLs used in the system modeling process.

For example, the meta-model in Fig. 5.2 describes the composition rules for three DSMLs: hazard description, functional decomposition, and Bow-Tie Diagram (BTD). A *Hazard Description* model simply contains one element for each identified *Hazard* including an attribute for the severity class of the hazard. *Functional Decomposition* models allow high-level *Functions* to be described as a composition of lower-level functions which can be joined with a *Conjunction* of the appropriate type. A BTD captures how a hazard may

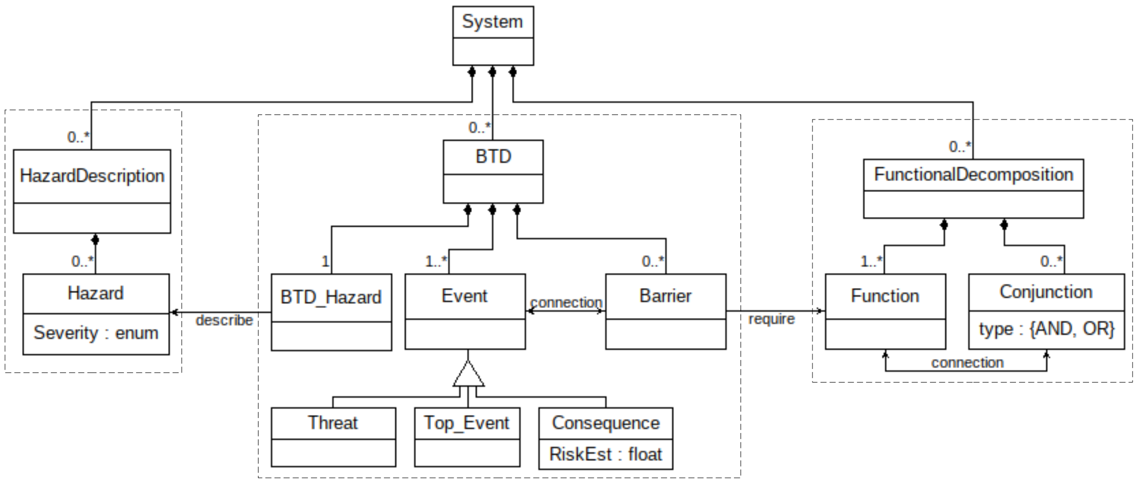


Figure 5.2: Meta-model defining the DSMLs used for BlueROV example system.

escalate through a chain of *events* and how this escalation can be prevented with control strategies known as *barriers*. These distinct DSMLs are interconnected with two cross-model relationships: (1) a *describe* relationship between the BTD and *Hazard* objects since a BTD provides additional explanation of possible hazard escalation, and (2) a *require* relationship between the *Barrier* and *Function* objects since the control strategies which a barrier represents may require certain system functions to operate. Note that all object classes in this meta-model also have an inherent *name* attribute which is not shown.

5.3 Assurance Case Construction

In this section, we present our automated assurance case construction process which leverages information available in an existing set of system models to assign parameter values and instantiate patterns, then composes these patterns appropriately to form a larger assurance argument. The required information is gathered from multiple heterogeneous, interconnected models. While the process is fully automated where possible, certain decision points remain which must be resolved by the developer resulting in a guided construction process. Before applying this process, we assume a number of artifacts are available including:

- A library of assurance case patterns.
- A meta-model which defines the composition rules for each type of model and the possible relationships between models.
- A set of models describing the system which comply with the above meta-model.

The following subsections describe these inputs in more detail before explaining the assurance case construction process.

5.3.1 Pattern Formalization

Within an assurance case pattern, there exists one or more parameters which must be assigned values before the pattern can be instantiated. Depending on multiplicity in the pattern structure, a single abstract parameter may be instantiated with multiple concrete values. To differentiate between the abstract and concrete parameters, we introduce *Formal Parameters* which describe an abstract class for parameters and *Parameter Instances*, or simply *Parameters*, which refer to concrete parameters with assigned values. These concepts are defined as follows:

Definition 1 (Formal Parameter) *A formal parameter G is a tuple $\langle n, t, P \rangle$ where n is the identifier or name, t is the type, and P is a set of parameter instances which inherit from the formal parameter.*

Definition 2 (Parameter Instance) *A parameter p is a tuple $\langle v, id, G \rangle$ where v is the assigned value, id is the instance identifier, and G is the formal parameter which p inherits from. For a parameter to be valid, we require $type(v) = G.t$.*

A specific formal parameter is identified with both the name $G.n$ and type $G.t$ as $G.n$::$G.t$. For example, formal parameter *A* with type *Hazard* would be written "A::Hazard". Similarly, a specific parameter p is identified using both the formal name $p.G.n$ and instance id $p.id$ as $p.G.n$:$p.id$. For example, the first instance of formal parameter *A* would be written as "A:0" while subsequent instances are written "A:m" where m is the index of the parameter instance. Note that the indexing of parameters is useful for uniquely identifying parameter instances, but the specific ordering of parameters within a formal parameter is unimportant for the construction method presented in this paper.

We use the formal definition of an assurance case pattern as presented in (Denney and Pai, 2013) with slight modifications including functions for mapping nodes to *formal parameters* and identifying *undeveloped* nodes as well as the addition of a *parameter hierarchy*. For clarity, the complete definition is stated here as follows:

Definition 3 (Pattern) *A pattern $\mathcal{P}$ is a tuple $\langle N, \mathcal{H}, l, t, p, m, c, u, \rightarrow \rangle$ where $\langle N, \rightarrow \rangle$ is a finite, directed hypergraph in which each hyperedge has a single source and possibly multiple targets. $\mathcal{H}$ is a parameter hierarchy, defined later in this section, and the functions l, t, p, m, and c are as follows:*

- *l and t are labeling functions where $l : N \rightarrow \{s, g, e, a, j, c\}$ gives the type of each node and $t : N \rightarrow str$ gives the string contents of each node.*
- *$p : N \rightarrow \mathcal{G}$, where $\mathcal{G}$ is a set of formal parameters, maps each node to the corresponding formal parameters.*

- $m : (\rightarrow) \times \mathbb{N} \rightarrow \mathbb{N}^2$ *gives the multiplicity range for the* i^{th} *outgoing connector of the hyperedge* $e \in (\rightarrow)$ *as a pair* $\langle l, h \rangle$*. This pair represents an inclusive value range* $[l, h]$ *where* $l \leq h$*. An optional connection is represented with the range* $\langle 0, 1 \rangle$*.*

- $c : (\rightarrow) \rightarrow \mathbb{N}^2$ *gives the choice range of a hyperedge as a pair* $\langle l, h \rangle$*. This is commonly written as "*$l..h$ *of* n*" in pattern specifications, however the* n *is omitted here as it can be inferred from the pattern structure.*

- $u : N \rightarrow \{True, False\}$ *is a boolean operator where a true value indicates that a node is undeveloped.*

Since circular paths are allowed in the pattern structure, it is possible that all nodes have incoming edges and therefore no true root node exists. However, in such a cycle there is still an intended starting node from which the instantiation should begin which can be considered as a pseudo-root node. We define the function $isroot_{\mathcal{P}}(r)$ *for checking if the node* r *is either a true root or a pseudo-root in the pattern. The pattern must also satisfy additional conditions including:*

- *Each root of the pattern is a goal:* $isroot_{\mathcal{P}}(r) \Rightarrow l(r) = g$*.*

- *Connectors only leave strategies or goals:* $n \rightarrow m \Rightarrow l(n) = s, g$*.*

- *Strategies cannot connect to other strategies or evidence:* $(n \rightarrow m) \;\; \& \;\; [l(n) = s] \Rightarrow l(m) = g, a, j, c$*.*

Within a pattern, there is a parameter dependency chain where certain predecessor parameters must be instantiated before successor parameters. For example, in Fig. 5.1 the formal parameter *A :: system* must be assigned a value before any values can be assigned for the subsequent formal parameter *B :: hazard*. While this dependency can often be inferred from the pattern structure, there are certain conditions where multiple valid dependency chains are possible including: circular paths in the pattern structure, nodes with multiple parents, and nodes with multiple formal parameters. To address this, we make the parameter dependency chain explicit with a *Parameter Hierarchy*, defined as follows:

Definition 4 (Parameter Hierarchy) *A parameter hierarchy* $\mathcal{H}$ *is a tuple* $\langle \mathcal{G}, m, \rightarrow \rangle$ *where* $\mathcal{G}$ *is a set of formal parameters,* $\langle \mathcal{G}, \rightarrow \rangle$ *forms a finite directed graph where each edge represents a parameter instantiation dependency from the parent formal parameter to the child formal parameter, and* $m : (\rightarrow) \rightarrow \mathbb{N} \times \mathbb{N}$ *is function which returns the multiplicity range of each edge as a lower bound, upper bound pair.*

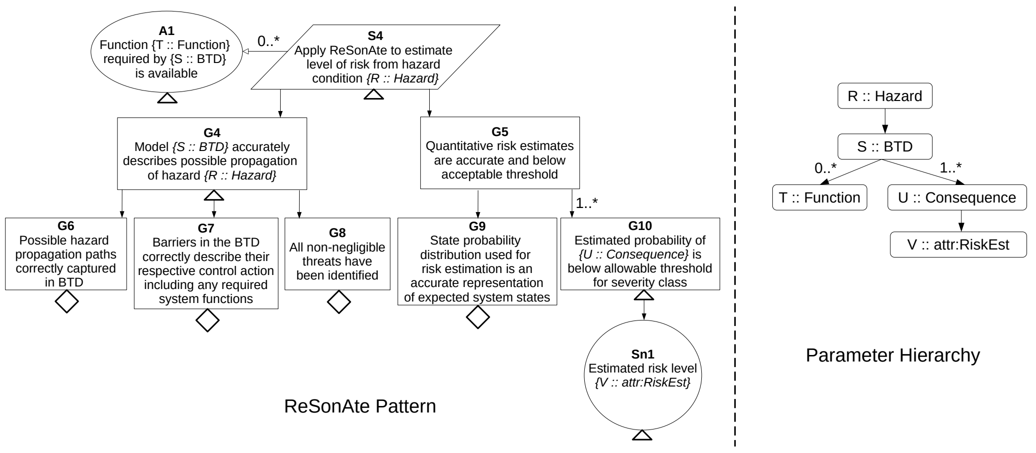

A1
Function {T :: Function} required by {S :: BTD} is available
0..*
S4
Apply ReSonAte to estimate level of risk from hazard condition {R :: Hazard}
G4
Model {S :: BTD} accurately describes possible propagation of hazard {R :: Hazard}
G5
Quantitative risk estimates are accurate and below acceptable threshold
1..*
G6
Possible hazard propagation paths correctly captured in BTD
G7
Barriers in the BTD correctly describe their respective control action including any required system functions
G8
All non-negligible threats have been identified
G9
State probability distribution used for risk estimation is an accurate representation of expected system states
G10
Estimated probability of {U :: Consequence} is below allowable threshold for severity class
Sn1
Estimated risk level {V :: attr:RiskEst}
ReSonAte Pattern
R :: Hazard
S :: BTD
0..*
1..*
T :: Function
U :: Consequence
V :: attr:RiskEst
Parameter Hierarchy

Figure 5.3: Assurance Case Pattern for application of the ReSonAte framework.

Whenever the parameter hierarchy cannot be uniquely inferred from the pattern structure, the developer must provide the desired hierarchy alongside the pattern definition. For example, Fig. 5.3 introduces the ReSonAte pattern and associated parameter hierarchy. Unlike the previous ALARP pattern, an explicit hierarchy is required for the ReSonAte pattern since the correct order of parameter assignment is not clear from the pattern structure. This pattern argues that the risk from a particular hazard has been sufficiently reduced, in accordance with the ALARP principle, through the use of the ReSonAte framework (Hartsell et al., 2021). This framework captures possible hazard escalation paths and the hazard control strategies which prevent this escalation using BTDs, then applies statistical methods to estimate the probability of various undesirable events, known as consequences, from operational data. Thus, the ReSonAte assurance pattern can be broken into two primary sections: (1) an argument for the correctness of the BTD and the hazard escalation paths which it describes, and (2) an argument for the accuracy of the estimated consequence probabilities.

A similar parameter dependency issue occurs when a formal parameter is instantiated with one or more concrete parameter instances. Multiple parameter instances may inherit from the same formal parameter yet have different parent parameters. For example, in Fig. 5.1 three parameters of type hazard may be instantiated for formal parameter *B :: hazard*: *B:0*, *B:1*, and *B:2*. Each of these parameters have a child parameter which inherits from formal parameter *C :: severity*: *C:0*, *C:1*, and *C:2* where parameter $C : i$ is a child of $B : i$ for $i = 0, 1, 2$. In this case, parameters *C:0*, *C:1*, and *C:2* share the same formal parameter C but have different parent parameters *B:0*, *B:1*, and *B:2* respectively. To capture these parameter relationships, we define a *Parameter Graph* as follows:

Definition 5 (Parameter Graph) *A parameter graph $\mathcal{K}$ is a tuple $\langle P, r, \rightarrow\rangle$ where P is a set of parameter instances and $\langle P, \rightarrow\rangle$ forms a finite DAG. Each edge $(\rightarrow) : \langle P, P\rangle$ represents a parent/child relationship*

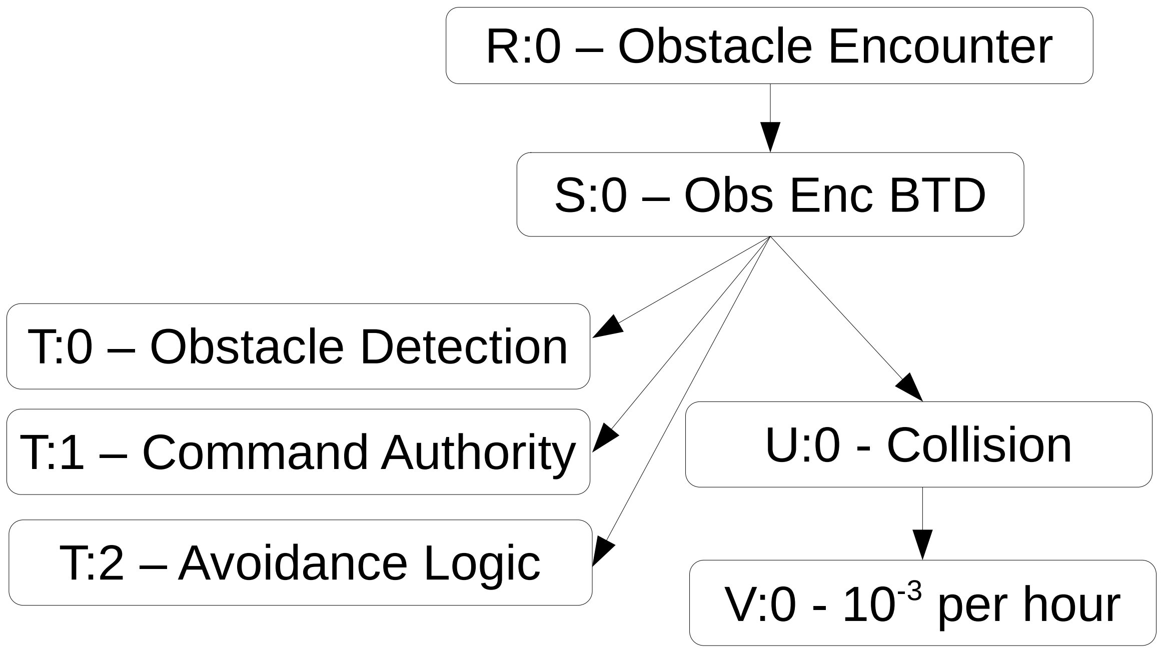


Figure 5.4: Instantiated parameter graph which is valid with respect to the ReSonAte parameter hierarchy on the right of Fig. 5.3

between two parameter instances. The function $r : (\rightarrow) \rightarrow R$, where R is a set of expressions, provides an explanation of the reasoning behind each parent/child relationship and is explained in more detail in Section 5.3.3.

We say that a Parameter Graph $\mathcal{K}$ is valid with respect to a Parameter Hierarchy $\mathcal{H}$, written as $\mathcal{K}_{\mathcal{H}}$, iff all parameter edges in $\mathcal{K}$ respect the formal parameter ordering defined in $\mathcal{H}$. That is, for all edges $(p_1, p_2) \in \mathcal{K}.(\rightarrow)$ there exists a corresponding edge $(p_1.G, p_2.G) \in \mathcal{H}.(\rightarrow)$.

Figure 5.4 shows an example of an instantiated parameter graph which is valid with respect to the parameter hierarchy in Fig. 5.3. The construction process this parameter graph is outlined as part of our illustrative example in Section 5.4.

5.3.2 System Models

For systems developed using MBE techniques, much of the information required to assign parameter values and instantiate an assurance case pattern is available in the system models. These models are the result of system analysis and design processes such as architecture models from structural decomposition or hazard models from hazard identification and analysis. Typically, each aspect of the system is described using a DSML which is tailored for the specific task. In order to apply our assurance case construction process, we assume we have a set of such models describing the system and that these models conform to a provided meta-model such as the one shown in Fig. 5.2. We define this model set as follows:

Definition 6 (System Model Set) *A system model set $\mathcal{S}_M$ defined with respect to a particular meta-model M is a tuple $\langle N, t, r, a, \rightarrow \rangle$ where N is a set of model objects and $\langle N, \rightarrow \rangle$ forms a finite directed graph which respects the composition rules defined by M. The functions t, r, and a are as follows:*

- *t gives the object-type of each node. $t : N \rightarrow T$, where T is the set of object types defined by the corresponding meta-model.*
- *r gives the relationship-type of each edge. $r : (\rightarrow) \rightarrow R$, where R is the set of relationship types defined by the corresponding meta-model.*
- *a gives the attribute value of a node, string pair. $a : (N \times str) \rightarrow str$.*

5.3.3 Technique

As the first step in the construction process, the developer must select an appropriate top-level pattern from the pattern library, based on the desired goal, which will act as the root of the assurance case. For example, a developer aiming to support the claim that all system requirements have been addressed, either with direct support or by refinement to lower-level requirements, might select the "Requirements Breakdown" pattern (Denney and Pai, 2015) as a starting point. Once this selection has been made, assignments for all free parameters must be made before the pattern can be instantiated.

For our approach, this means we must construct a valid *parameter graph* for the selected pattern based on information contained in the *system model set*. Once this parameter graph is constructed, we apply an existing algorithm to perform pattern instantiation (Denney and Pai, 2013). As discussed in Section 5.2.1, this algorithm requires a table of parameter assignments, or $\mathcal{P} - table$, which can be obtained from the *parameter graph* with a straightforward transformation. While much of the process is automated, certain decision points remain which require manual guidance and are denoted in the following algorithms by the text **Decision Point**. At these points, the developer is presented a set of candidates, automatically pruned to the extent possible based on pattern constraints, and must select one or more options from this set before the process can continue.

Construction of the parameter graph is a non-trivial process with several prerequisite algorithms. First, we define a function which finds all paths from one object in the the system model set to another. This is a similar concept to the transitive closure of the system model set, but we allow the directed edges in the system model to be traversed in either direction. This is suitable since the directional meaning of each relationship can be reversed. For example, the *require* relationship between types *Function* and *Barrier* in Fig. 5.2 can be interpreted either as "Barriers require Functions" or "Functions are required by Barriers". For brevity, we introduce the $pop(X)$ operator, which returns the head of a list X and removes the returned item from X, and define the path finding function as follows:

Algorithm 1 Find Paths

```
1:  function FINDPATHS($\mathcal{S}$: System Model Set, $n_{start}$: Node, $n_{end}$: Node, $path$, $visited$)
2:      require $n_{start}, n_{end} \in \mathcal{S}.N$
3:      $visited \leftarrow visited \cup n_{start}$
4:      if $n_{start} = n_{end}$ then
5:          return $\{path\}$
6:      end if
7:      $paths \leftarrow \emptyset$
8:      $q \leftarrow \{e \in \mathcal{S}.(\rightarrow) \mid e = \langle n_{start}, N \rangle\}$
9:      while $q \neq \emptyset$ do
10:         $n_0 \leftarrow pop(q)$
11:         if $n_0 \notin visited$ then
12:             $relation \leftarrow \mathcal{S}.r(\langle n_{start}, n_0 \rangle)$
13:             $step \leftarrow \langle n_{start}, n_0, relation \rangle$
14:             $paths_{new} \leftarrow FindPaths(\mathcal{S}, n_0, n_{end}, path \cup step, visited)$
15:         end if
16:         $paths \leftarrow paths \cup paths_{new}$
17:     end while
18:     $q \leftarrow \{e \in \mathcal{S}.(\rightarrow) \mid e = \langle N, n_{start} \rangle \wedge \mathcal{S}.r(e) \neq contain\}$
19:     while $q \neq \emptyset$ do
20:         $n_0 \leftarrow pop(q)$
21:         if $n_0 \notin visited$ then
22:             $relation \leftarrow \mathcal{S}.r(\langle n_0, n_{start} \rangle)$
23:             $step \leftarrow \langle n_0, n_{start}, relation \rangle$
24:             $paths_{new} \leftarrow FindPaths(\mathcal{S}, n_0, n_{end}, path \cup step, visited)$
25:         end if
26:         $paths \leftarrow paths \cup paths_{new}$
27:     end while
28:     return $Paths$
29: end function
```

Note that each $path$ in the $Paths$ set returned by this function is an ordered set of tuples $\langle n_1, n_2, relation \rangle$ where n_1, n_2 are model objects and $relation$ is the type of relationship connecting the two objects. This allows each path to contain a complete explanation for how the initial two objects, n_{start} and n_{end}, are related through one or more model relationships. This explanation is useful later in the construction process for enabling the developer to (1) make an informed decision when selecting between alternative parameter assignment options and (2) understand why parameters were assigned a particular value when reviewing the final assurance case.

When assigning a value to a new parameter instance, two primary primary conditions must be satisfied: (1) the value must be of the correct type, and (2) the value must be suitably related to its parent parameter, except in the case of the root parameter which has no parent. The second subroutine for the construction process is a graph traversal function, defined as Algorithm 2, which searches the system model set and returns a list of all objects which satisfy both constraints and are therefore valid candidates for parameter assignment. Note that it is possible for multiple paths to exist between a candidate object and the parent object in the system model set. In this case, one potential match is added to the list of matches for each possible path to the candidate object.

Algorithm 2 Search Models

```
function SearchModels($\mathcal{S}$: System Model Set, $t$: Meta Type, $p_{parent}$: Parameter)
    $nodes \leftarrow \{n \in \mathcal{S}.N \mid \mathcal{S}.t(n) = t\}$
    if $p_{parent} = \emptyset$ then
        return $nodes$
    end if
    $matches \leftarrow \emptyset$
    for all $node \in nodes$ do
        $paths_{parent} \leftarrow FindPaths(\mathcal{S}, p_{parent}, node, \emptyset, \emptyset)$
        if $paths_{parent} \neq \emptyset$ then
            for all $path \in paths_{parent}$ do
                $matches \leftarrow \langle node, path \rangle$
            end for
        end if
    end for
    return $matches$
end function
```

Once an appropriate set of matching model objects has been selected, a parameter instance needs to be instantiated and added to the parameter graph for each object in the set. Since the *value* of a parameter is a direct link to a model object, traceability from each parameter in the assurance case to the artifact it was sourced from is maintained. We define functions for instantiating a parameter in Algorithm 3.

Algorithm 3 Add New Parameter

```
function InstParam($G$: Formal Parameter, $v$)
    require $type(v) = G.t$
    $id \leftarrow len(G.P)$
    $p_{new} \leftarrow \langle v, id, G \rangle$
    $G.P \leftarrow G.P \cup p_{new}$
    return $G, p_{new}$
end function

function AddParam($\mathcal{K}$: Parameter Graph, $p_{parent}$: Parameter, $p_{new}$: Parameter, $path$)
    $\mathcal{K}.P \leftarrow \mathcal{K}.P \cup p_{new}$
    if $p_{parent} \neq null$ then
        $\mathcal{K}.(\rightarrow) \leftarrow \mathcal{K}.(\rightarrow) \cup \langle p_{parent}, p_{new} \rangle$
        define $\mathcal{K}.r(\langle p_{parent}, p_{new} \rangle) \rightarrow path$
    end if
    return $\mathcal{K}$
end function
```

Using these subroutines, we define the overall process for instantiation of the parameter graph in Algorithm 4 and provide an explanation of this algorithm as follows. First, we begin with the parameter hierarchy of the pattern to be instantiated and initialize an empty parameter graph. Next, a set for formal parameter information (GI) is initialized with a tuple containing the root formal parameter in the hierarchy and a $null$ parent. For each element in GI, we identify all model objects which are potential candidates for parameter instantiation - i.e. model objects which match the parameter type and are related to the parent object. For the root formal parameter which has no parent, all objects in the system model set with the correct type are identified. Additionally, patterns are often sequentially composed with other patterns where parameter values may already be assigned. For this case, we provide the argument p_{ext} which indicates the root formal parameter should be instantiated with a predefined value. Otherwise, the developer must choose a subset of the matching candidates which are then instantiated and added to the parameter graph. Each of these instances may themselves have successor formal parameters in the hierarchy. For each successor, an entry is added to the set GI containing both the formal parameter and the current parameter instance as a parent. The process then repeats until all entries in GI have been considered and the parameter graph is complete.

Finally, we define the complete process for assurance case construction in Algorithm 5 which operates as follows. Once a parameter graph has been constructed for the initial pattern, the pattern can be instantiated by applying the algorithm detailed in (Denney and Pai, 2013), identified as $InstantiatePattern(\mathcal{P}, \mathcal{K})$, which takes in a pattern definition, parameter graph pair and returns an instantiated version of the same pattern. This instantiated pattern forms an initial *partial* assurance case. We call this initial pattern *partial* since instantiation of a single pattern is rarely sufficient for a complete assurance argument and most often will contain *undeveloped* nodes requiring additional support before the argument can be considered sound.

The next step in the construction process is to identify any such undeveloped nodes and suggest patterns from the pattern library which could provide additional support for each node. Candidate patterns are identified by type matching any parameters associated with the undeveloped node to the root parameter of each pattern in the library. Any patterns which match the required parameter type are suggested to the developer as a possible avenue for expanding the assurance case through sequential composition. Once a pattern has been selected, the instantiation process is repeated and we define the function $join(\mathcal{P}', \mathcal{A}, node)$ which sequentially composes the instantiated pattern $\mathcal{P}'$ with the existing partial assurance case $\mathcal{A}$ at a particular $node$. This process repeats until all undeveloped nodes in the partial assurance case have been addressed.

For brevity, Algorithm 5 uses a convenience function $parameterType(node)$, which returns the type of the parameter associated with $node$, and assumes that each undeveloped node has exactly one parameter. However, this can be expanded to nodes with zero parameters, which are simply skipped, and nodes with multiple parameters, where each parameter could be sequentially considered as an avenue for expansion. This algorithm produces an assurance case composed of one or more patterns instantiated from information

Algorithm 4 Build Parameter Graph

```
 1: procedure BuildParameterGraph($H$: Parameter Hierarchy, $\mathcal{S}$: System Model Set, $p_{ext}$: Parameter)
 2:     $\mathcal{K} \leftarrow \langle \emptyset, r, \emptyset \rangle$
 3:     $G_{root} \leftarrow root(H)$
 4:     $GI \leftarrow \langle G_{root}, null \rangle$
 5:     while $GI \neq \emptyset$ do
 6:         $\langle G, p_{parent} \rangle \leftarrow pop(GI)$
 7:         if $(p_{parent} = null) \wedge (p_{ext} \neq null)$ then
 8:             require $G.t = p_{ext}.G.t$
 9:             $selections \leftarrow \{\langle p_{ext}.v, \emptyset \rangle\}$
10:         else
11:             if $p_{parent} = null$ then
12:                 $\langle l, u \rangle \leftarrow \langle 1, 1 \rangle$
13:             else
14:                 $\langle l, u \rangle \leftarrow H.m(\langle p_{parent}.G, G \rangle)$
15:             end if
16:             $matches \leftarrow SearchModels(\mathcal{S}, G.t, p_{parent})$
17:             Decision Point: $selections \subseteq matches$
18:         end if
19:         require $l \leq len(selections) \leq u$
20:         for all $\langle value, path \rangle \in selections$ do
21:             $\langle G, p_{new} \rangle \leftarrow InstParam(G, value)$
22:             $\mathcal{K} \leftarrow AddParam(\mathcal{K}, G, p_{parent}, p_{new}, path)$
23:             $G_{successors} \leftarrow \{g \in H.G | \langle G, g \rangle \in H.(\rightarrow)\}$
24:             for all $\langle g_s \rangle \in G_{successors}$ do
25:                 $GI \leftarrow GI \cup \langle g_s, p_{new} \rangle$
26:             end for
27:         end for
28:     end while
29:     return $\mathcal{K}$
30: end procedure
```

in the system model set. The generated assurance case will also require manual review and refinement to (1) address undeveloped nodes that could not be supported with patterns from the pattern library, either with hand-crafted supporting arguments or with direct support from evidence, and (2) decide if the generated assurance case provides sufficient support for the original goal.

Algorithm 5 Build Assurance Case

```
 1: procedure BuildAssuranceCase($L$: Pattern Library, $\mathcal{S}$: System Model Set)
 2:     Decision Point: $\mathcal{P} \in L$
 3:     $\mathcal{K} \leftarrow BuildParameterGraph(\mathcal{P}.H, \mathcal{S}, null)$
 4:     $\mathcal{A} \leftarrow InstantiatePattern(\mathcal{P}, \mathcal{K})$
 5:     $nodes_u \leftarrow \{n \in \mathcal{A}.N \mid \mathcal{A}.u(n) = True\}$
 6:     while $nodes_u \neq \emptyset$ do
 7:         $node \leftarrow pop(nodes_u)$
 8:         $type_n \leftarrow parameterType(node)$
 9:         $candidates \leftarrow \{\mathcal{P} \in L \mid root(\mathcal{P}.H).t = type_n\}$
10:         Decision Point: $\mathcal{P} \in candidates$
11:         $\mathcal{K} \leftarrow BuildParameterGraph(\mathcal{P}.H, \mathcal{S}, node)$
12:         $\mathcal{P}' \leftarrow InstantiatePattern(\mathcal{P}, \mathcal{K})$
13:         $nodes_u \leftarrow nodes_u \cup \{n \in \mathcal{P}'.N \mid \mathcal{P}'.u(n) = True\}$
14:         $\mathcal{A} \leftarrow join(\mathcal{P}', \mathcal{A}, node)$
15:     end while
16:     return $\mathcal{A}$
17: end procedure
```

5.4 Example Application

In this section, we provide an illustrative example by applying the assurance case construction process to a UUV based on the BlueROV2 (Blue Robotics, 2016) vehicle. The objective of this system is to autonomously track a pipeline on the seafloor while avoiding static obstacles (e.g., plants, rocks, etc.) and respecting predefined limits on the area of operation. In this example, we intend to argue for system safety through identification of potential hazards and reduction of the risk posed by those hazards based on the ALARP principle. The complete model set for this system is comprised of many different DSMLs, but a restricted set, including BowTie, Functional, and Hazard models, is considered here. The pattern library consists of the ALARP and ReSonAte patterns introduced earlier, and both patterns are instantiated with information sourced from the system models.

An overview of the BlueROV2 system model set is shown in Fig. 5.5, with the associated meta-model shown in Fig. 5.2, and consists of a hazard description model, three functional decomposition models, and three BTDs. Three potential hazards have been identified for the system including deviation from the operating area, obstacle encounter, and loss of pipeline. For each hazard, a severity level has been assigned based on any potential consequences that may result and an object describing the hazard exists in the hazard description model. Each hazard also has an associated BTD which describes potential hazard escalation paths. For

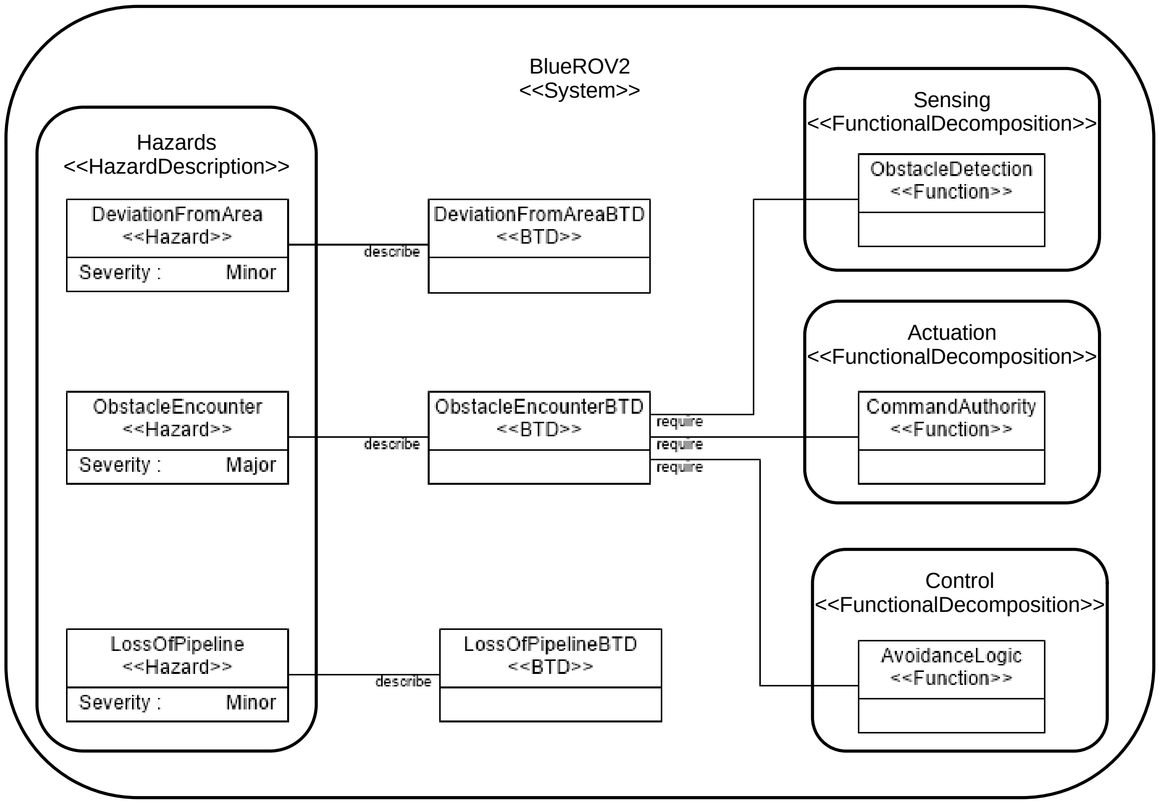
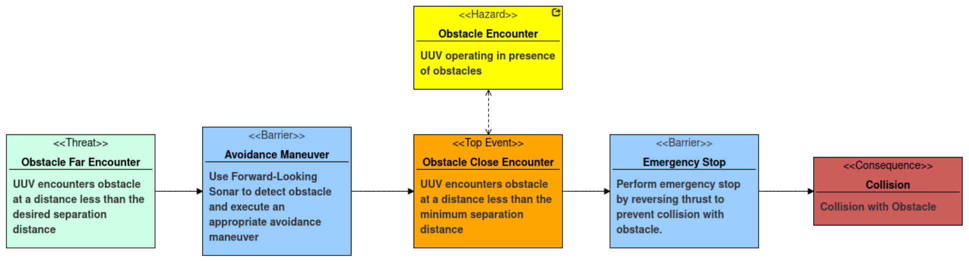


Figure 5.5: Overview of the model set for the BlueROV2 system.

brevity, only one such BTDs describing the obstacle encounter, shown in Fig. 5.6, is utilized in this assurance case construction example. The *Avoidance Maneuver* and *Emergency Stop* barriers in this BTD represent hazard control strategies implemented in the system, but require certain functions to operate as shown in Fig. 5.5. Finally, one functional decomposition model exists for each of three primary system functions: sensing, actuation, and control. For simplicity, each of these models contains only a single function: obstacle detection, command authority, and avoidance logic respectively.

To begin the construction process, we select the ALARP pattern shown in Fig. 5.1 as the initial pattern for our assurance case and initialize an empty parameter graph. The root parameter group of this pattern is *A :: System* for which there is only one matching object in the model set: *BlueROV2*. We instantiate the parameter *A:0 - BlueROV2* which becomes the root of the parameter graph. For the successor group *B :: Hazard*, three hazard objects exist which are part of the BlueROV2 model set. All three hazards are selected for instantiation, and parameters *B:0 - Loss of Pipeline*, *B:1 - Deviation from Area*, and *B:2 - Obstacle Encounter* are added to the parameter graph. Each of these hazard objects contain a *Severity* attribute which satisfies the final parameter group *C :: Hazard.Severity* resulting in the addition of parameters *C:0 - Minor*, *C:1 - Minor*, and *C:2 - Major* to the parameter graph. The final parameter graph for this pattern is shown in



<<Hazard>>
Obstacle Encounter
UUV operating in presence of obstacles
<<Threat>>
Obstacle Far Encounter
UUV encounters obstacle at a distance less than the desired separation distance
<<Barrier>>
Avoidance Maneuver
Use Forward-Looking Sonar to detect obstacle and execute an appropriate avoidance maneuver
<<Top Event>>
Obstacle Close Encounter
UUV encounters obstacle at a distance less than the minimum separation distance
<<Barrier>>
Emergency Stop
Perform emergency stop by reversing thrust to prevent collision with obstacle.
<<Consequence>>
Collision
Collision with Obstacle

Figure 5.6: BowTie diagram which describes potential propagation of the *Obstacle Encounter* Hazard.

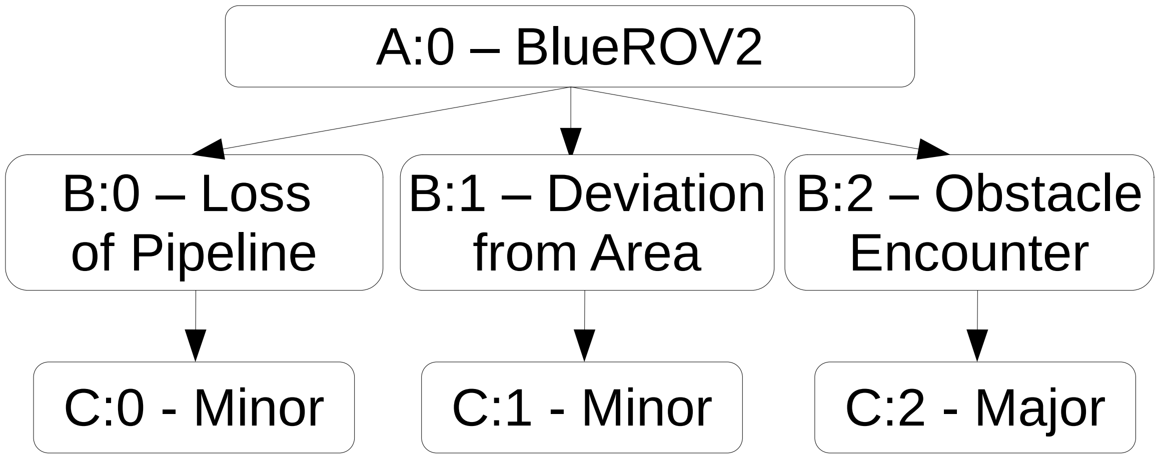
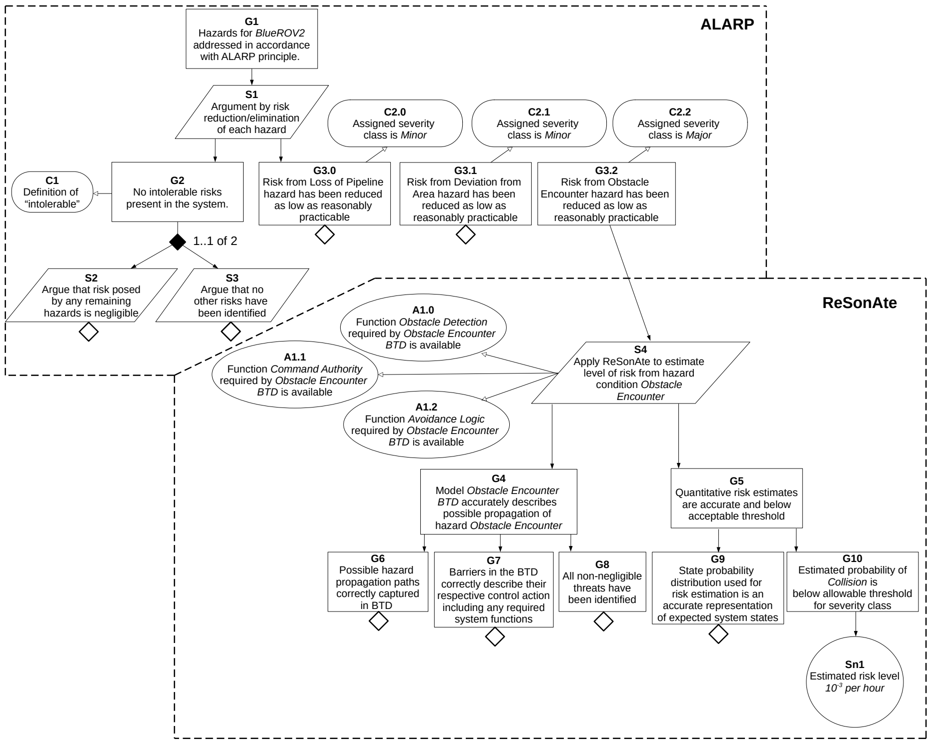


Figure 5.7: Parameter graph instantiated for ALARP pattern from BlueROV2 model set.

Fig. 5.7 and is used to instantiate the ALARP portion of the assurance case as shown in Fig. 5.8.

Next, each undeveloped node in the initial assurance case is examined for possible expansion with another pattern from the pattern library. While several such nodes exist in the assurance case at this stage, we show how one particular node, G3.2, can be further expanded. Since this node was instantiated from the parameter *B:2 - Obstacle Encounter* of type *Hazard*, a valid candidate pattern for sequential composition must have a root parameter that is also of type *Hazard*. The ReSonAte pattern presented in Fig. 5.3 matches this requirement with the root parameter group *R :: Hazard* and is selected for instantiation. Since this pattern will be an extension of an existing assurance case, the root parameter is assigned with the same value, *R:0 - Obstacle Encounter*. The child parameter group, *S :: BTD*, can be assigned the value *S.0 - Obstacle Encounter BTD* based on the direct *describes* relationship between these two objects in the system model set. From here, the parameter hierarchy splits with two children of group *S*, *T :: Function* and *U :: Consequence*. The obstacle encounter BTD contains one object of type Consequence, and thus we instantiate the parameter *U:0 - Collision*. For group *T :: Function* however, the relationship between BTDs and *Functions* is indirect and requires two steps: (1) BTDs contain *Barriers* and (2) *Barriers* require *Functions*. Using Algorithm 2, three



ALARP
G1
Hazards for BlueROV2 addressed in accordance with ALARP principle.
S1
Argument by risk reduction/elimination of each hazard
C2.0
Assigned severity class is Minor
C2.1
Assigned severity class is Minor
C2.2
Assigned severity class is Major
C1
Definition of "intolerable"
G2
No intolerable risks present in the system.
G3.0
Risk from Loss of Pipeline hazard has been reduced as low as reasonably practicable
G3.1
Risk from Deviation from Area hazard has been reduced as low as reasonably practicable
G3.2
Risk from Obstacle Encounter hazard has been reduced as low as reasonably practicable
1..1 of 2
S2
Argue that risk posed by any remaining hazards is negligible
S3
Argue that no other risks have been identified
ReSonAte
A1.0
Function Obstacle Detection required by Obstacle Encounter BTD is available
A1.1
Function Command Authority required by Obstacle Encounter BTD is available
A1.2
Function Avoidance Logic required by Obstacle Encounter BTD is available
S4
Apply ReSonAte to estimate level of risk from hazard condition Obstacle Encounter
G4
Model Obstacle Encounter BTD accurately describes possible propagation of hazard Obstacle Encounter
G5
Quantitative risk estimates are accurate and below acceptable threshold
G6
Possible hazard propagation paths correctly captured in BTD
G7
Barriers in the BTD correctly describe their respective control action including any required system functions
G8
All non-negligible threats have been identified
G9
State probability distribution used for risk estimation is an accurate representation of expected system states
G10
Estimated probability of Collision is below allowable threshold for severity class
Sn1
Estimated risk level 10^{-3} per hour

Figure 5.8: Generated assurance case for BlueROV2 system. Consists of ALARP pattern sequentially composed with *resonate* pattern.

functions required by the obstacle encounter BTD are identified and the parameters *T:0 - Obstacle Detection*, *T:1 - Command Authority*, and *T:2 - Avoidance Logic* are instantiated. The last remaining parameter group, *V :: Consequence.RiskEst* refers to an attribute of the previously instantiated *U:0 - Collision* and is instantiated as *V:0* - 10^{-3} *per hour*. The resulting parameter graph is shown in Fig. 5.4 and is used to instantiate the ReSonAte portion of the assurance case in Fig. 5.8 which is joined with the previous ALARP pattern at node G3.2.

The process could continue with a similar expansion of the undeveloped nodes G3.0 and G3.1, but we stop the example here for brevity. As discussed in Section 5.3.3, the generated assurance case is not complete and several undeveloped nodes remain which do not contain parameters and cannot be automatically instantiated with this process. While significant portions of the assurance case can be mechanically constructed, the precise extent of this automation depends on both the selection of patterns available in the pattern library and the amount of information contained in the system model set. Additionally, the process maintains traceability from parameters in the generated assurance case to the system models, shown in Fig. 5.5, as well as explainability of the relationships between models which drove the parameter assignment process. These features facilitate further human review and refinement of the assurance case.

5.5 Related Work

A formalization of assurance case pattern definitions and semantics has been presented in (Denney and Pai, 2013) along with an algorithm for mechanical instantiation of these patterns. Given a pattern $\mathcal{P}$, this algorithm uses a table of parameter assignments, known as a $\mathcal{P}$-table, to replace abstract elements in the pattern with concrete values. Algorithmic pattern instantiation offers several benefits over informal techniques, but construction of the $\mathcal{P}$-table itself is left as a largely manual process. The construction process presented in this paper defines an algorithmic approach for building this parameter assignment table by leveraging information from other system models. AdvoCATE (Denney et al., 2012), (Denney and Pai, 2018) is a tool based on these formalizations which provides automated support for a number of assurance case construction and maintenance tasks including an implementation of the pattern instantiation algorithm. However, a $\mathcal{P}$-table is still required and must be manually constructed, either interactively or as the direct product of another design process such as Functional Hazard Analysis (FHA).

Resolute (Gacek et al., 2014), (Amundson and Cofer, 2021) is another formally-based approach to assurance case construction which allows developers to include formal claims alongside component definitions in a system architecture model - e.g., the developer may claim that a particular component only produces messages that conform to a particular message specification. Using this enriched architecture model, Resolute then attempts to synthesize assurance cases which prove various system-level properties by composing component-level claims through a set of logical rules. Similarly, (Bagheri et al., 2020) presents a method for rigorous construction of assurance cases by synthesizing component-level properties from system-level requirements. While these techniques are rigorous, the range of system properties which can be proven is limited.

Other work has shown how common design patterns can be defined and applied to existing system models using a generic graph transformation language (Agrawal et al., 2005). Another graph transformation approach utilizes *weaving models* (Marcos et al., 2005) to capture fine-grained relationships between elements of heterogeneous models. The authors of (Hawkins et al., 2015) show how these weaving models can be used to instantiate assurance case patterns from system models. In this approach, the weaving model provides a mapping from objects in a particular DSML to corresponding parameters in a pattern definition and can be used to automate the instantiation process. However, this weaving model is specific to each pattern, DSML pair and must be manually constructed for each such pair.

5.6 Conclusion

We have presented an automated process for assurance case construction which utilizes information contained in system design models to instantiate patterns and compose these patterns into a larger assurance argument.

In order to support manual review and refinement of the generated assurance case, the approach ensures traceability from assigned parameters to the model objects they originate from as well as explainability of the relationships between parameters. These features also enable a level of consistency checking where portions of the generated assurance case can be invalidated if the associated model objects have changed. When a set of system design models is available, the technique can greatly reduce the effort required to compile relevant information from these models and construct an assurance case which should allow for faster iteration of the safety assurance argument alongside an evolving system design.

To estimate the amount of time saved using this method, the assurance case presented for our example system in Fig. 5.8 was first constructed manually by a developer who: identified appropriate assurance patterns, extracted the necessary information from a collection of existing design models, used this information to instantiate and compose each pattern, then added hyperlinks from the nodes in the constructed assurance case to the relevant model objects. This manual process took one developer approximately 2 hours and lacked the explanation of relationships between model objects described in Section 5.3.3. Our automated method was able to construct the same assurance case in under 5 minutes with the majority of this time being spent waiting for the developer to make selections at each of the decision points during the construction process. Note that in either case the constructed assurance case is not complete and will require further manual refinement. We expect the time savings from this technique to become even more significant as the number of system models and size of the generated assurance case increase.

We have identified multiple potential avenues for future work. First, a rich set of system design models often provides additional context beyond what a pattern definition requires. In many cases, it may be useful to automatically infer and add *context* and *assumption* blocks to the generated assurance case based on this contextual information. Second, the process currently only attempts to develop nodes which contain parameters for which patterns of matching types can be suggested. However, patterns can be identified with other means, e.g. by considering the context of nearby nodes in the argument, and used to expand undeveloped nodes that do not contain a parameter. Third, the richness of the model set dictates the extent to which a pattern can be automatically instantiated. This could be used to identify areas of the model set which are lacking or need review - e.g., an identified hazard that cannot be supported in the argument is not sufficiently well-described in the model set and thus requires a corresponding model to be constructed.

CHAPTER 6

Assurance Case Evaluation

Foreword

Assurance cases are a useful tool for better understanding how a system can achieve an acceptable level of safety. Qualitative evaluation methods can help identify potential problems in the argument such as anti-patterns, logical fallacies, and invalid assumptions. These techniques can be particularly helpful when applied early in the design process to assess the feasibility of adequately assuring safety for potential system architectures. However, many existing and emerging standards require some level of quantification for expected failure rates and the probability that system safety requirements will be satisfied.

Quantitative techniques are most often applied at design-time to generate static probability estimates for the validity of claimed system properties. These estimates represent average values over all expected operating conditions. However, conditions can change rapidly at run-time and may greatly impact the estimated probabilities. For example, assumptions made in the assurance argument may become invalid, which in turn can invalidate entire sections of the assurance case. Similarly, fault conditions may arise which increase the risk of failure but do not preclude system operation entirely. There is a need for quantitative evaluation techniques which can dynamically estimate risk levels posed to a system as the internal and environmental conditions evolve.

The publication in this chapter presents ReSonAte, a dynamic risk estimation framework for autonomous systems which captures information about possible hazard escalation paths and corresponding control strategies using BTDs. With ReSonAte, the likelihood of hazards and effectiveness of control strategies are subject to change conditional on the state of the system and environment which impacts the level of risk the system is exposed to. ReSonAte includes a technique for estimating these conditional relationships from simulated data encompassing a wide variety of possible scenarios. At runtime, the framework produces continually updated risk estimates based on observations about the environment and internal system state including component failure modes. These risk estimates can be used to guide system decision making as well as to evaluate the validity of assurance arguments. ReSonAte is applied to an autonomous ground vehicle and the estimated risk levels are validated in simulation against observed accident rates.

ReSonAte: A Runtime Risk Assessment Framework for Autonomous Systems

Charles Hartsell, Shreyas Ramakrishna, Abhishek Dubey,
Daniel Stojcsics, Nagabhushan Mahadevan, and Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
charles.a.hartsell@vanderbilt.edu

6.1 Introduction

Autonomous Cyber Physical Systems (CPSs)[1] are expected to handle uncertainties and self-manage the system operation in response to problems and increase in risk to system safety. This risk may arise due to distribution shifts (Schwalbe and Schels, 2020), environmental context or failure of software or hardware components (Koopman and Wagner, 2016). Safety Risk Management (SRM) (FAA, 2000) has been a well-known approach used to assess the system's operational risk. It involves design-time activities such as *hazard analysis* for identifying the system's potential hazards, *risk assessment* to identify the risk associated with the identified hazards, and a system-level *assurance case* (Bishop and Bloomfield, 2000) to argue the system's safety. However, the design-time hazard analysis and risk assessment information it uses is inadequate in highly dynamic situations at runtime. To better address the dynamic operating nature of CPSs, dynamic assurance approaches such as runtime certification (Rushby, 2008), dynamic assurance cases (Denney et al., 2015), and modular safety certificates (Schneider and Trapp, 2013) have been proposed. These approaches extend the assurance case to include system monitors whose values are used to update the reasoning strategy at runtime. A prominent approach for using the runtime information has been to design a discrete state space model for the system, identity the risk associated with each possible state transition action, then perform the action with the least risk (Wardziński, 2008).

The effectiveness of dynamic risk estimation has been demonstrated in the avionics (Kurd et al., 2009) and medical (Leite et al., 2018) domains, but these techniques often encounter challenges with state space explosion which may limit their applicability to relatively low-complexity systems. Also, real-time CPSs often have strict timing deadlines in the order of tens of milliseconds requiring any dynamic risk assessment technique to be computationally lightweight. Besides, the dynamic risk assessment technique should also take into consideration the uncertainty introduced by the LECs because of Out-Of-Distribution (OOD) data (Sundar et al., 2020). Assurance monitors (Cai and Koutsoukos, 2020; Sundar et al., 2020) are a type of OOD detector often used to tackle the OOD data problem. The output of these monitors should be considered when computing the dynamic risk.

Recently, there is a growing interest in dynamic risk assessment of autonomous CPS. For example, the authors in (Katrakazas et al., 2019) have used Dynamic Bayesian Networks to incorporate the broader effect

[1]CPS with learning enabled components (LEC)

of spatio-temporal risk gathered from road information on the system's operational risk. To perform dynamic risk assessment of a system with autonomous components, this paper introduces the ReSonAte framework. ReSonAte uses the design-time hazard analysis information to build BTDs which describe potential hazards to the system and how common events may escalate to consequences due to those hazards. The risk posed by these hazards can change dynamically since the frequency of events and effectiveness of hazard controls may change based on the state of the system and environment. To account for these dynamic events at runtime, ReSonAte uses design-time BTD models along with information about the system's current state derived from system monitors (e.g. anomaly detectors, assurance monitors, etc.) and the operating environment (e.g. weather, traffic, etc.) to estimate dynamic hazard rates. The estimated hazard rate can be used for high-level decision making tasks at runtime, to support self-adaptation of CPSs.

The specific contributions of this paper are the following. We present the ReSonAte framework and outline the dynamic risk estimation technique which involves design-time measurement of the conditional relationships between hazard rates and the state of the system and environment. These conditional relationships are then used at run-time along with state observations from multiple sources to dynamically estimate system risk. Further, we describe a process that uses an extended BTD model for estimating the conditional relationships between the effectiveness of hazard control strategies and the state of the system and environment. To improve scalability and reduce the amount of data required, this process considers each control strategy in isolation and composes several single-variate distributions into one complete multi-variate distribution for the control strategy in question. A key contribution is our scenario specific language that enables data collection by specifying prior distributions over threat conditions and environmental conditions and initial conditions of the vehicle.

We implement ReSonAte for an AV example in the CARLA simulator (Dosovitskiy et al., 2017b) and through comprehensive simulations across 600 executions, we show that there is a strong correlation between our risk estimates and eventual vehicular collisions. The dynamic risk calculations on average take only 0.3 milliseconds at runtime in addition to the overhead introduced by the system monitors. Further, we exhibit ReSonAte's generalizability with preliminary results from an UUV example.

6.2 Related Work

System health management (Mahadevan et al., 2011; Srivastava and Schumann, 2011) with model based reasoning have been popularly used for self-adaptation of traditional CPSs. As discussed in (Steinbauer and Wotawa, 2013), a pre-requisite in these approaches has been that the system has knowledge about itself, its objectives, and its operating environments, including the ability to estimate when the adaption should occur. One way to do this is to estimate the operational risk to the system at runtime. Estimating the runtime risk is

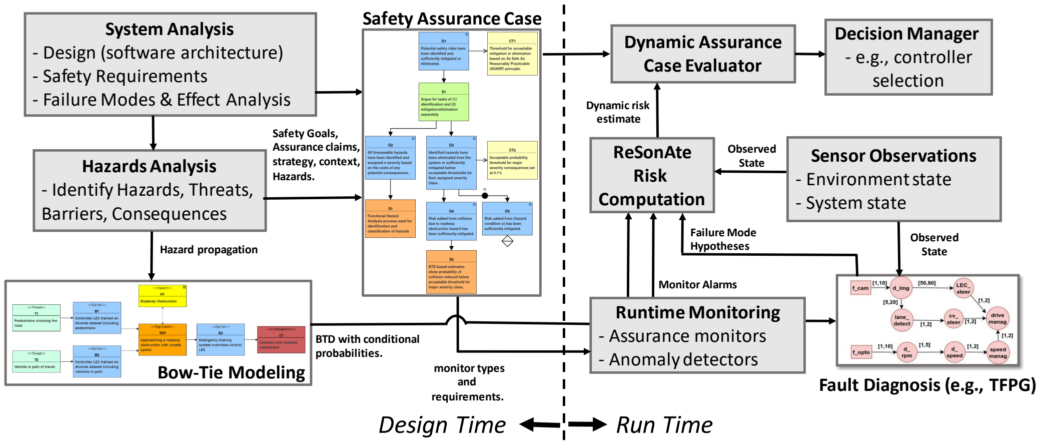

System Analysis
- Design (software architecture)
- Safety Requirements
- Failure Modes & Effect Analysis
Safety Assurance Case
Safety Goals, Assurance claims, strategy, context, Hazards.
Hazards Analysis
- Identify Hazards, Threats, Barriers, Consequences
Hazard propagation
Bow-Tie Modeling
BTD with conditional probabilities.
monitor types and requirements.
Dynamic Assurance Case Evaluator
Decision Manager
- e.g., controller selection
Dynamic risk estimate
ReSonAte Risk Computation
Observed State
Sensor Observations
- Environment state
- System state
Failure Mode Hypotheses
Observed State
Monitor Alarms
Runtime Monitoring
- Assurance monitors
- Anomaly detectors
Fault Diagnosis (e.g., TFPG)
Design Time
Run Time

Figure 6.1: Overview of the steps involved in ReSonAte. Steps performed at design-time/run-time are shown on the left/right.

more difficult in autonomous systems with LECs due to the black box nature of the learning components and their susceptibility to distribution shifts and environmental changes.

Our hypothesis is that we can use BTDs derived from the system information and static assurance cases and use them to compose the anomaly and the threat likelihoods at runtime, including the likelihood for the failure of LECs (Sundar et al., 2020; Cai and Koutsoukos, 2020; Byun and Rayadurgam, 2020; Schwalbe and Schels, 2020). BTDs are graphical models that provide a mechanism to learn the conditional relationships between threat events, hazards, and the success probability of barriers. BTDs have been proactively used for qualitative risk assessment at design-time (Clothier et al., 2015; Williams et al., 2014), and has been recently combined with assurance arguments for operational safety assurance (Denney et al., 2017, 2019).

Although BTDs have been proactively used for risk assessment, there are several limitations in using them for quantitative risk computations, they are: (1) *Static structure* - BTDs have mostly been used as a graphical visualization tool for hazard analysis, and its static structure limits real-data updating which is required for dynamic risk estimation(Khakzad et al., 2012), (2) *Reliance on domain experts* - quantitative risk estimations mostly rely on domain experts to compute the probabilities for the BTD events such as threats, and barriers (Delvosalle et al., 2006), (3) *Data uncertainty* - introduced because of data non-availability or insufficiency, and expert's limited knowledge makes it difficult to compute quality probability estimates (Ferdous et al., 2009).

Several approaches have been proposed to overcome these limitations and make BTDs suitable for quantitative risk assessment. Bayesian techniques are one widely used approach that dynamically learns the BTD structure from design-time data, and updates the conditional probabilities for its events (e.g., threats, barriers) (Badreddine and Ben Amor, 2010; Badreddine and Amor, 2013). Further, (Teimourikia et al., 2017) transforms a BTD into a Bayesian Network where each node of the BTD is modelled as a Bayesian Node. The other approach uses techniques such as fuzzy set and evidence theory to address the data uncertainty problems (Ferdous et al., 2009; Zhang and Guan, 2018). In this approach, the expert's knowledge is translated into numerical quantities that are used in the BTDs. (Vileiniskis and Remenyte-Prescott, 2017) extends BTDs with petri-nets models and monte-carlo simulations to generate data.

Although prior approaches have made BTDs suitable for quantitative risk estimation, the design-time hazard information used for risk estimation is inadequate for CPSs. The main focus of this work is to dynamically estimate risk by fusing design-time information captured in the BTDs with run-time information about the system and the environment. Additionally, we concentrate on generating large-scale simulation data for conditional probability estimation of the BTD events.

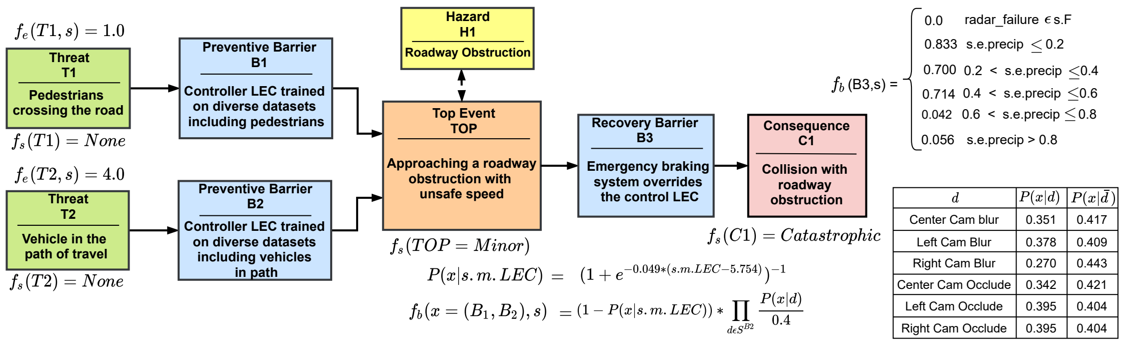
6.3 ReSonAte Framework

The goal of the ReSonAte framework is the dynamic estimation of *risk* based on runtime observations about the state of the system and environment. We define risk as a product of the likelihood and severity of undesirable events, or *consequences*. Fig. 6.1 outlines the ReSonAte workflow which is divided into design-time and run-time steps. The design-time steps include System Analysis, Hazard Analysis, Assurance Case Construction, and BTD Modeling. Sections 6.3.1.1 through 6.3.2 provide background information about each of these well-studied techniques. However, our BTD formalism described in Section 6.3.2 is distinct from existing formalisms with additional model attributes and restrictions for the ReSonAte framework. Sections 6.3.3 through 6.3.5 introduce the risk calculation equations, the conditional probability estimation process, and the dynamic assurance case evaluation method of the ReSonAte framework respectively.

6.3.1 Background

6.3.1.1 System Analysis

System analysis involves the design-time analysis of the system operation, system faults, the available runtime monitors, and the operating environment such as weather (e.g. rain, snow, fog, etc.), traffic, etc. Thereafter, we perform failure mode analysis to identify the possible component faults and their potential impacts on the safety of the system. Fault propagation paths can be described with the use of an appropriate fault modeling language (e.g. Timed Failure Propagation Graph (TFPG) (Misra, 1994)), and run-time monitors for fault identification can be designed as appropriate. Monitors to detect faults in traditional components have been designed in prior work (Mahadevan et al., 2012), but monitors for LECs are often more complex (e.g. assurance monitors (Sundar et al., 2020)). Alarms raised by these monitors at run-time can then be fed into an appropriate fault diagnosis engine (e.g. the TFPG reasoning engine) to isolate particular fault modes


$f_e(T1, s) = 1.0$
Threat T1
Pedestrians crossing the road
$f_s(T1) = None$
$f_e(T2, s) = 4.0$
Threat T2
Vehicle in the path of travel
$f_s(T2) = None$
Preventive Barrier B1
Controller LEC trained on diverse datasets including pedestrians
Preventive Barrier B2
Controller LEC trained on diverse datasets including vehicles in path
Hazard H1
Roadway Obstruction
Top Event TOP
Approaching a roadway obstruction with unsafe speed
$f_s(TOP = Minor)$
Recovery Barrier B3
Emergency braking system overrides the control LEC
Consequence C1
Collision with roadway obstruction
$f_s(C1) = Catastrophic$

$$f_b(\text{B3,s}) = \begin{cases} 0.0 & \text{radar\_failure} \in \text{s.F} \\ 0.833 & \text{s.e.precip} \leq 0.2 \\ 0.700 & 0.2 < \text{s.e.precip} \leq 0.4 \\ 0.714 & 0.4 < \text{s.e.precip} \leq 0.6 \\ 0.042 & 0.6 < \text{s.e.precip} \leq 0.8 \\ 0.056 & \text{s.e.precip} > 0.8 \end{cases}$$

$$P(x|s.m.LEC) = (1 + e^{-0.049*(s.m.LEC-5.754)})^{-1}$$

$$f_b(x = (B_1, B_2), s) = (1 - P(x|s.m.LEC)) * \prod_{d \epsilon S^{B2}} \frac{P(x|d)}{0.4}$$

d	$P(x\|d)$	$P(x\|\bar{d})$
Center Cam blur	0.351	0.417
Left Cam Blur	0.378	0.409
Right Cam Blur	0.270	0.443
Center Cam Occlude	0.342	0.421
Left Cam Occlude	0.395	0.404
Right Cam Occlude	0.395	0.404

Figure 6.2: Example Bow-Tie Diagram for autonomous vehicle example "Roadway Obstruction" hazard created using the ALC Toolchain. Each block includes a brief description and the type of each node is denoted at the top of the block. The equations depicted are described in Section 6.3.2.

that may be present in the system.

6.3.1.2 Hazard Analysis

Hazard analysis involves the identification of events that may lead to hazardous conditions, implementation of barriers to prevent loss of control over hazard conditions or recovery control after a loss, and estimation of the risk posed by the identified hazards. Guidelines for performing hazard analysis are available in a variety of domains including the ISO 26262 standard (ISO, 2018) for the automotive domain and the FAA System Safety Handbook (Federal Aviation Administration, 2019) for the aerospace domain.

6.3.1.3 Assurance Case Construction

An assurance case (Bishop and Bloomfield, 2000) is a structured argument supported by a body of evidence which shows that system goals will be met in a defined operating environment and is often documented using the GSN (Kelly, 1999). Multiple authors have noted the importance of having "hazard analysis and risk reduction arguments" within an assurance case (Haddon-Cave, 2009),(Leveson, 2011b). In this work, we only concentrate on the hazard analysis argument of the assurance case, and earlier works (Matsuno and Yamamoto, 2012; Denney et al., 2015) can be referred for a more comprehensive dynamic assurance case.

6.3.2 Our Contributions: Bow-Tie Formalization & Extensions

Bow-Tie Diagrams are used in ReSonAte as the means of describing hazard propagation paths and the control strategies used to prevent that propagation. BTDs are intended for describing linear propagation and do not have concepts for capturing non-linear causality or complex interactions between events. However, these linear models are sufficient for many hazards commonly encountered by CPSs such as those demonstrated for the example systems described in Section 6.4. To perform dynamic risk estimation, the hazard models must also contain information about the expected rate of threat events and the success probability of barriers, both of which may be conditional on the current conditions. In this section, we present a BTD formalization which includes this conditional information not captured in existing BTD interpretations. Under our formalism, each node has an associated function that is conditional on the state of the system and environment. Before constructing a BTD, we assume the following sets have been identified: all potential hazard classes H, all events E relevant for hazard analysis, barriers B which may either *prevent* a loss of control or *recover* after such a loss has occurred, possible system failure modes FM, and event severity classes SV. We also assume a function f_a which maps each severity class to the maximum acceptable rate of occurrence threshold $f_a : SV \rightarrow \mathbb{R}$ has been defined.

For risk calculation, we define the state of the system and the environment as $S = (F, e, m)$ where:

- F is a set of failure modes currently present in the system. $F \subseteq FM$
- e is an n-tuple describing the current environmental conditions where n is the number of environmental parameters that are relevant for risk calculations and each parameter may be continuous or discrete valued.
- m is a k-tuple containing the outputs of runtime monitors in the system where k is the number of such monitors and each monitor may be continuous or discrete valued.

When modeling the environment, the system developer should choose an appropriate level of abstraction based on which environmental parameters are relevant for risk calculation. More environmental parameters may increase the fidelity of the model but also increases the required effort when determining conditional probabilities for events and barriers. It is not required to consider every environmental parameter for all events and barriers. Instead, each event or barrier may be conditional upon a smaller subset of environmental parameters that have the largest impact on that particular event or barrier. Formally, we define a Bow-Tie Diagram as a tuple $(N, C, f_t, h, f_d, f_b, f_e, f_s)$ where:

- N is a set of nodes where each node represents either an event $e \in E$ or a barrier $b \in B$.
- C represents a set of directed connections such that $C \subseteq N \times N$. For a given connection $c \in C$, $src(c)$ and $dst(c)$ represent the source and destination of c respectively.
- The tuple (N, C) is a DAG representing the temporal ordering of events as well as the barriers which may break this ordering and prevent further propagation of events.
- The function f_t gives the type of each node. $f_t : N \rightarrow \{event, barrier\}$. Using this function, we let $N_e = \{n \in N \mid f_t(n) = event\}$ and $N_b = \{n \in N \mid f_t(n) = barrier\}$. This gives the following properties: $N_e \subseteq E$, $N_b \subseteq B$, $N = N_e \cup N_b$, and $N_e \cap N_b = \emptyset$.

- $h \in H$ is the hazard class associated with the BTD.
- The function f_d gives a textual description of each node. $f_d : N \rightarrow string$
- f_b is a probability function conditional on the state of the system and environment defined for each barrier node. This function represents the probability that the barrier will successfully prevent further event propagation. $f_b : N_b \times S \rightarrow [0, 1]$.
- f_e is a function conditional on the state of the system and environment defined for each event node. It gives the expected frequency of the event over a fixed period. The values of this function must be specified for *threat* events (i.e. root events with no preceding input events), but can be calculated for subsequent events in the diagram as discussed in Section 6.3.3. $f_e : N_e \times S \rightarrow [0, \inf)$.
- A function f_s which maps each event to the appropriate severity class. $f_s : N_E \rightarrow SV$

BTDs are often constructed in a chained manner where a consequence event in one section of the BTD may serve as a threat or top event in a subsequent section of the same BTD. However, such a chained BTD can be broken into multiple, single-scope BTDs where each event has one unique type (i.e. threat, top event, or consequence). As a simplification, ReSonAte operates only on these single-scope BTDs which satisfy the additional restrictions listed below. Note that the symbol $\Rightarrow$ is used to denote precedence in the graph. That is, given two nodes $a, b \in N$, then $a \Rightarrow b$ states that a precedes b in the BTD, but this does not necessitate a direct edge such that $a \rightarrow b$. Instead, there may be any number of intermediate nodes $c_i \in N$ such that $a \rightarrow c_1 \rightarrow c_2 \rightarrow ... \rightarrow c_n \rightarrow b$.

- Exactly one event must be designated as the *top event*, denoted e_{top}.
- There must be at least one *threat*. i.e. $\exists\, t \in N_e \mid t \Rightarrow e_{top} \land \nexists\, n \in N \mid n \Rightarrow t$. We denote the set of all threat events as N_t.
- There must be at least one *consequence*. i.e. $\exists\, c \in N_e \mid e_{top} \Rightarrow c \land \nexists\, n \in N \mid c \Rightarrow n$. We denote the set of all consequence events as N_c.
- All events must be a threat, a top event, or a consequence. i.e. No intermediate events.
- All *barriers* must lie between a threat and the top event, or between the top event and a consequence. i.e. $\forall\, b \in N_b$ either $t \Rightarrow b \Rightarrow e_{top}$ or $e_{top} \Rightarrow b \Rightarrow c$
- No branching or joining of the graph is allowed, except for at the top event.

For the example BTD shown in Fig. 6.2, the type of each event is denoted on the top of the block and we can define the sets $N_e = \{T1, T2, TOP, C1\}$ and $N_b = \{B1, B2, B3\}$. For each event $e \in N_e$, the associated severity class is given by the function $f_s(e)$ which is shown under each event block. Each of the threats, $T1$ and $T2$, have been assigned to the "None" severity class because these events are considered to be common occurrences that do not result in any safety violation by themselves. The top event TOP has been assigned to the "Minor" severity class since this event can still be mitigated before a safety violation occurs but mitigation requires the Automatic Emergency Braking System (AEBS) to override the primary LEC controller. Finally, the consequence $C1$ has been assigned to the "Catastrophic" severity class since this event is a safety violation and may result in significant damage to the system or environment.

The conditional functions f_e and f_b are shown near their respective nodes in Fig. 6.2. The functions $f_e(T1, \mathbf{s})$ and $f_e(T2, \mathbf{s})$ give the expected frequency of threats $T1$ and $T2$ in units of expected number of occurrences per minute, and their values were measured for our simulator configuration described in Section 6.4. The probability function for barriers $B1$ and $B2$, f_b(x = (B1,B2), **s**), shows how these barriers are dependent on both the continuous-valued output from the assurance monitor and on the binary state of other monitors. A sigmoid function, $P(x|\mathbf{s}.m.LEC)$, is used to capture the conditional relationship with the assurance monitor output. Finally, $f_b(B3, \mathbf{s})$ shows how barrier $B3$ is less likely to succeed as the precipitation increases and will not function in the case of a radar failure. Section 6.3.3 explains the generic process for conditional probability estimation and Section 6.4.1.4 provides more detail on the functions in this BTD.

6.3.3 Run-time Risk Computation

Each BTD includes functions describing the conditional frequency for all *threat* events and the conditional probability of success for all *barrier* nodes. However, the likelihood of each *consequence* is necessary to estimate the overall level of risk for the system. These probabilities can be calculated by the propagation of the initial threat rates through the BTD. When a particular event occurs, barrier nodes reduce the probability that the event will continue to propagate through the BTD based on the following equation:

$$R(e_2|s) = R(e_1|s)P(\overline{b_1} \wedge \overline{b_2} \wedge ... \wedge \overline{b_n}|s)$$

$$assume\ a \perp b\ \forall\ a, b \in \{b_1, b_2, ..., b_n\} \mid a \neq b$$

$$R(e_2|s) = R(e_1|s)[\Pi_{i=1}^{n} P(\overline{b_i}|s)] \tag{6.1}$$

where $e_1, e_2 \in N_e$ and $b_i \in N_b$ such that $e_1 \rightarrow b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_n \rightarrow e_2$. $R(e_i|s)$ represents the frequency of event e_i given state s. Eq. (6.1) makes the assumption that no barriers share any common failure modes and that the effectiveness of each barrier is independent from the outcome of other barriers. Similarly, $P(\overline{b_i}|s)$ represents the probability that barrier b_i will fail to prevent event propagation given state s, i.e. $P(\overline{b_i}|s)$ = $1 - P(b_i|s)$. Letting S represent the set of all states we are concerned with, then we can calculate the overall frequency of e_2 as $R(e_2) = \Sigma_{s \in S}[R(e_2|s)P(s)]$ where $P(s)$ is the probability of each particular state $s \in S$. As discussed in the BTD formalization in Section 6.3.2, no joining or splitting of paths is allowed in a BTD with the exception of the top event e_{top}. We treat the top event as a summation operation for all incoming edges, i.e. any threat event may independently cause a top event if the associated barriers are unsuccessful. All outgoing edges from the top event are treated as independent causal chains, i.e. any potential consequence may occur from a top event, independent of other consequences in the BTD. The probability for the top event can be calculated as:

$$\begin{aligned} R(t^{top}) &= \Sigma_{s \in S}[R(t|s)P(s)[\Pi_{i=1}^{n} P(\overline{b_i}|s)]] \\ R(e_{top}) &= \Sigma_{t \in N_t} R(t^{top}) \end{aligned} \tag{6.2}$$

where $R(e_{top})$ is the frequency for the top event and $P(t^{top})$ represents the contribution of each threat t to this rate after passing through any intermediate prevention barriers $b_i \in N_b$ such that $t_i \rightarrow b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_n \rightarrow e_{top}$. Finally, the probability of each consequence can be calculated with:

$$R(c_i) = R(e_{top})\Sigma_{s \in S}[P(s)[\Pi_{i=1}^{n} P(\overline{b_i}|s)]] \tag{6.3}$$

where $R(c_i)$ is the frequency of consequence $c_i \in N_c$ after passing through any recovery barriers $b_i \in N_b$ such that $e_{top} \rightarrow b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_n \rightarrow c_i$.

If the state is not known uniquely, then each potential state $\mathbf{s} \in S$ must be enumerated and a probability function $P(\mathbf{s})$ must be assigned such that $\Sigma_{\mathbf{s} \in S} P(\mathbf{s}) = 1$. At design-time when only the expected distributions of the state variables are known, this state probability function is typically calculated as a product of the probability mass functions (or probability density functions for continuous variables) of each individual state variable. Recall that for ReSonAte the state S is restricted to only those variables which have a conditional impact on the functions contained in the BTDs and thus not all state variables describing the system must be considered in this calculation. When the system is deployed at run-time, observations about the current state can be used to refine the set of prior probabilities $P(s)$ and dynamically calculate current risk values. The equations outlined here use a discrete treatment of probability, but continuous distributions can be used as well where summation operations are replaced by an appropriate integration. If the state can be identified uniquely to a particular state $\mathbf{s}_0 \in S$, then we can assign $P(\mathbf{s}_0) = 1$ and simplify the risk equations. For example, we could calculate the rate of occurrence for events TOP and $C1$ which are part of the BTD B shown in Fig. 6.2 using the following equations:

$$\begin{aligned} R(TOP|s_0) = B.f_e(T1, s_0) * (1 - B.f_b(B1, s_0)) \\ + B.f_e(T2, s_0) * (1 - B.f_b(B2, s_0)) \end{aligned}$$

$$R(C1|s_0) = R(TOP|s_0) * [1 - B.f_b(B3, s_0)] \tag{6.4}$$

6.3.4 Estimating Conditional Relationships

6.3.4.1 Conditional Relationships

In ReSonAte, both the rate of occurrence of events and the effectiveness of barriers may be conditionally dependent on the state including system failure modes, runtime monitor values, and environmental conditions. For each of these categories, it is necessary to identify the factors which should be examined for their impact on this conditional relationship. A failure analysis should be performed using an appropriate failure modeling language as discussed in Section 6.3.1.1. Similarly, the environmental parameters relevant to the system must be identified and the operating environment defined in terms of bounds and expected distributions of these parameters. We assume the environmental conditions are known uniquely and provided to ReSonAte. Monitor values that may impact the conditional relationships should also be identified.

For some nodes in a BTD it may be possible to analytically derive the appropriate conditional relationship with each state variable, but often this relationship must be inferred from data. In this section, we consider a generic threat to the top event chain with a single barrier described as $t \rightarrow b \rightarrow e_{top}$. The contribution to the rate of the top event e_{top} from this singular threat t with a single barrier b is shown in Eq. (6.5). Normally, multiple threats can lead to the top event and the contribution of all threats must be considered as described in Eq. (6.2). However, if all other threat conditions can be eliminated, then the top event may only occur as a result of threat t. In this case, the probability of success for barrier b can be calculated as a function of the ratio of frequencies between the top event and threat t shown in Eq. (6.6). This approach can also be used for threats with multiple associated barriers by considering each barrier in isolation since individual barriers are assumed to be independent in Eq. (6.1).

$$R(e_{top}^{t}|\mathbf{s}) = R(t|\mathbf{s}) * (1 - P(b|\mathbf{s})) \tag{6.5}$$

$$R(t_j|\mathbf{s}) = 0 \, \forall \, t_j \in N_t \mid t_j \neq t \rightarrow R(e_{top}|\mathbf{s}) = R(e_{top}^{t}|\mathbf{s})$$

$$P(b|\mathbf{s}) = 1 - [R(e_{top}|\mathbf{s})/R(t|\mathbf{s})] \tag{6.6}$$

We isolate individual threats in simulation using a custom Scenario Description Language (SDL), described in Section 6.3.4.2, to generate scenarios where only the one threat of interest is allowed to occur and all other possible threats are eliminated. Each time the threat t is encountered, the top event e_{top} may either occur or not occur. If the top event does not occur, then the associated barrier b was successful - i.e. prevented hazard propagation along this path. Otherwise, the barrier was unsuccessful. Since the occurrence of a threat is a discrete event that results in a boolean outcome (i.e. top event does/does not occur), the barrier effectiveness can be modeled as a conditional Binomial distribution where the probability of barrier success is dependent on the ratio of top event frequency to threat frequency as shown in Eq. (6.6).

For each barrier b in the BTD, any state variables which are likely to impact the conditional frequency or probability of that node should be identified, and we denote this reduced set of state variables as S^b. For discrete state variables (e.g., presence or absence of a particular fault condition, urban/rural/suburban environment, etc.), this ratio can be estimated using Laplace's rule of succession (Zabell, 1989) as shown in Eq. (6.7) where $n_{s_i=a}$ is the number of scenes where the state variable s_i had the desired value a and $k_{s_i=a}$ is the number of such scenes where the top event also occurred. This equation can be applied for each of the possible values of the state variable s_i to estimate the discrete probability distribution $P(b|s_i)$, and the process can then be repeated for each relevant state variable $s_i \in S^c$. Eq. (6.8) can be used to fuse the estimated distributions of each individual state variable into one multivariate distribution $P(b|\mathbf{s})$. Similar to Naive Bayes classifiers, this equation assumes each of the state variables s_i are mutually independent conditional on the success of barrier b. If a stronger assumption is used that the state variables in S^b are mutually independent, then the term $\Pi_{j=0}^{m}[P(s_j)]P(s_0 s_1 ... s_m)^{-1}$ reduces to 1 as was the case for all of the probabilities estimated in our examples. For each continuous state variable s_j (e.g. output of an assurance monitor), maximum likelihood estimation was used in place of Eq. (6.7) to estimate $P(b|s_j)$. Similarly, Eq. (6.8) can be revised for continuous values by replacing the probability mass functions $P(s_j)$ with probability density functions $p(s_j)$.

```
scene sample {
    type string
    type int
    entity town_description{
        id:string
        map:string  }
    entity weather_description{
        cloudiness: uniform
        precipitation: uniform
        precipitation_deposits: uniform  }
    entity uniform{
        low: int
        high: int  }
    }
```

Figure 6.3: This listing shows a fragment of a CARLA scene description that was generated using our SDL written in textX meta language.

$$P(b|s_i = a) = 1 - \frac{k_{s_i=a} + 1}{n_{s_i=a} + 2} \tag{6.7}$$

$$P(b|\mathbf{s}) = \frac{P(s_0, s_1, ..., s_m|b)P(b)}{P(s_0, s_1, ..., s_m)}$$

$$assume\ s_j \perp s_k \mid b \,\forall\, s_j, s_k \in S^b \mid s_j \neq s_k$$

$$P(b|s_0, s_1, ..., s_m) = \frac{P(b)\Pi_{j=0}^{m} P(s_j|b)}{P(s_0, s_1, ..., s_m)}$$

$$P(b|s_0, s_1, ..., s_m) = \frac{\Pi_{j=0}^{m} P(b|s_j)P(s_j)}{P(b)^{m-1}P(s_0 s_1 ... s_m)} \tag{6.8}$$

While the conditional probability estimation process is outlined here for prevention barriers, it may be modified for recovery barriers by replacing occurrences of any threats *t* with the top event e_{top} and replacing occurrences of the top event with each consequence of interest.

6.3.4.2 Scenario Description Language

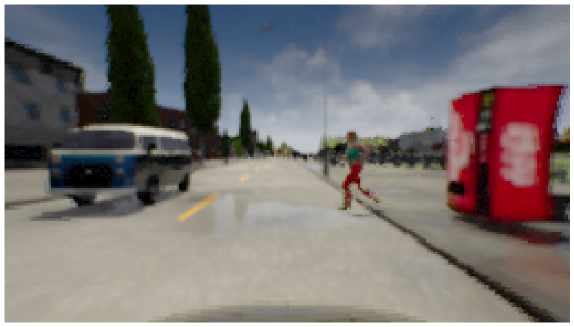
Description and generation of scenarios that cover the full range of expected operating environments is an important aspect for the design of CPS. Several domain-specific SDLs such as Scenic (Fremont et al., 2019) and MSDL (Foretellix, 2021) with probabilistic scene generation capabilities are available. While these languages have powerful scene generation capabilities, they are targeted specifically at the automotive domain. We have developed a simplified SDL using the textX (Dejanović et al., 2017) meta language to generate varied scenes for multiple domains including our AV and UUV systems.

A fragment of the scene description for the CARLA AV example is shown in Fig. 6.3. A scene S = $\{e_1,e_2,...,e_i\}$ is described as a collection of entities (or set points), with each entity representing information either about the ego vehicle (e.g., type, route, etc.), or the operating environment (e.g., weather, obstacle). Further, each of these entities has parameters whose value can be sampled using techniques such as Markov chain Monte Carlo to generate different scenes in the simulation space. The larger the number of sampling, the wider is the simulation space coverage. For the AV example, our scene was defined as S = {town_description, weather_description, av_route}. While the parameters of town_description and av_route remained fixed, the parameters of weather_description such as cloud, precipitation, and precipitation deposits were randomly sampled to take a value in [0,100]. We generated 46 different CARLA scenes by randomly sampling the weather parameters, a few of which were used for estimating the conditional relationships.

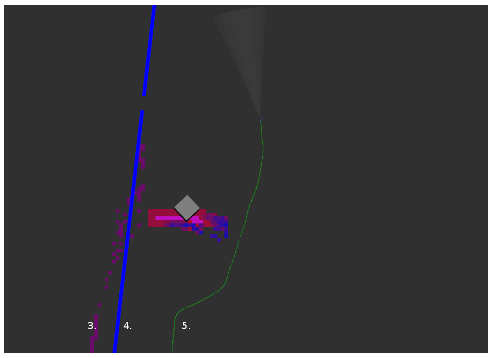
Currently we perform unbiased sampling to generate each scenario, then use the resulting unbiased dataset for the probability calculations described in Section 6.3.4. This approach proved sufficient for the example systems described in Section 6.4. However, these systems are prototypes where consequences occur relatively often. For more refined production systems, consequences do not typically occur under nominal operation but instead are often the result of rare combinations of adverse operating conditions and/or system failure modes. This is an example of the long tails problem where the probability of observing this undesired system behavior is low if unbiased random sampling is used. In future work, guided sampling of the state space (e.g. (Karunakaran et al., 2020)) can be used to better observe these rare events and perform conditional probability estimation with the resulting biased dataset.

6.3.5 Dynamic Assurance Case Evaluation

Safety Risk Management (FAA, 2000) is a common technique used in the system safety assurance process which involves identification of potential hazards, analysis of the risks posed by those hazards, and reduction of these risks to acceptable levels. The amount of risk remaining after risk control strategies have been implemented is known as the *residual risk*. An appropriate assurance argument pattern, such as the ALARP pattern outlined by Kelly (Kelly, 1999), is often used to document the means used for risk reduction and show that the estimated levels of residual risk are within tolerable bounds. With traditional SRM techniques, residual risk estimates are static results of design-time analysis techniques. However, using the risk estimated by ReSonAte the residual risk for each hazard can be updated dynamically at run-time and the associated goal in the assurance argument can be invalidated if the risk exceeds a predefined threshold. When this risk threshold is violated, contingency plans can be enacted to place the system in a safe state. For example, stopping the AV or surfacing the UUV are simple contingency actions used for the example systems described in Section 6.4. The risk scores produced by ReSonAte may also be used in assurance case adaptation techniques such as Dynamic Safety Cases (Denney et al., 2015) or ENTRUST (Calinescu et al., 2017).



(a) Image from one of three forward-looking cameras used for navigation in CARLA simulator, shown at the actual resolution.



(b) Top-down view of BlueROV2.

Figure 6.4: Screenshots from each autonomous system simulation. Fig. 6.4a shows an image from the forward-looking camera of the AV as it navigates through the city. Fig. 6.4b shows a top-down view of the UUV where the vehicle (trajectory shown as a green line) is inspecting a pipeline (thick blue line) until an obstacle (grey box) is detected by the forward-looking sonar and the vehicle performs an avoidance maneuver.

Machine

6.4 Evaluation

We evaluate ReSonAte using an AV example in the CARLA simulator (Dosovitskiy et al., 2017b) and show its generalizability with preliminary results from an UUV example (Blue Robotics, 2016). The experiments[2] in this section were performed on a desktop with AMD Ryzen Threadripper 16-Core Processor, 4 NVIDIA Titan Xp GPU's and 128 GiB memory.

6.4.1 Autonomous Ground Vehicle

6.4.1.1 System Overview

Our first example system is an autonomous car which must safely navigate through an urban environment while avoiding collisions with pedestrians and other vehicles in a variety of environmental and component failure conditions. The architecture of our AV, shown in Fig. 6.5, relies on a total of 9 sensors including two forward-looking radars, three forward-looking cameras, a Global Positioning System (GPS) receiver, an Inertial Measurement Unit (IMU), and a speedometer. The "Navigation" LEC, adapted from previous work (Chen et al., 2020), produces waypoints for the desired position and velocity of the vehicle at a sub-meter granularity using a neural network for image processing along with higher-level information about the desired route provided by a map. These waypoints are passed to the "Motion Estimator" which computes throttle and steering angle error between the current and desired waypoints. The Motion Estimator also serves as a supervisory controller which will override the primary Navigation component if an alarm is sent by the AEBS safeguard component. This AEBS component will raise an alarm when the vehicle safe stopping

[2]source code to replicate the CARLA AV experiments can be found at: https://github.com/scope-lab-vu/Resonate

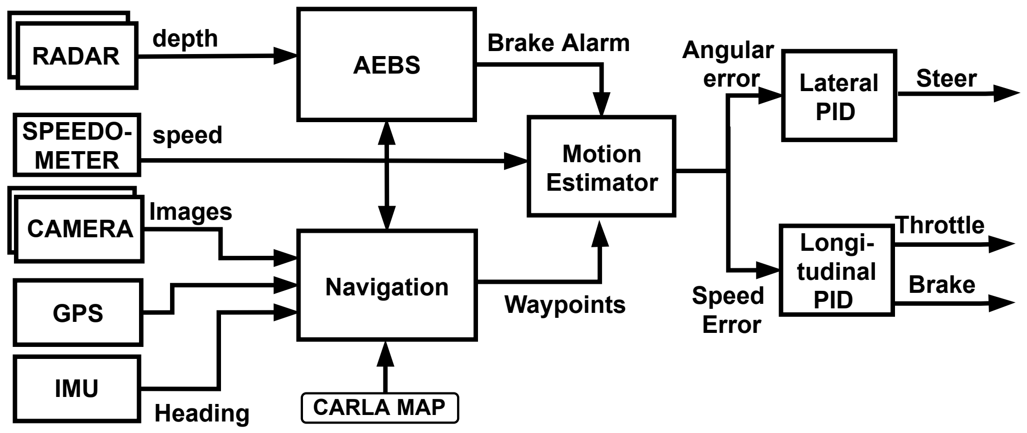


Figure 6.5: A block diagram of our AV example in CARLA simulation.

distance, estimated based on the current vehicle speed, exceeds the distance returned by the radars indicating that the AV is approaching an object at an unsafe speed. Finally, the output of the Motion Estimator is sent to two PID controllers which generate appropriate steering, throttle, and brake control signals.

6.4.1.2 System Analysis

We start with the analysis of the AV system and its operating environment (e.g., weather, traffic, etc). As the AV primarily uses a perception LEC, parameters such as weather conditions (e.g., cloud spread, precipitation level, and precipitation deposit level), high brightness, and camera related faults such as blur and occlusion influenced the control actions generated by the LEC controller. To detect the camera faults and adverse operating conditions, we have designed several monitors. Each of the three cameras is equipped with OpenCV based blur and occlusion detectors to detect image distortions. The blur detector uses the variance of the laplacian (Pech-Pacheco et al., 2000) to quantify the level of blur in the image where a high variance indicates that the image is not blurred, while a low value ($<$30) indicates the image is blurred. The occlusion detector is designed to detect continuous black pixels in the images, with the hypothesis that an occluded image will have a higher percentage of connected black pixels. In this work, a large value ($>$15%) of connected black pixels indicated an occlusion. The blur detector has an F1-score of 99% and the blur detector has an F1-score of 97% in detecting the respective anomalies.

Additionally, we leverage our previous work (Sundar et al., 2020) to design a reconstruction based β-VAE assurance monitor for identifying changes in the operating scenes such as high brightness. The β-VAE network has four convolutional layers $32/64/128/256$ with (5x5) filters and (2x2) max-pooling followed by four fully connected layers with 2048, 1000, and 250 neurons. A symmetric deconvolutional decoder structure is used as a decoder, and the network uses hyperparameters of β=1.2 and a latent space of size=100. This network is trained for 150 epochs on 6000 images from CARLA scenes of both clear and rainy scenarios.

The reconstruction mean square error of the β-VAE is used with Inductive Conformal Prediction (Shafer and Vovk, 2008) and power martingale (Fedorova et al., 2012) to compute a martingale value. The assurance monitor has an F1 score of 98% in detecting operating scenes with high brightness. Further, TFPG models for the identified camera faults were constructed, but the TFPG reasoning engine was not used since the available monitors were sufficient to uniquely isolate fault conditions without any additional diagnostic procedures.

6.4.1.3 Hazard Analysis and BTD Modeling

For our AV example, we consider a single hazard of a potential collision with roadway obstructions. The potential threats for this hazard were identified to be pedestrians crossing the road (T1), and other vehicles in the AV's path of travel (T2). The top event was defined as a condition where the AV is approaching a roadway obstruction with an unsafe speed. The primary LEC is trained to safely navigate in the presence of either of these threats and served as the first hazard control strategy for both threat conditions, denoted by prevention barriers B1 and B2. Finally, the AEBS served as a secondary hazard control strategy denoted by the recovery barrier B3. The BTD shown in Fig. 6.2 was constructed based on these identified events and control strategies. For full-scale systems, the BTD construction process will usually result in the identification of a large number of events and barriers modeled across many BTDs. For our example, we have restricted our analysis to these few events and barriers contained in a single BTD.

6.4.1.4 Conditional Relationships

Here we consider the barrier node B2 in Fig. 6.2 as an example for the probability calculation technique described in Section 6.3.4. Barrier B2 is part of the event chain $T2 \rightarrow B2 \rightarrow TOP$ and describes the primary control system's ability to recognize when it is approaching a slow-moving or stopped vehicle in our lane of travel (T2) too quickly and slow down before control of the roadway obstruction hazard (H1) is lost. To isolate barrier B2, simulation scenes were generated using our SDL where T2 was the only threat condition present but all other system and environmental parameters were free to vary. It is important for these scenes to cover the range of expected system states and environmental conditions since the resulting dataset is used to estimate the conditional probability of success for barrier B2. A total of 300 simulation scenarios were used for estimating the probability of barrier B2. As a convenience for our example, the threat (T2) was set to occur once during each scene, regardless of other state parameters, which allows the denominator of this ratio to be set as $R(T2|\mathbf{s}) = 1$ occurrence per scene.

As perception LEC is the primary controller, the success rate of barriers B1 and B2 will be dependent on the image quality. The image quality will in turn be dependent on image blurriness, occlusion, and environmental conditions. While the level of blur and occlusion for each camera is a configurable simulation parameter, our example system is not provided this information and instead relies on blur and occlusion detectors for each of the three cameras as discussed in Section 6.4.1.2. Each detector provides a boolean output indicating if the level of blur or occlusion exceeds a fixed threshold. The primary LEC is also susceptible to OOD data, and an assurance monitor was trained for this LEC to detect such conditions.

This results in barrier $B2$ being dependent on a total of 7 state-variables described as the set S^{B2}. For the 6 boolean variables, Laplace's rule of succession shown in Eq. (6.7) was applied to the simulation dataset resulting in probability table in the lower-right section of Fig. 6.2. A sigmoid function was chosen to model the conditional relationship with the continuous assurance monitor output. Maximum likelihood estimation was used to produce the function $P(x|\mathbf{s}.m.LEC)$ shown in Fig. 6.2 where $\mathbf{s}.m.LEC$ represents the output of the LEC assurance monitor. Each of these single variable functions was combined into the multivariate conditional probability distribution $f_b(B2, \mathbf{s})$ using Eq. (6.8). Note that the LEC was observed to be similarly effective in identifying pedestrians and vehicles, and a simplifying assumption was made to use the same conditional probability function for both barriers B1 and B2 given by function f_b(x = (B1,B2), $\mathbf{s}$) in Fig. 6.2.

The AEBS described by barrier B3 is independent of the camera images and relies on the forward looking radar. The level of noise in our simulated radar sensor increased with increasing precipitation levels, indicating that the effectiveness of barrier B3 would likely decrease as precipitation increased. Also, failure of the radar sensor may occur on a random basis which reduces the effectiveness of the AEBS to zero. A similar conditional estimation process as used for B2 was applied here to calculate the function $f_b(B3, \mathbf{s})$ shown in Fig. 6.2.

6.4.1.5 Results

To validate the ReSonAte framework, the AV was tasked to navigate 46 different validation scenes that were generated using our SDL. In these scenes, the weather_description parameters of cloud (c), precipitation (p), and precipitation deposits (d) were varied in the range [0,100]. Other adversities were synthetically introduced using OpenCV including increased image brightness, camera occlusion (15%-30% black pixels), and camera blur (using 10x10 Gaussian filters). During each simulation, the ReSonAte's risk calculations continuously estimate the hazard (or collision) rate $h(t)$ based on changing environmental conditions, presence of faults, and outputs from the runtime monitors. The frequency of the risk estimation can be selected either based on the vehicle's speed, environmental changes, or available compute resources. In our experiments, we estimate the risk every inference cycle.

Fig. 6.6 shows the estimated collision rate and the likelihood of collision as the AV navigates 2 validation scenes. The occurrence of a collision can be described as a random variable following a Poisson distribution where the estimated collision rate is the expected value λ. The likelihood of collision can then be computed

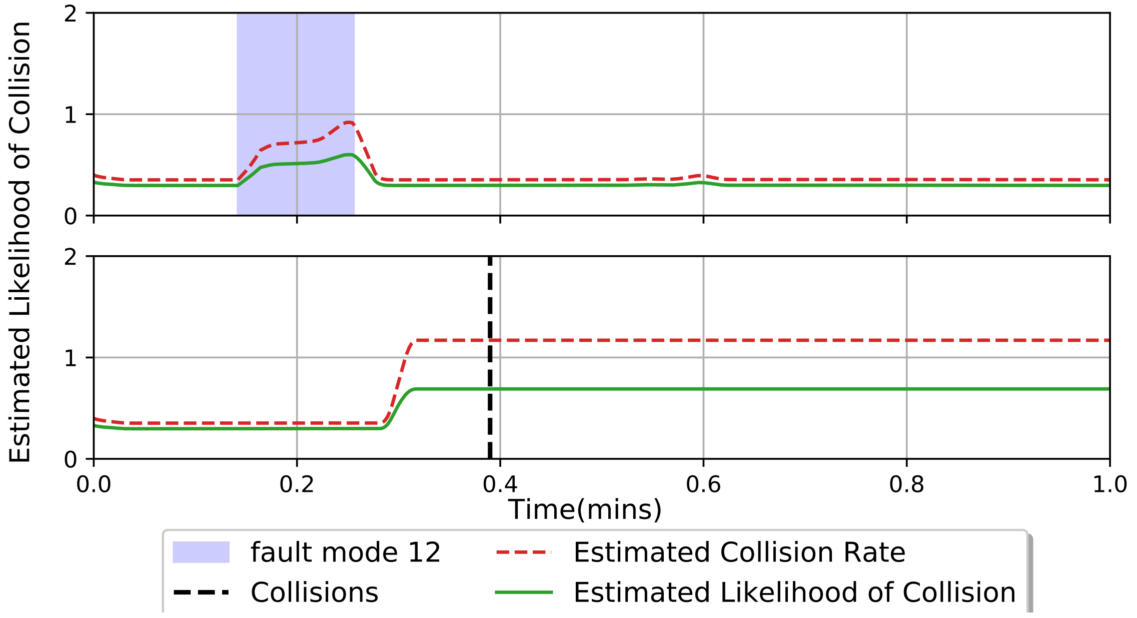


Figure 6.6: ReSonAte estimated collision rate and likelihood of collision for 2 validation scenes. (Top) Scene1 - nominal scene with good weather and an intermittent occlusion fault for left and center cameras. (Bottom) Scene2 - initially nominal scene until 27 seconds when image brightness is increased. A collision occurs at 38 seconds denoted by the vertical dotted line.

as $(1 - e^{-\lambda \cdot t})$, where λ is the estimated collision rate and t is the operation time which is fixed to 1 minute in our experiments.

Fig. 6.7 shows the estimated average collision rate plotted against the observed collisions for 6 validation scenes described in the figure caption. The estimated average collision rate is calculated as $\frac{\int_{T_1}^{T_2} h(t)\,dt}{T_2 - T_1}$, where, h(t) is the estimated collision rate, T_1 = 0 and T_2 = 1 minute for our simulations. A moving average is used to smooth the estimated collision rate and a window size of 20 was selected to balance the desired smoothing against the delay incurred by the moving average. An overall trend in the plot shows a strong correlation between the actual and the estimated collisions. Also, a visible trend is that the collision rate changes with the weather patterns, increasing for adverse weather conditions (S5-S6). However, the estimated risk tends to slightly overestimate when the collision rate is greater than 1.0.

Further, each of the 608 data points shown in Fig. 6.8a represents the outcome of one simulated scene with the actual number of collisions plotted against the average collision rate estimated by ReSonAte. Since the occurrence of a collision is a probabilistic event, there is significant variation in the actual number of collisions observed in each scene. Using our dynamic rate calculation approach, the rate parameter λ of the Poisson distribution changes for each scene as shown on the x-axis of Fig. 6.8a. Maximum likelihood analysis was used to compare our dynamic approach against a static, design-time collision rate estimate where λ is fixed for all scenes. The observed average collision rate across all scenes was found to be 0.829 collisions per

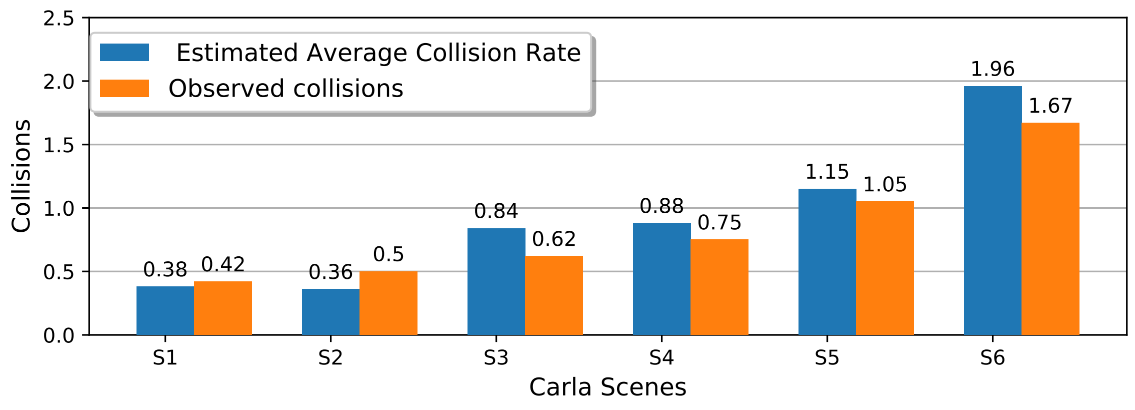
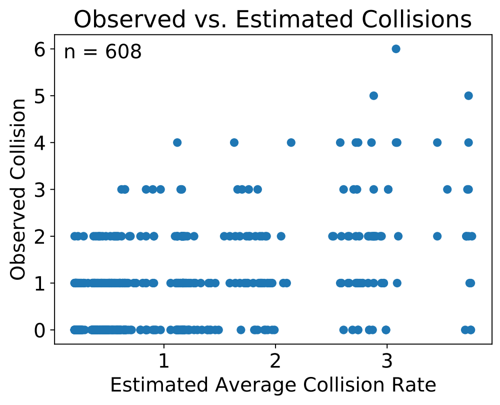
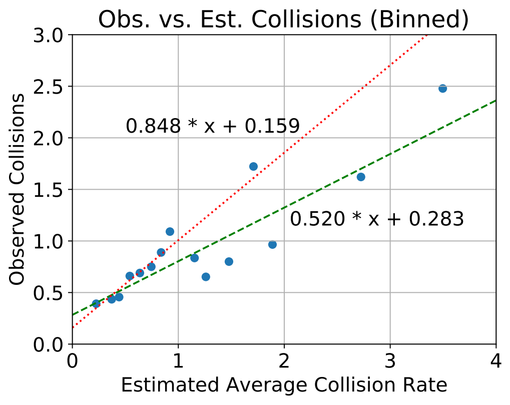


Figure 6.7: ReSonAte estimated average collision rate vs. observed collisions compared across 6 validation scenes. The results are averaged across 20 simulation runs for each scene. Each subsequent scene represents increasingly adverse weather and component failure conditions.



(a) Validation data scatter plot.



(b) Binned validation data.

Figure 6.8: Results from validation scenes for the ReSonAte framework. Each data point shown in Fig. 6.8a represents the outcome of one simulated scene with the actual number of collisions observed plotted against the estimated average collision rate. These same data points have been divided into bins and averaged in Fig. 6.8b, along with two least-squares fit trend lines.

minute. Using this static λ value gave a log likelihood of -740.7 while the dynamically updated λ resulted in a log likelihood of -709.8 for a likelihood ratio of 30.9 in favor of our dynamic approach. Note that the static collision rate used here is a posterior estimate calculated from the true number of collisions observed. Any static risk estimate made without this post facto knowledge would result in a lower likelihood value and further increase the gap between the dynamic and static approaches.

To better show the correlation between estimated and actual collisions, the same data points have been divided into bins and averaged in Fig. 6.8b along with two least-squares fit trend lines. The dashed green trend line shows a linear fit to the complete data set while the dotted red line shows a linear fit to only those data points where the average estimated collision rate was less than or equal to 1.0. Both trend lines show

System Configuration	GPU		CPU		Execution Time (s)
	Util (%)	Mem (%)	Util (%)	Mem (%)	
LEC only	14.3	47.2	17.6	10.4	0.024
LEC + Runtime Monitors	14.7	86.5	18.5	10.5	0.217
LEC + Monitors + ReSonAte	16.1	87.0	18.7	10.4	0.218

Table 6.1: Resource requirements and execution times for different configurations of the system. Average values computed across 20 simulation runs.

a strong positive correlation between the estimated collision rate and the number of observed collisions, but the dotted red line more closely resembles the desired 1-to-1 correspondence between estimated and actual collisions (i.e. slope of 1). These results indicate that our dynamic risk calculation tends to over-estimate when the estimated collision rate is greater than 1.0.

Table 6.1 shows the resource requirements and execution times for system with different configurations. As seen from the shaded columns, the additional sensors, and system monitors, particularly the resource intensive β-VAE monitor, increases the GPU memory used by $\sim$ 40% and execution time by $\sim$ 0.2 seconds. However, the ReSonAte risk calculations require minimal computational resources taking only 0.3 milliseconds.

6.4.2 Unmanned Underwater Vehicle

In this section we discuss the primary results of applying ReSonAte to a UUV testbed based on the BlueROV2 (Blue Robotics, 2016) vehicle. The testbed is built on the ROS middleware (Quigley et al., 2009) and simulated using the Gazebo simulator environment (Koenig and Howard, 2004) with the UUV Simulator (Manhães et al., 2016) extensions.

6.4.2.1 System Overview

In this example, the UUV was tasked to track a pipeline while avoiding static obstacles (e.g., plants, rocks, etc.) as shown in Fig. 6.4b. The UUV is equipped with 6 thrusters, a forward looking sonar (FLS), 2 side looking sonars (SLS), an IMU, a GPS, an altimeter, an odometer, and a pressure sensor. The vehicle has several ROS nodes for performing pipe tracking, obstacle avoidance, degradation detection, contingency planning, and control. Additional contingency management features such as thruster reallocation, return-to-home, and resurfacing are available. The UUV uses the SLS along with the FLS, odometry, and altimeter to generate the HSD commands for pipe tracking and obstacle avoidance. The degradation detector is an LEC which employs a feed-forward neural network to detect possible thruster degradation based on thruster efficiency information. This LEC sends information to the contingency manager including the identifier of the degraded thruster, level of degradation, and a value measuring confidence in the predictions. The contingency manager may then perform a thruster reallocation (i.e. adjustment of the vehicle control law) if necessary to adapt to any degradation. A ROS node implementation of ReSonAte was used to dynamically estimate and publish the likelihood of a collision.

6.4.2.2 Hazard Analysis and BTD Modeling

Using the system requirements and its operating conditions we identified UUV operating in presence of static obstacles as the hazard condition which could result in the UUV's collision. The static obstacle which appears at a distance less than the desired separation distance of 30 meters is considered to be a threat in this example. Further, the static obstacle appearing at a distance less than a minimum separation distance of 5 meters is considered to be a TOP event for the BTD. This information was used to outline a BTD for the UUV example.

6.4.2.3 Conditional Relationships

The probability estimation method described in Section 6.3.4 was used to compute the conditional probabilities for the BTD. We used data from 350 simulation scenarios for the conditional probability calculations. These simulations were generated using several scenes generated by varying parameters such as the obstacle sizes (cubes with lengths (0.5,1,2,5,10) meters), the obstacle spawn distance (value in range [5-30] meters), and random thruster failures that were synthetically introduced by varying the thruster1 efficiency in the range [0-60].

6.4.2.4 Results

Fig. 6.9 shows the ReSonAte estimated likelihood of collision. The efficiency of thruster 1 degrades to 60% at 45 seconds, and the estimated likelihood of collision increases to 0.25. Soon after, the contingency manager performs a thruster reallocation to improve the UUV's stability, the likelihood of collision decreases to 0.15. We are currently validating the estimated likelihood across large simulation runs.

6.5 Conclusion and Future Work

ReSonAte captures design-time information about system hazard propagation, control strategies, and potential consequences using BTDs, then uses the information contained in these models to calculate risk at run-time. The frequency of threat events and the effectiveness of hazard control strategies influence the estimated risk and may be conditionally dependent on the state of the system and operating environment. A technique for measuring these conditional relationships in simulation using a custom SDL was demonstrated. ReSonAte was then applied to an AV example in the CARLA simulator to dynamically assess the risk of

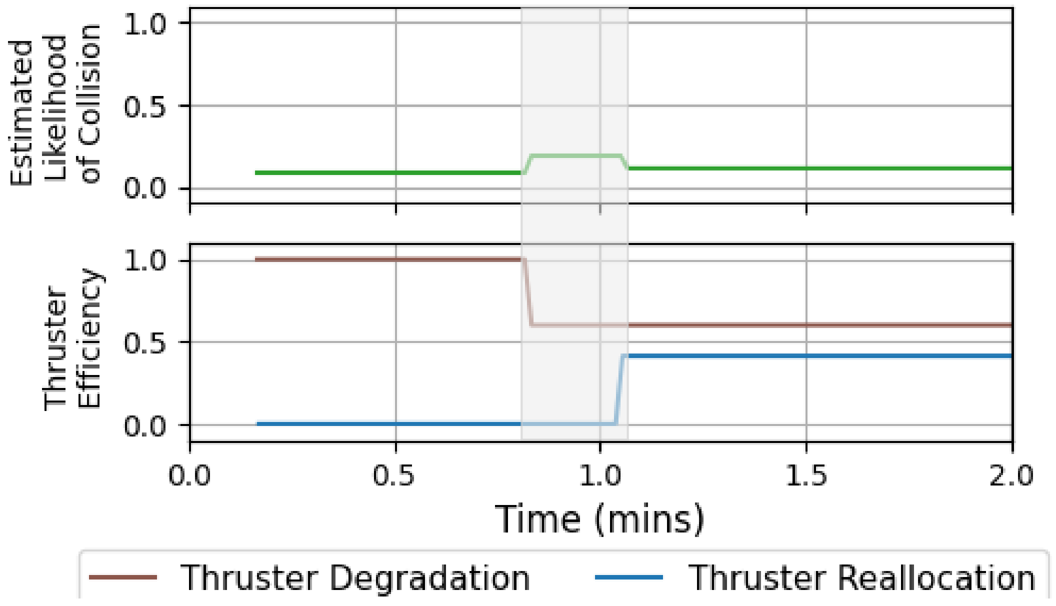


Figure 6.9: ReSonAte estimated likelihood of collision for the BlueROV2 example. As seen in gray shaded region, the likelihood of collision increases when thruster degradation occurs, then reduces after thruster control reallocation.

collision, and a strong correlation was found between the estimated likelihood of a collision and the observed collisions. Additionally, ReSonAte's risk calculations require minimal computational resources making it suitable for resource-constrained and real-time CPSs.

Future extensions and applications for ReSonAte include: (1) dynamic estimation of event severity in addition to event likelihood, (2) inclusion of state uncertainty into risk calculations to produce confidence bounds on risk estimates, (3) forecasting future risk based on expected changes to system or environment, (4) use of estimated risk for higher-level decision making such as controller switching or enactment of contingency plans, and (5) continuous improvement of conditional probability estimates at run-time from operational data.

CHAPTER 7

Conclusion

Autonomous CPSs, and the LECs they commonly depend on, present new challenges for safety assurance. Standards and regulations which have traditionally guided safety assurance processes have struggled to keep pace with rapid technological advancement. Data-driven techniques such as machine learning shift the development focus from analytical design to collection and curation of data. LECs also introduce new failure modes which are often not well handled with existing techniques and require architectural solutions for resiliency. These changes, combined with ever-increasing system complexity, have made analyzing and understanding these systems more challenging than before.

At the same time, the need for strong safety assurance of these systems is greater than ever due to the increasing scale, widespread use, and societal impact of these systems. Assurance cases have supported a shift from highly prescriptive safety processes to more flexible, goal-based processes, but it is critical that safety be treated as a primary development concern which evolves alongside the system design. In turn, safety assurance techniques for these systems must support frequent iteration and help build better understanding of system behavior.

This dissertation presents several research contributions to the safety assurance field related to the areas of formal system analysis, assurance case construction and evaluation, and integration of safety assurance and data-driven techniques into the CPS development cycle. Together, these techniques enable developers to better plan, analyze, and understand how their designs ensure the system will operate safely. The automation provided by these techniques enables safety assurance to be more tightly integrated in the system development process in a cost-effective manner.

7.1 Future Work

While each publication in this dissertation presented possible avenues of future work, we emphasize some of the most prominent directions for each technique as follows:

- The CPN-based timing analysis presented in Chapter 3, along with other similar formal analysis techniques, can offer strong guarantees for safety properties or otherwise provide clear counter-examples, but poor scalability limits the applicability to realistic, full-scale systems operating in highly non-deterministic environments. An algorithm for generating branches of the state space in parallel, combined with modern GPU-acceleration techniques, may offer significant scalability improvements, as would additional techniques for reducing the size of the state space required to verify system properties.

- The ALC Toolchain presented in Chapter 4 allows many heterogeneous DSMLs used for development of autonomous CPSs to be modeled and interconnected in a single integrated environment. These interconnected models provide a rich library of information about a system, and Chapter 5 showed how this information can be leveraged to automatically construct assurance cases. However, many similar opportunities for utilizing the information contained in these models remain unexplored. For example, a detailed operating environment model could be used as a basis for computing coverage metrics such as the sufficiency of an LEC training data set or the completeness of statistical analyses used in the ReSonAte framework.

- Chapter 5 presents a automated method for constructing an assurance case from an existing set of system design models. To facilitate manual review and refinement, the method maintains traceability from objects in the generated assurance case to the corresponding model artifacts as well as explainability of the model relationships which drove the construction process. However, validating that the assurance case is correct and provides sufficient assurance for system safety objectives is left as an open question.

- Finally, Chapter 6 presents a method for dynamically estimating risk posed by various hazard conditions through the ReSonAte framework. These dynamic estimates are used to periodically evaluate risk thresholds in existing assurance arguments. The risk estimates would also be useful for system decision making - i.e., to determine the lowest risk path to an objective or the safest contingency plan in case of a failure - but these possibilities remain unexplored.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Adelard, LLP (2011). Assurance and Safety Case Environment (ASCE). https://www.adelard.com/asce.

Administration, F. A. (2013). Unmanned Aircraft Systems (UAS) Operational Approval. *online: https://www.faa.gov/documentLibrary/media/Notice/N_8900.227.pdf*.

Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., and Karsai, G. (2005). Reusable idioms and patterns in graph transformation languages. *Electronic Notes in Theoretical Computer Science*, 127(1):181–192.

Aiello, M. A., Hocking, A. B., Knight, J., and Rowanhill, J. (2014). Sct: a safety case toolkit. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 216–219. IEEE.

Alur, R., Courcoubetis, C., and Dill, D. (1990). Model-checking for real-time systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

Alves, E. E., Bhatt, D., Hall, B., Driscoll, K., Murugesan, A., and Rushby, J. (2018). Considerations in assuring safety of increasingly autonomous systems.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. (2004). *TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems*, pages 60–72. Springer Berlin Heidelberg, Berlin, Heidelberg.

Amundson, I. and Cofer, D. (2021). Resolute assurance arguments for cyber assured systems engineering. In *Proceedings of the Workshop on Design Automation for CPS and IoT*, pages 7–12.

Austin, R. A., Mahadevan, N., Witulski, A. F., Evans, J., and Witulski, A. F. (2018). Radiation assurance of cubesat payloads using bayesian networks and fault models. In *2018 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–5.

Ayoub, A., Chang, J., Sokolsky, O., and Lee, I. (2013). Assessing the overall sufficiency of safety arguments.

Badreddine, A. and Amor, N. B. (2013). A bayesian approach to construct bow tie diagrams for risk evaluation. *Process Safety and Environmental Protection*, 91(3):159–171.

Badreddine, A. and Ben Amor, N. (2010). A new approach to construct optimal bow tie diagrams for risk analysis. In García-Pedrajas, N., Herrera, F., Fyfe, C., Benítez, J. M., and Ali, M., editors, *Trends in Applied Intelligent Systems*, pages 595–604, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bagheri, H., Kang, E., and Mansoor, N. (2020). Synthesis of assurance cases for software certification. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 61–64.

Baheti, R. and Gill, H. (2011). Cyber-physical systems. *The impact of control technology*, 12(1):161–166.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Banks, A. and Gupta, R. (2014). Mqtt version 3.1. 1. *OASIS standard*, 29:89.

Barroca, L. M. and McDermid, J. A. (1992). Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599.

Barry, M. R. (2011). Certware: A workbench for safety case production and analysis. In *2011 Aerospace conference*, pages 1–10. IEEE.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1995). Uppaal—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1996). *UPPAAL — a tool suite for automatic verification of real-time systems*, pages 232–243. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bishop, P. and Bloomfield, R. (2000). A methodology for safety case development. In *Safety and Reliability*, volume 20, pages 34–42. Taylor & Francis.

Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., et al. (2012). Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 076, pages 173–184. Linköping University Electronic Press.

Blue Robotics (2016). Bluerov2. *Datasheet*.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Broll, B., Maroti, M., Volgyesi, P., and Ledeczi, A. (2018). DeepForge: A Scientific Gateway for Deep Learning. In *Gateways 2018*.

Broy, M., Kirstan, S., Krcmar, H., and Schätz, B. (2012). What is the benefit of a model-based design of embedded software systems in the car industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369. IGI Global.

Byun, T. and Rayadurgam, S. (2020). Manifold for machine learning assurance. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 97–100.

Cai, F. and Koutsoukos, X. (2020). Real-time out-of-distribution detection in learning-enabled cyber-physical systems. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 174–183, Los Alamitos, CA, USA. IEEE Computer Society.

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M. U., Habli, I., and Kelly, T. (2017). Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11):1039–1069.

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M. U., Habli, I., and Kelly, T. (2018). Entrust: engineering trustworthy self-adaptive software with dynamic assurance cases. In *Proceedings of the 40th International Conference on Software Engineering*, pages 495–495.

Chen, D., Zhou, B., Koltun, V., and Krähenbühl, P. (2020). Learning by cheating. In *Conference on Robot Learning*, pages 66–75. PMLR.

Chollet, F. (2015). Keras. https://keras.io/.

Chowdhury, T., Lin, C.-W., Kim, B., Lawford, M., Shiraishi, S., and Wassyng, A. (2017). Principles for systematic development of an assurance case template from iso 26262. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 69–72. IEEE.

Clothier, R. A., Williams, B. P., and Fulton, N. L. (2015). Structuring the safety case for unmanned aircraft system operations in non-segregated airspace. *Safety science*, 79:213–228.

Cyra, L. and Górski, J. (2011). Support for argument structures review and assessment. *Reliability Engineering & System Safety*, 96(1):26–37. Special Issue on Safecomp 2008.

Daskaya, I., Huhn, M., and Milius, S. (2011). Formal safety analysis in industrial practice. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 68–84. Springer.

de Ruijter, A. and Guldenmund, F. (2016). The bowtie method: A review. *Safety Science*, 88:211–218.

Dejanović, I., Vaderna, R., Milosavljević, G., and Vuković, Ž. (2017). Textx: a python tool for domain-specific languages implementation. *Knowledge-Based Systems*, 115:1–4.

Dekker, S., Cilliers, P., and Hofmeyr, J.-H. (2011). The complexity of failure: Implications of complexity theory for safety investigations. *Safety science*, 49(6):939–945.

Delvosalle, C., Fievez, C., Pipart, A., and Debray, B. (2006). Aramis project: A comprehensive methodology for the identification of reference accident scenarios in process industries. *Journal of Hazardous Materials*, 130(3):200–219.

Denney, E. and Pai, G. (2013). A formal basis for safety case patterns. In *International Conference on Computer Safety, Reliability, and Security*, pages 21–32. Springer.

Denney, E. and Pai, G. (2018). Tool support for assurance case development. *Automated Software Engineering*, 25(3):435–499.

Denney, E., Pai, G., and Habli, I. (2011). Towards measurement of confidence in safety cases. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 380–383. IEEE.

Denney, E., Pai, G., and Habli, I. (2015). Dynamic safety cases for through-life safety assurance. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 587–590. IEEE.

Denney, E., Pai, G., and Pohl, J. (2012). Advocate: An assurance case automation toolset. In *International Conference on Computer Safety, Reliability, and Security*, pages 8–21. Springer.

Denney, E., Pai, G., and Whiteside, I. (2013). Hierarchical safety cases. In *NASA Formal Methods Symposium*, pages 478–483. Springer.

Denney, E., Pai, G., and Whiteside, I. (2017). Modeling the safety architecture of uas flight operations. In Tonetta, S., Schoitsch, E., and Bitsch, F., editors, *Computer Safety, Reliability, and Security*, pages 162–178, Cham. Springer, Springer International Publishing.

Denney, E., Pai, G., and Whiteside, I. (2019). The role of safety architectures in aviation safety cases. *Reliability Engineering & System Safety*, 191:106502.

Denney, E. W. and Pai, G. J. (2015). Safety case patterns: theory and applications. *NASA/TM2015218492*.

Dezfuli, H., Benjamin, A., Everett, C., Smith, C., Stamatelatos, M., and Youngblood, R. (2011). Nasa system safety handbook. volume 1; system safety framework and concepts for implementation.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017a). CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017b). Carla: An open urban driving simulator. *arXiv:1711.03938*.

Dowek, G., Muñoz, C., and Păsăreanu, C. (2007). A formal analysis framework for PLEXIL. In *Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems*.

Dowek, G., Muñoz, C., and Păsăreanu, C. (2008). A small-step semantics of PLEXIL. Technical Report 2008-11, National Institute of Aerospace, Hampton, VA.

Duan, L., Rayadurgam, S., Heimdahl, M. P., Sokolsky, O., and Lee, I. (2014). Representing confidence in assurance case evidence. In *International Conference on Computer Safety, Reliability, and Security*, pages 15–26. Springer.

EASA (2020). Concepts of Design Assurance for Neural Networks (CoDANN). Publication, European Union Aviation Safety Agency.

Easwaran, A., Shin, I., Sokolsky, O., and Lee, I. (2006). Incremental schedulability analysis of hierarchical real-time components. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 272–281, New York, NY, USA. ACM.

Eisele, S., Mardari, I., Dubey, A., and Karsai, G. (2017). Riaps: Resilient information architecture platform for decentralized smart systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 125–132. IEEE.

FAA (2000). FAA System Safety Handbook.

Federal Aviation Administration, O. (2019). Risk management handbook (faa-h-8083-2).

Fedorova, V., Gammerman, A., Nouretdinov, I., and Vovk, V. (2012). Plug-in martingales for testing exchangeability on-line. *arXiv preprint arXiv:1204.3251*.

Feiler, P. (2019). The open source aadl tool environment (osate). Technical report, Carnegie Mellon University Software Engineering Institute Pittsburgh United ....

Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Feiler, P. H., Lewis, B., Vestal, S., and Colbert, E. (2005). *An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering*, pages 3–15. Springer US, Boston, MA.

Ferdous, R., Khan, F., Sadiq, R., Amyotte, P., and Veitch, B. (2009). Handling data uncertainties in event tree analysis. *Process safety and environmental protection*, 87(5):283–292.

Fitzgerald, J., Larsen, P. G., and Verhoef, M. (2014). From embedded to cyber-physical systems: Challenges and future directions. In *Collaborative design for embedded systems*, pages 293–303. Springer.

Foretellix, O. (2021). Open m-sdl.

Fremont, D. J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. (2019). Scenic: A language for scenario specification and scene generation. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*.

Friedenthal, S., Moore, A., and Steiner, R. (2014). *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

Gacek, A., Backes, J., Cofer, D., Slind, K., and Whalen, M. (2014). Resolute: an assurance case language for architecture models. *ACM SIGAda Ada Letters*, 34(3):19–28.

Galbraith, J. and Saarenmaa, O. (2006). SSH file transfer protocol.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer.

Gehman, H. W. (2003). *Columbia accident investigation board report*. Columbia Accident Investigation Board.

Goodenough, J. B., Weinstock, C. B., et al. (2012). Toward a theory of assurance case confidence. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples. *arXiv e-prints*, page arXiv:1412.6572.

Graydon, P., Habli, I., Hawkins, R., Kelly, T., and Knight, J. (2012). Arguing conformance. *IEEE software*, 29(3):50–57.

Graydon, P. J. (2015). Formal assurance arguments: A solution in search of a problem? In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 517–528. IEEE.

Graydon, P. J. and Holloway, C. M. (2017). An investigation of proposed techniques for quantifying confidence in assurance arguments. *Safety science*, 92:53–65.

Graydon, P. J., Knight, J. C., and Strunk, E. A. (2007). Assurance based development of critical systems. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 347–357. IEEE.

Groen, F. J., Evans, J. W., and Hall, A. J. (2015). A vision for spaceflight reliability: Nasa's objectives based strategy. In *Reliability and Maintainability Symposium (RAMS), 2015 Annual*, pages 1,6. IEEE.

Group, G. W. (2011). Gsn community standard version 1. *Origin Consulting (York) Limited*.

Group, O. M. et al. (2010). Uml profile for marte: Modeling and analysis of real-time embedded systems, version 1.0. *On line: http://www.omg.org/spec/MARTE*.

Haddon-Cave, C. (2009). *The Nimrod Review: an independent review into the broader issues surrounding the loss of the RAF Nimrod MR2 aircraft XV230 in Afghanistan in 2006, report*, volume 1025. DERECHO INTERNACIONAL.

Haider, A. A. and Nadeem, A. (2013). A survey of safety analysis techniques for safety critical systems. *International Journal of Future Computer and Communication*, 2(2):134.

Han, T. A., Kencana Ramli, C., and Damásio, C. (2008). An implementation of extended p-log using xasp. volume 5366, pages 739–743.

Harbour, M. G., García, J. J. G., Gutiérrez, J. C. P., and Moyano, J. M. D. (2001). MAST: Modeling and analysis suite for real time applications. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 125–134.

Hartmann, T., Moawad, A., Fouquet, F., and Le Traon, Y. (2017). The next evolution of mde: a seamless integration of machine learning into domain modeling. *Software & Systems Modeling*, pages 1–20.

Hartsell, C., Ramakrishna, S., Dubey, A., Stojcsics, D., Mahadevan, N., and Karsai, G. (2021). Resonate: A runtime risk assessment framework for autonomous systems. *arXiv preprint arXiv:2102.09419*.

Hawkins, R., Habli, I., Kolovos, D., Paige, R., and Kelly, T. (2015). Weaving an assurance case from design: a model-based approach. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 110–117. IEEE.

Held, D., Thrun, S., and Savarese, S. (2016). Learning to track at 100 fps with deep regression networks. In *European Conference on Computer Vision*, pages 749–765. Springer.

Hintjens, P. (2013). *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

Hobbs, C. and Lloyd, M. (2012). The application of bayesian belief networks to assurance case preparation. In *Achieving Systems Safety*, pages 159–176. Springer.

Holloway, C. M. (2013). Making the implicit explicit: Towards an assurance case for do-178c.

IAEA (2019). *IAEA Safety Glossary: 2018 Edition*. Non-serial Publications. INTERNATIONAL ATOMIC ENERGY AGENCY, Vienna.

IBM (2021). Rational rhapsody. http://www.ibm.com/software/awdtools/rhap sody/.

IEC (2016). Reliability Block Diagrams. Standard, International Electrotechnical Commission.

ISO (2018). ISO 26262:2018 Road vehicles - Functional safety.

Jensen, F. V. et al. (1996). *An introduction to Bayesian networks*, volume 210. UCL press London.

Jensen, K. and Kristensen, L. M. (2009a). *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer.

Jensen, K. and Kristensen, L. M. (2009b). *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media.

Jøsang, A. (2001). A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(03):279–311.

Karunakaran, D., Worrall, S., and Nebot, E. (2020). Efficient statistical validation with edge cases to evaluate highly automated vehicles. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE.

Katrakazas, C., Quddus, M., and Chen, W.-H. (2019). A new integrated collision risk assessment methodology for autonomous vehicles. *Accident Analysis & Prevention*, 127:61–79.

Kelly, T. and McDermid, J. (1998). Safety case patterns-reusing successful arguments. In *IEEE Colloquium on Understanding Patterns and Their Application to Systems Engineering (Digest No. 1998/308)*, pages 31–39.

Kelly, T. and Weaver, R. (2004). The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, page 6. Citeseer.

Kelly, T. P. (1999). *Arguing safety: a systematic approach to managing safety cases*. PhD thesis, University of York York, UK.

Kelly, T. P. and McDermid, J. A. (1997). Safety case construction and reuse using patterns. In *Safe Comp 97*, pages 55–69. Springer.

Khakzad, N., Khan, F., and Amyotte, P. (2012). Dynamic risk analysis using bow-tie approach. *Reliability Engineering & System Safety*, 104:36–44.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., et al. (2016). Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90.

Koenig, N. P. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Citeseer.

Koopman, P., Kane, A., and Black, J. (2019). Credible autonomy safety argumentation.

Koopman, P. and Wagner, M. (2016). Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24.

Kristensen, L. M., Billington, J., Petrucci, L., Qureshi, Z. H., and Kiefer, R. (2002). Formal specification and analysis of airborne mission systems. In *Proceedings. The 21st Digital Avionics Systems Conference*, volume 1, pages 4D4–1–4D4–13 vol.1.

Kumar, P. S., Dubey, A., and Karsai, G. (2014). Colored petri net-based modeling and formal analysis of component-based applications. page 79–88.

Kumar, P. S. and Karsai, G. (2015). Integrated analysis of temporal behavior of component-based distributed real-time embedded systems. In *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 50–57. IEEE.

Kurd, Z., Kelly, T., McDermid, J., Calinescu, R., and Kwiatkowska, M. (2009). Establishing a framework for dynamic risk management in 'intelligent'aero-engine control. In *International Conference on Computer Safety, Reliability, and Security*, pages 326–341. Springer.

Larsen, P. G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., et al. (2016). Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6. IEEE.

Le Sergent, T., Dormoy, F.-X., and Le Guennec, A. (2016). Benefits of model based system engineering for avionics systems.

Lee, E. A. (2008). Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE.

Lee, I., Bremond-Gregoire, P., and Gerber, R. (1994). A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, 82(1):158–171.

Lee, P. M. (1989). *Bayesian statistics*. Oxford University Press London:.

Lee, W.-S., Grosh, D. L., Tillman, F. A., and Lie, C. H. (1985). Fault tree analysis, methods, and applications: A review. *IEEE transactions on reliability*, 34(3):194–203.

Leite, F. L., Schneider, D., and Adler, R. (2018). Dynamic risk management for cooperative autonomous medical cyber-physical systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 126–138. Springer.

Leveson, N. (2011a). *Engineering a safer world: Systems thinking applied to safety*.

Leveson, N. G. (1995). *Safeware: system safety and computers*. ACM.

Leveson, N. G. (2011b). The use of safety cases in certification and regulation.

Lewis, E. (1995). *Introduction to Reliability Engineering, 2nd Edition*. Wiley.

Lopez, P., Medina, J. L., and Drake, J. M. (2006). Real-time modelling of distributed component-based applications. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 92–99.

Mahadevan, N., Dubey, A., and Karsai, G. (2011). Application of software health management techniques. In *Proceedings of the 6th international symposium on software engineering for adaptive and self-managing systems*, pages 1–10.

Mahadevan, N., Dubey, A., and Karsai, G. (2012). Architecting health management into software component assemblies: Lessons learned from the arinc-653 component mode. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 79–86. IEEE.

Maksimov, M., Fung, N. L. S., Kokaly, S., and Chechik, M. (2018). Two decades of assurance case tools: A survey. In Gallina, B., Skavhaug, A., Schoitsch, E., and Bitsch, F., editors, *International Conference on Computer Safety, Reliability, and Security*, pages 49–59, Cham. Springer, Springer International Publishing.

Manhães, M. M. M., Scherer, S. A., Voss, M., Douat, L. R., and Rauschenbach, T. (2016). UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey*. IEEE.

Marcos, D. D. F., Jean, B., Frédéric, J., Erwan, B., and Guillaume, G. (2005). Amw: A generic model weaver. *Proc. of the 1eres Journées sur l'Ingénierie Dirigée par les Modeles*, 200.

Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., and Lédeczi, Á. (2014). Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60.

Matsuno, Y. (2011). D-case editor: A typed assurance case editor. *University of Tokyo*.

Matsuno, Y. and Yamamoto, S. (2012). Toward dynamic assurance cases. In *JCKBSE*, pages 154–160.

McComas, D. (2012). NASA/GSFC's Flight Software Core Flight System. Flight Software Workshop.

McFarlane, N. (2007). Generic safety assessment for atc surveillance using wide area multilateration, helios technology. *WAM Safety Study & Surveillance Generic Safety. Eurocontrol. Bob Darby*.

Medina, J. L. and Cuesta, A. G. (2011). From composable design models to schedulability analysis with uml and the uml profile for marte. *SIGBED Rev.*, 8(1):64–68.

Misra, A. (1994). *Senor-based diagnosis of dynamical systems*. PhD thesis, Vanderbilt University Ph. D. dissertation.

Molnár, V., Graics, B., Vörös, A., Majzik, I., and Varró, D. (2018). The gamma statechart composition framework. In *Internation Conference on Software Engineering*. ICSE.

Mosterman, P. J. and Zander, J. (2016). Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems. *Software & Systems Modeling*, 15(1):5–16.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Object Management Group (2009). *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*.

OMG (2017). OMG Systems Modeling Language (OMG SysML), Version 1.5.

Ortmeier, F. and Reif, W. (2006). Failure-sensitive specification: A formal method for finding failure modes.

Ortmeier, F., Reif, W., and Schellhorn, G. (2005). Deductive cause-consequence analysis (dcca). *IFAC Proceedings Volumes*, 38(1):62–67.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag.

Ozay, M., Esnaola, I., Vural, F. T. Y., Kulkarni, S. R., and Poor, H. V. (2016). Machine learning methods for attack detection in the smart grid. *IEEE transactions on neural networks and learning systems*, 27(8):1773–1786.

Papadopoulos, H. (2008). Inductive conformal prediction: Theory and application to neural networks. In *Tools in artificial intelligence*. InTech.

Papadopoulos, H., Vovk, V., and Gammerman, A. (2011). Regression conformal prediction with nearest neighbours. *Journal of Artificial Intelligence Research*, 40:815–840.

Pardo-Castellote, G. (2003). Omg data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206. IEEE.

Pasaje, J. L. M., Harbour, M. G., and Drake, J. M. (2001). MAST Real-Time View: a graphic UML tool for modeling object-oriented real-time systems. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pages 245–256.

Pech-Pacheco, J. L., Cristóbal, G., Chamorro-Martinez, J., and Fernández-Valdivia, J. (2000). Diatom autofocusing in brightfield microscopy: a comparative study. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 3, pages 314–317. IEEE.

Pei, K., Cao, Y., Yang, J., and Jana, S. (2017). DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Rae, A. and Alexander, R. (2017). Forecasts or fortune-telling: When are expert judgements of safety risk valid? *Safety Science*, 99:156–165. Risk Analysis Validation and Trust in Risk management.

Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Design Automation Conference*, pages 731–736. IEEE.

Ratzer, A. V., Wells, L., Lassen, H. M., Laursen, M., Qvortrup, J. F., Stissing, M. S., Westergaard, M., Christensen, S., and Jensen, K. (2003). *CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets*, pages 450–462. Springer Berlin Heidelberg, Berlin, Heidelberg.

Renault, X., Kordon, F., and Hugues, J. (2009a). Adapting models to model checkers, a case study : Analysing aadl using time or colored petri nets. In *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 26–33.

Renault, X., Kordon, F., and Hugues, J. (2009b). From aadl architectural models to petri nets: Checking model viability. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 313–320.

Rhodes, T., Boland, F., Fong, E., and Kass, M. (2010). Software assurance using structured assurance case models. *Journal of research of the National Institute of Standards and Technology*, 115(3):209.

Rinehart, D. J., Knight, J. C., and Rowanhill, J. (2015). Current practices in constructing and evaluating assurance cases with applications to aviation.

Rinehart, D. J., Knight, J. C., and Rowanhill, J. (2017). Understanding What It Means for Assurance Cases to 'Work'.

Rosques, A. (2017). *PlantUML.*

Rouvroye, J. L. (2001). *Enhanced Markov Analysis as a method to assess safety in the process industry.*

Rouvroye, J. L. and van den Bliek, E. G. (2002). Comparing safety analysis techniques. *Reliability Engineering & System Safety*, 75(3):289–294.

RTCA (2011). DO-178C: Software Considerations in Airborne Systems and Equipment Certification.

Rushby, J. (2008). Runtime certification. In *International Workshop on Runtime Verification*, pages 21–35. Springer.

Rushby, J. (2015). The interpretation and evaluation of assurance cases. *Comp. Science Laboratory, SRI International, Tech. Rep. SRI-CSL-15-01.*

Saint-Andre, P. et al. (2004). Extensible messaging and presence protocol (xmpp): Core.

Schneider, D. and Trapp, M. (2013). Conditional safety certification of open adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(2):1–20.

Schwalbe, G. and Schels, M. (2020). A survey on methods for the safety assurance of machine learning based systems. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020).*

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In *NIPS*.

Sentz, K., Ferson, S., et al. (2002). *Combination of evidence in Dempster-Shafer theory*, volume 4015. Citeseer.

Seshia, S. A. and Sadigh, D. (2016). Towards verified artificial intelligence. *CoRR*, abs/1606.08514.

Sha, L. (2001). Using simplicity to control complexity. *IEEE Software*, 4:20–28.

Shafer, G. and Vovk, V. (2008). A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9(Mar):371–421.

Sokolsky, O., Lee, I., and Clarke, D. (2006). Schedulability analysis of aadl models. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–.

Sombolestan, S. M., Rasooli, A., and Khodaygan, S. (2018). Optimal path-planning for mobile robots to find a hidden target in an unknown environment based on machine learning. *Journal of Ambient Intelligence and Humanized Computing*.

Spriggs, J. (2012). *GSN-the goal structuring notation: A structured approach to presenting arguments*. Springer Science & Business Media.

Srivastava, A. N. and Schumann, J. (2011). The case for software health management. In *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, pages 3–9. IEEE.

Stamatis, D. H. (2003). *Failure mode and effect analysis: FMEA from theory to execution*. Quality Press.

Steele, P., Collins, K., and Knight, J. (2011). Access: A toolset for safety case creation and management. In *Proc. 29th Intl. Systems Safety Conf.(August 2011)*.

Steinbauer, G. and Wotawa, F. (2013). Model-based reasoning for self-adaptive systems–theory and practice. In *Assurances for Self-Adaptive Systems*, pages 187–213. Springer.

Sundar, V. K., Ramakrishna, S., Rahiminasab, Z., Easwaran, A., and Dubey, A. (2020). Out-of-distribution detection in multi-label datasets using latent space of β-vae. *arXiv preprint arXiv:2003.08740*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1st edition.

Szczygielska, M. and Jarzębowicz, A. (2017). Assurance case patterns on-line catalogue. In *Advances in Dependability Engineering of Complex Systems*, pages 407–417. Springer.

Sztipanovits, J., Bapty, T., Koutsoukos, X., Lattmann, Z., Neema, S., and Jackson, E. (2018). Model and tool integration platforms for cyber–physical system design. *Proceedings of the IEEE*, 106(9):1501–1526.

Sztipanovits, J., Bapty, T., Neema, S., Howard, L., and Jackson, E. (2014). Openmeta: A model-and component-based design tool chain for cyber-physical systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 235–248. Springer.

Sztipanovits, J., Bapty, T., Neema, S., Koutsoukos, X., and Jackson, E. (2015). Design tool chain for cyber-physical systems: Lessons learned. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Teimourikia, M., Fugini, M., and Raibulet, C. (2017). Run-time security and safety management in adaptive smart work environments. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 256–261. IEEE.

The Assurance Case Working Group (2018). Goal Structuring Notation Community Standard (Version 2). https://scsc.uk/r141B:1.

UK Ministry of Defense (2007). Safety management requirements for defence systems.

U.S. Department of Defense (1991). Military Handbook: Reliability Prediction of Electronic Equipment. Standard, US Department of Defense.

U.S. Department of Transportation (2018). Preparing for the future of transportation: Automated vehicles 3.0.

US Food and Drug Administration and others (2010). Guidance for industry and fda staff-total product life cycle: Infusion pump- premarket notification[510 (k)] submissions. *Issued April*, 23.

USRA (2014). *PLEXIL*.

Verma, V., Jónsson, A., Pasareanu, C., and Iatauro, M. (2006). Universal executive and plexil: Engine and language for robust spacecraft control and operations. In *Proceedings of AIAA Space 2006*.

Vileiniskis, M. and Remenyte-Prescott, R. (2017). Quantitative risk prognostics framework based on petri net and bow-tie models. *Reliability Engineering & System Safety*, 165:62–73.

Vörös, A., Búr, M., Ráth, I., Horváth, Á., Micskei, Z., Balogh, L., Hegyi, B., Horváth, B., Mázló, Z., and Varró, D. (2018). Modes3: model-based demonstrator for smart and safe cyber-physical systems. In *NASA Formal Methods Symposium*, pages 460–467. Springer.

Wardziński, A. (2008). Safety assurance strategies for autonomous vehicles. In *International Conference on Computer Safety, Reliability, and Security*, pages 277–290. Springer.

Westergaard, M., Evangelista, S., and Kristensen, L. M. (2009). *ASAP: An Extensible Platform for State Space Analysis*, pages 303–312. Springer Berlin Heidelberg, Berlin, Heidelberg.

White, J., Schmidt, D. C., and Mulligan, S. (2007). The generic eclipse modeling system. In *Model-Driven Development Tool Implementer's Forum, TOOLS*, volume 7, page 82. Citeseer.

Williams, B. P., Clothier, R., Fulton, N., Johnson, S., Lin, X., and Cox, K. (2014). Building the safety case for uas operations in support of natural disaster response. In *14th AIAA Aviation Technology, Integration, and Operations Conference*, page 2286.

Wilmot, J. (2005). A core flight software system. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 13–14.

Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., and Sumner, A. (2000). Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4:6.

Xiang, W., Musau, P., Wild, A. A., Lopez, D. M., Hamilton, N., Yang, X., Rosenfeld, J. A., and Johnson, T. T. (2018a). Verification for machine learning, autonomy, and neural networks survey. *CoRR*, abs/1810.01989.

Xiang, W., Tran, H.-D., and Johnson, T. T. (2018b). Output reachable set estimation and verification for multi-layer neural networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*.

Yang, Q., Tian, J., and Zhao, T. (2017). Safety is an emergent property: Illustrating functional resonance in air traffic management with formal verification. *Safety science*, 93:162–177.

Zabell, S. L. (1989). The rule of succession. *Erkenntnis*, 31(2):283–321.

Zhang, Y. and Guan, X. (2018). Selecting project risk preventive and protective strategies based on bow-tie analysis. *Journal of Management in Engineering*, 34(3):04018009.