Project Number 101017258

D4.1 Safety Analysis Concept and Methodology for EDDI development
(Initial Version)

Version 1.0
23 December 2021
Final

Public Distribution

University of Hull and Fraunhofer IESE

Project Partners: Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York

PROJECT PARTNER CONTACT INFORMATION

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial outline	21 October 2021
0.2	Second version; sections 1 & 2	3 December 2021
0.3	Input from IESE; sections 3 & 4	16 December 2021
0.4	Further IESE input; sections 5 & 6; ready for internal review	16 December 2021
0.8	Reviewed version	20 December 2021
0.9	Final version for QA	22 December 2021
1.0	Final version	23 December 2021

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

This deliverable describes the proposed safety analysis concept and accompanying methodology to be defined in the SESAME project. Three overarching challenges to the development of safe and secure multi-robot systems are identified — complexity, intelligence, and autonomy — and in each case, we review state-of-the-art techniques that can be used to address them and explain how we intend to integrate them as part of the key SESAME safety and security concept, the EDDI.

The challenge of complexity is largely addressed by means of compositional model-based safety analysis techniques that can break down the complexity into more manageable parts. This applies both to scale — modelling systems hierarchically and embedding local failure logic at the component-level — and to tasks, where different safety-related tasks (including not just analysis but also requirements allocation and assurance case generation) can be handled by the same set of models. All of this can be combined with the existing DDI concept to create models — EDDIs — that store all of the necessary information to support a gamut of design-time safety processes.

Against the challenge of intelligence, we propose a trio of techniques: SafeML and Uncertainty Wrappers for estimating the confidence of a given classification, which can be used as a form of reliability measure, and SMILE for explainability purposes. By enabling us to measure and explain the reliability of ML decision making, we can integrate ML behaviour as part of a wider system safety model, e.g. as one input into a fault tree or Bayesian network. In addition to providing valuable feedback during training, testing, and verification, this allows the EDDI to perform runtime safety monitoring of ML components.

The EDDI itself is therefore our primary solution to the twin challenges of autonomy and openness. Using the ConSert approach as a foundation, EDDIs can be made to operate cooperatively as part of a distributed system, issuing and receiving guarantees on the basis of their internal executable safety models to collectively achieve tasks in a safe and secure manner.

Finally, a simple methodology is defined to show how the relevant techniques can be applied as part of the EDDI concept throughout the safety development lifecycle.

LIST OF ABBREVIATIONS

AADL	Architecture Analysis & Description Language	HAZOP	Hazard & Operability Study
ADL	Architecture Description Language	HiP-HOPS	Hierarchically performed Hazard Origin & Propagation Studies
AI	Artificial Intelligence	LIME	Local Interpretable Model-agnostic Explanations
ALARP	As Low As Reasonably Possible	MAS	Multi-Agent System
ASIL	Automotive Safety Integrity Level	MBSA	Model-based Safety Analysis
BE	Basic Event (i.e., root cause of a fault tree)	ML	Machine Learning
CAE	Claims, Arguments, Evidence framework	MRS	Multi-Robot System
CCA	Common Cause Analysis	MTTF	Mean Time To Failure (also, MTBF: Mean Time Before Failure)
CCF	Common Cause Failure	MTTR	Mean time to repair
CFT	Component Fault Tree	NN	Neural Network
CMC	Component Markov Chain	OAS	Open Adaptive System
CNN	Convolutional Neural Network	ODE	Open Dependability Exchange
DAL	Design Assurance Level	OOD	Out of Distribution
DDI	Digital Dependability Identity	PRA	Probabilistic Risk Assessment (or Analysis)
DFT	Dynamic Fault Tree	QM	Quality Management only (cf. ASIL)
DL	Deep Learning	RTN	Real-Time Network
DNN	Deep Neural Network	SACM	Structured Assurance Case Metamodel
DRA	Dynamic Risk Assessment	SAML	Safety Analysis Modelling Language
DSPN	Deterministic Stochastic Petri Net	SDM	Safety Domain Model
ECDF	Empirical Cumulative Distribution Function	SEFT	State/Event Fault Tree
EDDI	Executable Digital Dependability Identity	SIL	Safety Integrity Level
FMEA	Failure Modes & Effects Analysis	SINADRA	Situation-Aware Dynamic Risk Assessment
FPTA	Failure Propagation & Transformation Analysis	SMILE	Statistical Model-agnostic Interpretability with Local Explanations
FPTC	Failure Propagation & Transformation Calculus	TARA	Threat Analysis & Risk Assessment
FSAP/ NuSMV	Formal Safety Analysis Platform for NuSMV	TFPG	Timed Failure Propagation Graph
FTA	Fault Tree Analysis	UAS	Unmanned Aircraft/Airborne System
GHCFT	Generalised Hybrid Component Fault Tree	UAV	Unmanned Aerial Vehicle
GSN	Goal Structuring Notation	UW	Uncertainty Wrapper
HARA	Hazard Analysis & Risk Assessment (or Hazard And Risk Analysis)	XAI	Explainable AI

1. INTRODUCTION

1.1 OVERVIEW

Invisible to the majority of people, a crucial yet never-ending race is constantly being run. With every year that passes, society benefits from new devices, new technologies, and improved systems. At the same time, their expectations of those systems become increasingly stringent even as they become more and more advanced.

One of the most vital expectations is that of safety. Society as a whole is generally more safety-aware and risk-averse than ever before, both in terms of risk to human life and risk to the wider environment. And yet to fulfil that expectation and ensure modern systems remain safe means it is impossible for the field of safety engineering to stand still: it is forever racing to keep up with new technologies and ever-increasing complexity. Electronics and computer-based systems, in particular, pose difficulties for traditional safety analysis approaches.

To compound the issue further, safety-critical systems — those whose failure may cause harm to people, property, or the environment — are increasingly pervasive as we incorporate more and more technology into our daily lives. With additional exposure comes additional risk, and high-profile disasters like the Fukushima Daiichi nuclear accident means that safety is in the public consciousness, making it a forefront design consideration and not an afterthought.

To this end, various new safety standards have been introduced in recent years that impose stricter requirements and stringent methodologies. Standards like ISO 26262[1] (for automotive safety), ARP4754-A[2] (for the aerospace sector), and IEC 61508[3] (for general safety-related systems) define processes in which dependability — which encompasses not just safety, but related characteristics such as reliability, maintainability, and security — is a key design objective to be considered right through the design lifecycle, from the initial concept to implementation to deployment and even during operation. Similar methodological standards also exist in other fields, e.g. the rail and process industries, and designers of safety-critical systems must also take into account other related standards, like ISO 15858[4] (regulating exposure of UV-C light). Nor is regulation static; new standards are regularly introduced to deal with new technologies and new concerns, such as ISO 21448[5] (an extension of ISO 26262 with a specific focus on functional insufficiencies and unsafe nominal behaviour) or the increasing focus on regulating unmanned aircraft systems (UAS), as evidenced by recent EU regulations[6].

However, creating a standard is one thing but knowing how to implement it is quite another. The introduction of ISO 26262, for example, led to something of a scramble as automotive companies were forced to find or create new models, tools, and techniques to meet the standard's requirements. Traditional methods of safety analysis are

---

[1] https://www.iso.org/standard/68383.html
[2] https://www.sae.org/standards/content/arp4754a/
[3] https://webstore.iec.ch/publication/5515
[4] https://www.iso.org/standard/55553.html
[5] https://www.iso.org/standard/70939.html
[6] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32019R0945 ; https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32019R0947

intensive, largely manual processes and as such are insufficient to handle the demands of assessing modern safety-critical systems, which may be orders of magnitude more complex than those they were intended to deal with. The Space Shuttle's flight computer software, for example, consisted of around 400,000 lines of code[7]; by contrast, the Boeing 787 avionics consist of 6.5 million lines of code and even a modern passenger car may have upwards of 20 million in its navigation system alone[8]. Techniques that were applicable to the design of something even as complex as the Space Shuttle in the 1970-80s are simply not equipped to deal with much more widespread technologies of the 2020s.

To address this, efforts in the field of safety engineering have focused on introducing new computer-aided tools and new, more holistic approaches. In particular, **model-based safety analysis** or MBSA [1] is an attempt to address the problem of keeping dependability data in sync with an evolving design concept by sharing a common set of models. System models are extended with safety-related information such as failure data, and as the design model gets updated, so does the safety model. These models can then be subjected to common analysis techniques semi-automatically, making it possible to repeat the analysis as many times as necessary rather than relying on a single, intensive — and often largely manual — analysis at the end, when it is too late to make any significant changes to the design in response to any problems identified. In this way, MBSA allows dependability to be considered as part of an iterative process throughout the design lifecycle.

But even MBSA is not immune to the march of time and technology. Modern systems are increasingly dynamic, complex, and distributed. Artificial intelligence (AI) — particularly machine learning (ML) — plays an ever-more important role. Entirely new fields, like UAS or self-driving vehicles, present both practical, regulatory, and even ethical questions.

And safety engineering approaches like MBSA must continue to evolve to keep pace.

1.2 SESAME CONTEXT & KEY CHALLENGES

The context introduced by the SESAME project — with its emphasis on multi-robot systems (MRS) — highlights several of the major challenges of modern safety engineering. While robots are not a new invention in themselves, the way they operate in increasingly interconnected, adaptive, and AI-driven ways poses distinct difficulties when carrying out a comprehensive safety analysis. Conversely, UAS-like multirotor drones *are* a relatively new invention, and sufficiently different from manned aerial systems to demand a novel approach — especially when multiple units are operating together as part of a cohesive whole.

One of the most perplexing difficulties facing safety engineers is the fact that only a portion of such systems is known at design time. When a system can learn and adapt during its operational lifetime, or when a system adapts its behaviour dynamically (and autonomously) at runtime in response to its environment or to cooperate with other systems, a simple, static, design-time analysis is no longer sufficient by itself. In addition, such systems need to understand the environmental context in a more

---

[7] https://www.nasa.gov/mission_pages/shuttle/flyout/flyfeature_shuttlecomputers.html
[8] https://spectrum.ieee.org/this-car-runs-on-code

Confidentiality: Public Distribution

sophisticated way to come to safe and efficient adaptation decisions. That extends the required scope of the safety analysis beyond simple internal failure propagation to include interaction between the system and its environment, particularly in the case of safety issues that originate from the dynamic environment rather than the system itself.

These challenges can be broadly separated into three main categories:

- **Complexity** of dynamic systems in a dynamic environment

- **Intelligence** of systems driven by AI & machine learning

- **Autonomy** and unpredictability of open & distributed systems

One of the goals of the SESAME project is to develop a concept, methodology, and supporting tools to address these challenges. Central to this aim is the idea of an Executable Digital Dependability Identity (EDDI), which is synthesised as part of a design-time analysis but which operates alongside the host system at runtime to monitor and respond to potential failures in real time.

This deliverable will describe each challenge, the state of the art in each case, and how the proposed approach in SESAME aims to address them in Sections 2–4. The various concepts and techniques are brought together to form an overarching methodology in Section 5 and in Section 6 we present our concluding remarks and discuss the work ahead.

2. THE CHALLENGE OF COMPLEXITY

2.1 DEFINING THE PROBLEM

While few would argue with the claim that modern safety-critical systems are more complex than ever, complexity comes in many forms. A system may be complex due to sheer scale — either physically, or when it includes a large amount of software (or both). Alternatively, a system may be complex due to its dynamic behaviour: a system with different behaviour in many different states can be challenging even if its scale is relatively modest. And as will be discussed later, a system may also be complex due to AI-driven operation, a distributed nature, or adaptation at runtime.

This section will address the first two specifically: complexity of scale and complexity of state.

2.1.1 Definitions and general safety engineering approaches

Before delving deeper into the discussion, it is useful to define some common terms:

- **Safety** is the property of a system to avoid causing harm to people, property, or the environment. A **safety-critical system** is therefore one that has potential to cause harm if it fails. Note that safety is seldom absolute; it is unlikely for a system to ever be 100% safe.

- Distinct from safety, **reliability** is the property of a system to continue functioning as intended under stated conditions over a given period. An unreliable system is not necessarily unsafe as long as its failure poses no danger to people, property, or the environment.

- **Dependability** is an umbrella term encompassing safety, reliability, and related characteristics, including availability, maintainability, and security. In this report it primarily implies safety and reliability together.

- **Risk** describes the possibility of an adverse event, including one that has consequences for safety. In safety engineering, risk is typically defined as a combination of the **severity** of the consequences and the **likelihood** (i.e., probability) of the event occurring. Again, zero risk is typically impractical; instead the target is often to achieve a risk that is "as low as reasonably practicable"[2] (or ALARP, though other comparable acronyms also exist).

- A **hazard** is a condition, situation, or possible event that leads to an undesirable outcome. In a safety-critical context, a hazard is typically a potential source of harm caused by a failure or similar occurrence.

- **Faults**, **failures**, and **errors** are all related terms treated slightly differently by different sources. In general, a **fault** is an abnormal condition that can lead a system or component to **fail**, i.e., to stop functioning correctly. A **failure mode** is one possible manifestation of the way a system element can fail (and for the purposes of this report is synonymous with a **failure**). An **error** is typically a deviation between an intended value or condition and the actual value or condition. In summary, a fault causes a failure which leads to an error.

- A **safety goal** is a high-level requirement intended to address the risk of one or more hazards (whether by avoiding it, reducing the likelihood, or mitigating the effects). They may be decomposed into lower-level **safety requirements** across system elements, which collectively must be satisfied to achieve the safety goal.

- A **safety integrity level** or SIL is essentially a ranking of the stringency of a safety requirement. Different standards define different forms of SIL: in ISO 26262, they are known as ASILs and are ranked from A (least strict) to D (most strict); in ARP 4754, they are known as DALs and range from A (most strict) to E (least). IEC 61508 and CENELEC 50126 define SILs from 1 (least strict) to 4 (most strict). A SIL may be derived largely from probability (as in IEC 61508) or may involve other attributes, like severity and controllability (e.g. ASILs).

These terms all occur as part of the various safety engineering methodologies defined in various standards. While different standards can vary in the details, the overall process of modern safety engineering is typically as follows:

1. An initial **hazard and risk assessment (HARA)** is carried out on the system design. This is meant to identify the various potential hazards of the system and to assess the risk each poses.

2. On the basis of the risk analysis, high-level safety requirements are defined. Often these are linked directly to the hazards, as with safety goals, such that all hazards are addressed in some fashion.

3. As the design evolves into greater levels of detail, these high-level safety requirements are decomposed in parallel in a top-down, hierarchical manner. In this way, subcomponents of the system responsible for meeting overall safety goals are identified and the traceability of safety requirements is maintained across all levels of the system.

4. To identify design flaws and verify whether the safety requirements are being met, dependability analyses are performed. These analyses can be inductive — starting from individual component failures and determining the consequences — or deductive, starting with high-level hazards and working backwards to identify possible causes. Typically, these analyses also include a probabilistic component, but early on in the design, purely qualitative analyses may be carried out.

5. As part of implementation, a variety of tests are performed, e.g. integration tests or others depending on the stage. Again, this helps verify that requirements have been met and forms part of overall system validation.

6. If requirements are not met, a new iteration of the design is begun, repeating the steps 3–5 as necessary.

7. Although not strictly part of the design process per se, once complete, system maintenance is often required as part of safe operation. Maintainability is often a component of dependability-driven design, however, and as such efficient maintenance may be a design objective.

2.1.2 Classical safety analysis techniques

As mentioned in the introduction, safety analysis has historically been a largely manual process requiring intensive effort, a high degree of expert knowledge, and consequently a steep cost. In the past, this meant that a safety analysis was typically too expensive to carry out repeatedly, and it was often employed near the end of the design as a kind of acceptance test rather than being integrated as part of an iterative process. Even so, many of the original concepts pioneered back in the 1950-60s are still relevant today, and evolutions of classical techniques like Failure Modes & Effects Analysis (FMEA) and Fault Tree Analysis (FTA) remain important components of modern safety engineering.

As with modern techniques, many classical approaches are based on a form of **probabilistic risk assessment** or **PRA**. A PRA is a systematic approach to identify hazards and evaluate risk and as such is essentially a form of HARA. It involves three main aspects:

1. Identifying possible hazards and failures;

2. Assessing the severity of those hazards;

3. Estimating the probability or frequency of the failures causing the hazards.

2.1.2.1 HAZOP

Various classical analysis techniques exist to fulfil each aspect. For example, HAZOP (hazard and operability study) may be performed to identify and classify hazards [3]. HAZOP was developed in the 1960s in the chemical industry but spread to other similar safety-critical industries. Traditionally, it involves a team of suitable experts with the necessary domain- and system-specific knowledge who then follow a guided process to identify design deviations, possible causes, and likely consequences, along with actions necessary to safeguard against them. However, HAZOP is a very manual process that relies heavily on the expertise and experience of its participants; furthermore, while it does have formal elements, it is often based on an informal understanding of the subject system rather than a formal model, and the findings rapidly become out of date if the system design evolves.

2.1.2.2 FMEA

Alternatively, or in addition, an **FMEA** (Failure Modes & Effects Analysis) may be performed. The goal of an FMEA is to review the components of a system, identifying the potential failures of each one and then assessing the potential effects of those failures. Dating back to the late 1940s [4], it is one of the first structured safety analysis techniques. Like HAZOP, however, an FMEA is traditionally a manual process conducted by human experts, who follow a systematic inductive process on the basis of informal system knowledge [5]. While it can be effective in establishing the risk posed by low-level failures — based on severity, probability, and detectability — it has a number of drawbacks. Foremost amongst these is the inability to consider the effects of combinations of failure modes, which would rapidly spiral out of control if even pairs of failures are considered, let alone other combinations.

Table 1 - Example FMEA table

Item	Failure Mode	Effect	Potential Causes	Sev.	Prob.	Det.	RPN	Action
Smoke detector	Does not detect smoke	Failure to detect fire	Sensor malfunction	8	3	6	144	Replace detectors on a regular basis
			Battery failure		7	3	168	Ensure regular battery testing
Heat sensor	Does not detect heat	Failure to detect fire	Sensor malfunction	8	3	4	96	Replace detectors on a regular basis
			Gremlins		1	8	64	Do not feed Mogwai after midnight
Sprinkler	No water	Failure to extinguish fire	Water supply disruption	6	3	4	72	Add local water storage for emergency use
	Nozzle blockage	Failure to extinguish fire	Dust/debris infiltration	6	2	5	60	Ensure nozzles are cleaned regularly
Alarm	No sound	Failure to warn of fire	Speaker malfunction	7	2	7	98	Test alarm periodically

Even so, FMEA remains an important and useful technique in certain scenarios. In particular, when carried out effectively it excels in enumerating possible low-level root causes. In modern usage, it is often extended to incorporate additional considerations, e.g. as in FMECA (Failure Modes, Effects, & Criticality Analysis) or FMEDA (Failure Modes, Effects, and Diagnostic Analysis).

2.1.2.3 *Fault Tree Analysis*

Fault Tree Analysis or **FTA** [6] is the other major classical safety analysis technique. It was pioneered at Bell Labs in the 1960s for the purposes of evaluating intercontinental ballistic missile systems but has since been used across just about every safety and reliability engineering domain. It rose to particular prominence after its use in analysing the causes of the Challenger shuttle disaster.

Unlike FMEA, FTA is a deductive, top-down process. It begins with the identification of a particular hazard or undesirable event to be analysed, and then works backwards to identify the root causes step-by-step. Since each intermediate event can have multiple possible causes, whether singly or in combination, a tree of Boolean logic is formed from AND and OR gates (e.g. see Figure 1). This lends itself well to hierarchical system structures as well as analysis of failure propagation through a chain of system components. By assigning probabilities and other attributes to the root causes at the leaf nodes of the tree, calculations can be performed to obtain overall probabilities (amongst other things) for the top event of the tree, i.e., the hazard being analysed.
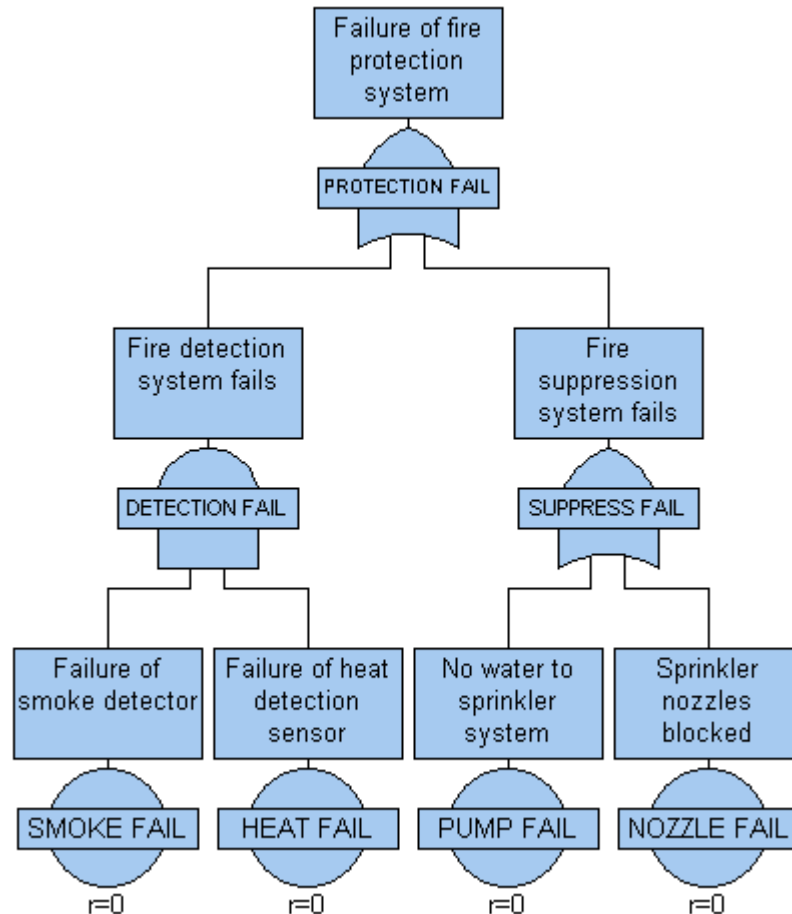
Figure 1 - Example fault tree

As with HAZOP and FMEA, FTA is traditionally a manual process. While software tool support exists and the probabilistic calculations can be automated, creating the fault tree itself is often still a manual process relying on the expertise and informal knowledge of the analysts involved. While it is very effective at investigating the root causes of particular events — and unlike FMEA, is capable of assessing the effects of combinations of failures — it is less effective at capturing all possible failures, relying instead on a suitably exhaustive hazard analysis to identify the initial high-level events to investigate.

2.1.2.4 *Dynamic state-based analysis*

As with FMEA, variations of FTA have been proposed over the years to address shortcomings. One major subset of evolved FTA techniques is meant to address the limitations of purely Boolean logic when analysing dynamic systems; of these, the most prominent is the Dynamic Fault Tree (DFT) approach [7], which extends FTA with additional types of gates to model functional dependencies and sequences of events. Other related approaches exist, such as Pandora [8] and Temporal Fault Trees [9].

Dynamic systems can also be modelled with other network-based modelling techniques. State machines and Markov models are common (and the latter are sometimes used to evaluate probability for DFTs), while Petri Nets and Bayesian networks are also increasingly used.

State machines are perhaps the most popular representations of dynamic behavioural models. Although there are many different versions and languages (e.g. UML, SDL, Statecharts), almost all share the same core elements:

- A state, representing a persisting condition;

- Transitions between them (typically with triggers labelled), indicating events that can cause a change in state;

- Usually, an indication of the starting state.
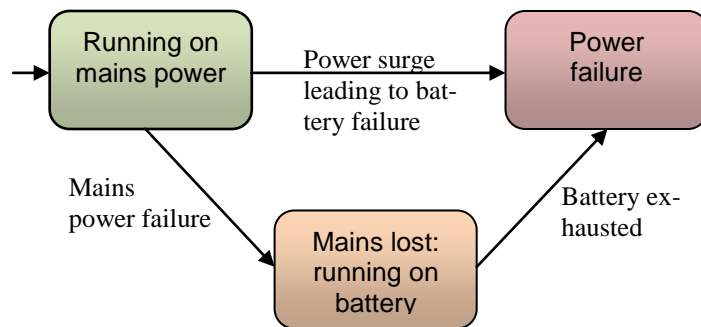
Figure 2 - A simple example state machine

Many state machines are purely visual diagrams (e.g. Figure 2). However, various similar models incorporate additional information to enable more sophisticated analyses. One of the most common examples are Markov chains, which assign each transition a probability:
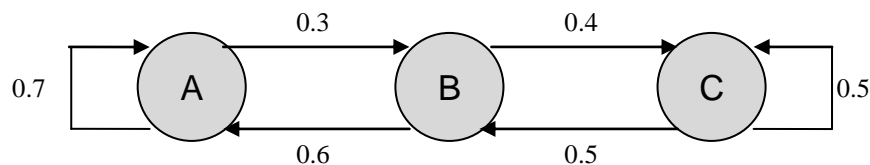
Figure 3 - Example Markov model

In general, a Markov chain adds probabilities to the transitions of a state machine (e.g. see Figure 3). As with state machines, Markov models come in many different forms, though the most common type found in the dependability domain are continuous-time Markov chains (CTMCs), which model stochastic processes using continuous time. The state changes according to an exponential random variable (which effectively models the passage of time) and the new state is decided according to the probabilistic weights of the transitions, typically described by a table known as a stochastic matrix.

Petri nets are another form of dynamic model that can be used to describe state-based systems. As with state machines, there are two main elements: places, represented as circles, and transitions, which are rectangles. The major difference is that Petri nets also have *tokens*, which reside in places. A transition is only available if all of its inputs contain at least one token. Stochastic Petri nets are the most common form found in

dependability, in which the transitions fire after a probabilistic delay according to a random variable, similar to a continuous-time Markov chain.

Finally, Bayesian networks also have two main elements (nodes and edges) which can be used to represent states and transitions. Unlike Markov chains and stochastic Petri nets, in which the probability of the next state is memory-less and independent of any prior states, a Bayesian network operates on conditional probability and each edge indicates a conditional dependency. Conditional probability functions determine the status of each node given the status of the other nodes on which it is dependent. One of the advantages of Bayesian networks over similar approaches like Markov chains is that they are capable of inference and can satisfy probabilistic queries, e.g. inferring the knowledge about an unobservable node based on the status of the others that can be observed.

P(A=T)	P(A=F)
0.7	0.3

P(B=T)	P(B=F)
0.6	0.4

P(D=T)	P(D=F)
0.8	0.2

A	B	P(C=T|A,B)	P(C=F|A,B)
T	T	0.6	0.4
T	F	0.3	0.7
F	T	0.8	0.2
F	F	0.01	0.99

C	D	P(E=T|C,D)	P(E=F|C,D)
T	T	0.9	0.1
T	F	0.2	0.8
F	T	0.7	0.3
F	F	0.05	0.95
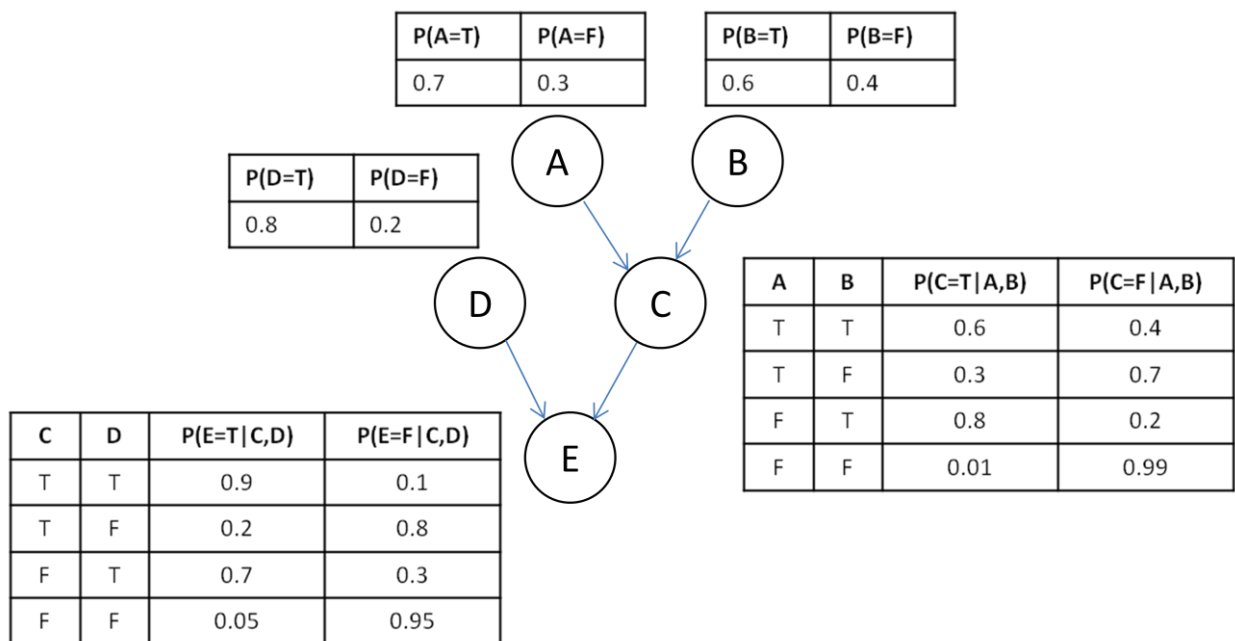
Figure 4 - Example Bayesian Network (from [10])

A general drawback to all of these is the issue of state-space explosion when applied to complex systems with lots of states; even relatively modest systems can soon surpass the scope of human analysts, and even computer software can struggle to evaluate large Markov models or Bayesian networks.

In any case, dynamic or not, the key commonality to all of these approaches is that they are primarily manual and rely on the informal system knowledge of the human analysts. If that knowledge is faulty, or if the design of the system changes, then the resulting analysis can be faulty in turn.

2.2 STATE OF THE ART: MODEL-BASED SAFETY ANALYSIS

Traditional safety analysis methods are well-established and can yield a great deal of valuable knowledge about the safety and reliability of a system, but as mentioned, their relatively informal, ad-hoc, and manual nature limits their effectiveness when applied to increasingly complex modern systems. Safety knowledge is separate from knowledge

about the system structure, which can result in discrepancies and means it quickly grows out of date; manual processes tend to be more error prone and expensive; and reuse of the information gained can be problematic, especially as part of an iterative design process.

Model-based safety analysis, or **MBSA**, has been one of the primary responses to these difficulties. Reflecting the move in engineering towards computer-aided, model-based development, MBSA formalises the various classical techniques by explicitly linking knowledge about the safety and reliability of the system to system design models, whether architectural, behavioural, or both. Ideally, both system and dependability engineering processes share a common model or set of models, meaning all the information about the system is centralised in one place. The resulting models are more formalised and lend themselves much more easily to reuse, analysis, and automation, improving the quality of the dependability analysis process [1].

The benefits are manifold. Centralising knowledge within models helps with coordination and communication because there is a single source of information about the system. Standardised models also help to foster reuse and compatibility of exchange between multiple stakeholders. They also aid in the design process itself by facilitating iterative modifications, with hierarchical structures making it possible for different parts of the system to be worked on simultaneously while preserving the integrity of the overall architecture.

The advantages posed by model-driven engineering help explain the rise of modelling languages and architecture description languages (ADLs) including, amongst others:

- SysML[9], a general-purpose architectural modelling language based on UML;

- AADL[10], an ADL originally developed for aerospace applications but applicable for modelling any software/hardware architecture of embedded, real-time systems;

- AUTOSAR[11], created by a consortium of automotive companies for standardising modelling of embedded automotive software applications;

- EAST-ADL[12], an ADL intended for automotive applications and compatible with AUTOSAR, in effect forming a superset capable of modelling wider automotive systems architecture throughout the design process.

- The ODE[13] (Open Dependability Exchange) metamodel, developed during the DEIS project to support MBSA of cyber-physical systems via the creation of Digital Dependability Identities.

In some of these, some degree of dependability information can be included directly (as in AADL's error annex [11] or EAST-ADL's error model [12]). In others, dependability information can be stored in other models linked with traceability elements to ensure

[9] https://sysml.org/
[10] https://aadl.info
[11] https://www.autosar.org/
[12] https://east-adl.info/
[13] https://github.com/DEIS-Project-EU/ODEv2

that dependencies are maintained. Alternatively, model transformation tools such as ATLAS[14] or Eclipse Epsilon[15] can be used to translate between systems engineering and dependability models [13].

Modelling is just one component of MBSA, however; models alone are insufficient without accompanying analysis techniques. Model-based analysis offers many advantages over ad-hoc traditional approaches, particularly in terms of automation of synthesis (i.e., automatic generation of analyses from dependability models) and in the usefulness of the results (since problems identified can be linked to specific, originating elements of the system model).

Two general paradigms of MBSA analysis techniques have emerged: compositional safety analysis approaches and behavioural simulation approaches [14]. These will both be described below, along with other related techniques such as safety requirement allocation and generation of safety argumentation.

2.2.1 Compositional safety analysis approaches

Compositional safety analysis approaches are generally deductive in nature, generating analysis results by working backwards from system failures to determine root causes, and construct their failure models using a process of hierarchical composition. These models are then typically evaluated with variations on established analysis techniques like FTA.

For the most part, compositional approaches are so named because they compose a system-wide failure propagation model from smaller component-level failure descriptions. These localised failure descriptions may be static (e.g. fault trees) or dynamic (e.g. state machines, Markov chains), but connect together to form the overall model of system failure behaviour.

The key difference with respect to behavioural approaches lies in the form of analysis used. Behavioural approaches make use of formal verification methods like model-checkers to simulate the occurrence and effects of failures with a high level of detail. Compositional approaches require less detail and do not rely on simulation or model-checking, making them better suited for use earlier in the development process, e.g. when only functional information is available and detailed decisions about timing constraints etc. have yet to be made.

Examples of compositional MBSA approaches include Failure Propagation & Transformation Calculus (FPTC), State-Event Fault Trees, Component Fault Trees, and HiP-HOPS.

2.2.1.1 Failure Propagation & Transformation Calculus

Failure Propagation & Transformation Calculus (FPTC) allows modular, compositional representation and analysis of both hardware and software components. In common with most compositional MBSA techniques, the intention is to show how system-level failure behaviour can be derived from component-level behaviour. FPTC accomplishes this by allowing an analyst to annotate components in a model of the system

---

[14] https://www.eclipse.org/atl/
[15] https://www.eclipse.org/epsilon/

architecture with modular failure expressions that describe how each component can fail.

FPTC is intended foremost for the real-time safety-critical software domain, so the primary element of FPTC is a statically schedulable code unit, modelling a single sequential thread of control. Code units are connected via software communications protocols such as handshakes, buffers, and prods etc., each modelled separately. The software architecture itself is statically determined, i.e., everything is known to the designers and assumed to exist on initialisation, with no dynamic creation or destruction during operation. Allocation of software units to the underlying hardware & network architecture is also taken into account, thereby making it possible to model e.g. the effects of a damaged processor on the software it is executing.

The system architecture is represented using a Real-Time Network (RTN) style notation. Essentially, an RTN is a graph consisting of nodes and arcs, where nodes represent hardware or schedulable code units while arcs are the communications between them, with the label being the type of protocol (e.g. blocking, non-blocking etc). RTN models can be hierarchical and can define code units as state machines, as well as allow automatic code generation.
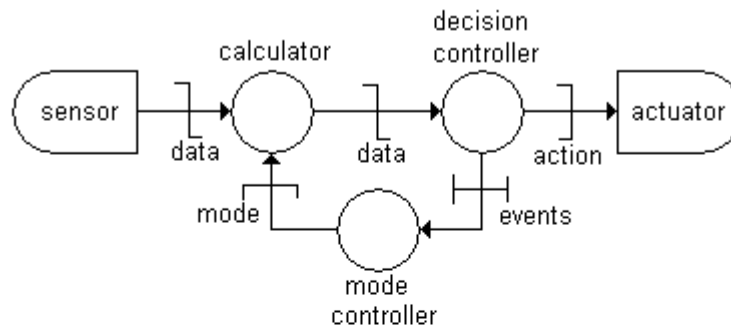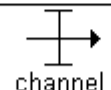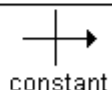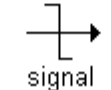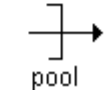
Figure 5 - Example RTN graph (from [15])

The FPTC process proposes a HAZOP-style system of guidewords (which can be tailored to the specific system or domain) and an inductive, FMEA-esque approach to studying each individual component. Each unit is considered in turn, with analysts determining how it originates failures or behaves in response to potential input failures of different types, e.g. timing failures, value failures, sequence failures etc. "Normal" behaviour is also part of the set of possible failure types (i.e., a lack of failure), which allows absorption of an input failure where appropriate, e.g. in the case of components capable of error correction. In general, component behaviours are either "propagational"

(i.e., failures are transmitted from input(s) to output(s)) or "transformational" (the failure type is transformed from one type to another during the course of propagation).

A functional pattern-based notation is used to represent this behaviour. For example:

```
omission → late
```

means that a component transforms an input failure of type "omission" into an output failure of type "late".

```
early → *
```

The * symbol indicates "no failure" (i.e., normal behaviour), so this expression indicates that the component is capable of correcting an input failure of type "early" with no effect on output behaviour. Combinations of values can be handled with tuples, e.g.:

```
late → (value, late)
```

Wildcards can be used to represent arbitrary inputs, e.g. (omission, _) means an omission failure at the first input and any other type of failure — including no failure — at the second input.

Both components and the connections between them are annotated with as many of these expressions (known as "clauses") as is needed to describe their behaviour in response to all the different types of failure. Connections are included since the protocols involved may also be capable of originating or modifying the propagation of failures.

The resulting annotated RTN model then effectively becomes a token-passing network, in which tokens represent failures and are passed from one node to another (potentially being transformed or destroyed along the way). An FPTC analysis therefore functions as a kind of simplified simulation, in which all of the clauses for each component are "executed" and any resulting output failures passed from the outputs, potentially triggering other clauses in connecting components. The process terminates once no new output failures are generated and all relevant clauses have been considered.

Because FPTC makes use of an existing software design model (RTN), both nominal and dependability information about the system is centralised in a single model. This makes it easy to adapt to changes to the design: when new components are added, they merely need to be annotated and the analysis re-run to see the impact of the changes.

Like FMEA, however, FPTC is an inductive process that works best when analysing the effects of individual failures; its "simulation" must be re-run for every originating clause of every component in order to achieve coverage, and even then this does not take into account the potential for combinations of failures. Cases where a critical system failure occurs as a result of two or more component failures occurring in conjunction can easily be missed.

2.2.1.2 Component Fault Trees

Component Fault Trees (CFTs) are an MBSA approach that aims to introduce hierarchical decomposition to fault trees by integrating them within the hierarchy of the system architecture [16], [17]. In effect, each component of the system is represented by its own extended fault tree (i.e., an eponymous Component Fault Tree). Overall system failure logic is obtained by connecting these component fault trees via ports, which represent the interface of a component. CFTs are still recognisably fault trees and can thus be analysed with standard FTA algorithms.

One of the primary motivations behind CFTs is the concept of reuse: rather than create an entire system architecture and accompanying dependability model from scratch for each new design, a library of components — each stored with their associated dependability information (i.e., their component fault trees) — can be made available, and relevant components can simply be imported. There are many advantages to such an approach; most obviously, this reduces effort, since a component only needs to be modelled once and any subsequent designs that use that component can simply import it. Different components can also be worked on separately and simultaneously. This form of reuse also potentially cuts down on modelling errors (assuming it was modelled correctly in the first place) and also allows future designers to make use of the components without necessarily possessing expert knowledge of those components — though it could be argued that this is not always a good thing.

The decompositional approach offered by CFTs also lends itself well to an iterative, top-down development process in which an initial, more abstract, top-level model is created first and then progressively evolved as the design matures by adding new hierarchical levels with more detail. This is the approach taken by EAST-ADL, for instance, with its initial feature and functional layers giving way to more detailed, lower-level hardware layers.
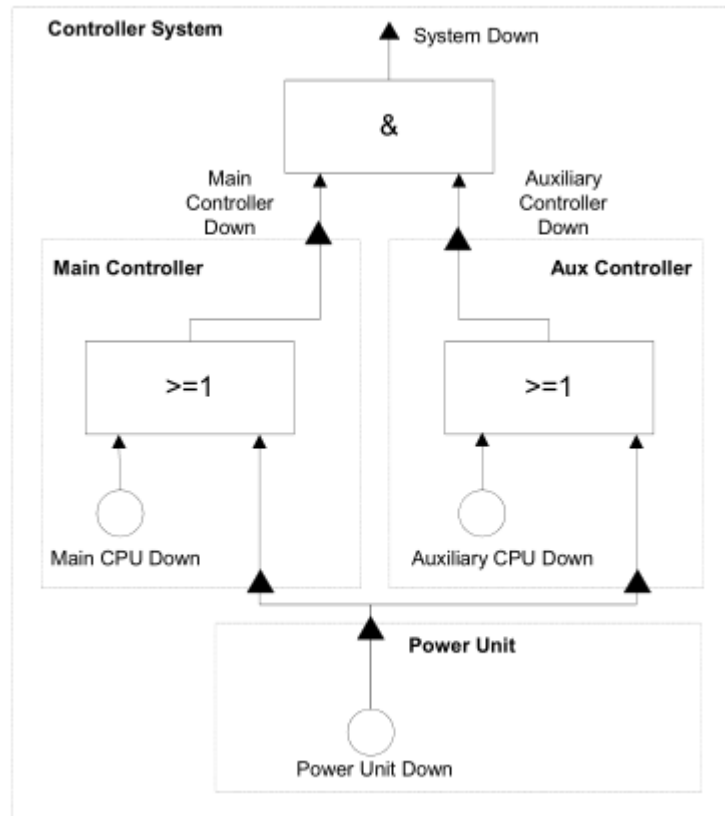
Figure 6 - Example CFT (from [16])

The fact that the fault trees are component-based means that branches may be shared, as in the figure above, acting as a common cause for multiple components. In this sense, CFTs are technically directed acyclic graphs, called Cause Effect Graphs by the authors. Unlike normal fault trees, multiple top-level system failures are also possible; in effect, the CFTs form a "forest" of classical fault trees, with roots and branches being shared across multiple trees with different outputs.

Although this may require some adjustment, in most cases it is still possible to represent the graph for a given system failure with a Boolean formula and thus evaluate it using traditional FTA algorithms, though in practice this means exporting the CFTs to a suitable FTA tool, thus potentially losing some of the traceability between the analysis results and the originating model.

Equally, however, the similarity between CFTs and classical fault trees mean CFT also inherits most of their disadvantages, particularly in terms of expressivity and ability to handle dynamic or temporal scenarios. Perhaps for this reason, various attempts have been made to extend or adapt CFTs with new functionalities, such as Generalized Hybrid Component Fault Trees or State/Event Fault Trees, both discussed below.

2.2.1.3 *Generalized Hybrid Component Fault Trees*

Generalized Hybrid Component Fault Trees (GHCFTs) are an evolution of CFTs intended to incorporate new features to enable the modelling of temporal or state-dependent behaviour [18]. The idea is to combine the best features of fault trees, which excel at modelling combinatorial failures, and Markov chains, which can handle stochastic state-based analysis, to produce an integrated solution: GHCFTs.

As with CFTs, the key principle is compositionality. But where CFTs made use of compositional component-based fault trees, GHCFTs introduce Component Markov Chain (CMC) elements to fulfil a similar role. CMCs are intended to be modular, compositional, and reusable, and when used in combination with normal CFTs, both combinatorial and sequential/state-based failure behaviour can be modelled within the system architecture.
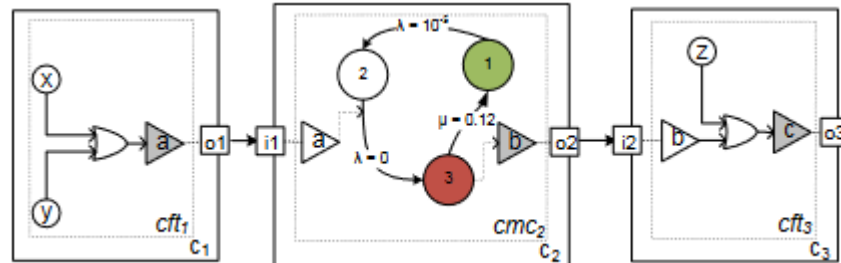
Figure 7 - Example GHCFT (from [18])

The advantage here is that static, non-dynamic components can be modelled using simple CFTs, while the more complex dynamic components can make use of CMCs. This helps limit the scope of the state-based modelling required and to a degree therefore helps to contain the state-space explosion problem. Since CMCs and CFTs are bound within their parent components and largely isolated from each other, they interact through the ports that comprise the component interfaces: failures propagate from the output of one component and are received at the input of the next with no need for knowledge of whether they originate in a CFT or a CMC.

Both qualitative (logical) and quantitative (probabilistic) analysis of GHCFTs is possible. In the former case, CMCs are converted into CFTs as follows:

- Every transition between two states is converted to a basic event.

- If the transition is connected to an input failure, then an OR gate is created and all input failure modes become children of the gate.

- For each output failure mode of the CMC, every path leading from the initial state to an error state connected with the output is enumerated and all transition-derived events encountered along the way are collected under an AND gate. If multiple paths exist, each becomes an AND gate and all are fed into a top-level OR gate.

This algorithm effectively flattens a CMC into a static fault tree, enabling it to be readily combined with existing CFTs and analysed using traditional FTA algorithms. The downside is that all dynamic information is lost in the process, and as such the qualitative analysis results are purely combinatorial, which can be misleading in some cases.

The quantitative analysis, on the other hand, typically employs a numerical integration scheme with step-size control for the CMCs. GHCFTs restrict themselves to Continuous Time Markov Chains (CTMCs) with constant transition rates; in most cases this is

Confidentiality: Public Distribution

sufficient, but does mean that cases with more exotic failure distributions (e.g. Binomial, Poisson, Weibull, or other variable failure distributions) are incompatible.

One important drawback is that common cause failures arising from repeated events (i.e., events that appear more than once in a fault tree) are incompatible with Markov-based analysis of the CMCs; unfortunately, this converts one of the advantages of CFTs (natural handling of common causes) into a disadvantage. Furthermore, in common with other state-based approaches, the state-space explosion problem can become an issue for larger models with many states, although the compositional nature helps keep them contained such that individual CMCs in theory can be analysed separately.

2.2.1.4 State/Event Fault Trees

The State/Event based Fault Tree (or SEFT) is an attempt to remedy one of the major drawbacks of classical FTA: its inability to model dynamic scenarios in which specific sequences of events or state transitions can cause failures. SEFTs are intended to overcome this limitation by introducing new capabilities for representing states and more complex types of events [19], [20]. In this sense, they are an evolution of the Component Fault Tree approach described above.

SEFTs allow direct modelling of states and transitions on a per-component basis. States are interpreted as conditions that prevail for a given period of time; events by contrast are defined as being instantaneous and may trigger state changes. This means that dynamic system failure behaviour can be modelled directly, without resorting to the use of separate types of models (e.g. an architecture model and a behavioural model) or relying solely on static dependability models.

In SEFTs, events occur in one of three ways:

- they may be triggered by other events;

- they occur after a deterministic delay;

- or they occur after an exponentially distributed probabilistic delay, beginning when the preceding state is entered.

There is a distinction between an *event* and its *occurrence*: the former represents a class of events that can occur at different times, while the latter indicates a specific occurrence of an event at a given time.

As in FTA, gates act as junctions and allow different operators to be applied to multiple inputs, meaning the effects of combinations of failures is also modelled. Unlike traditional FTA, however, the gates are not limited to purely Boolean operators. A distinction is also made between purely causal relations and temporal/sequential relations; the former apply to events while the latter to states.
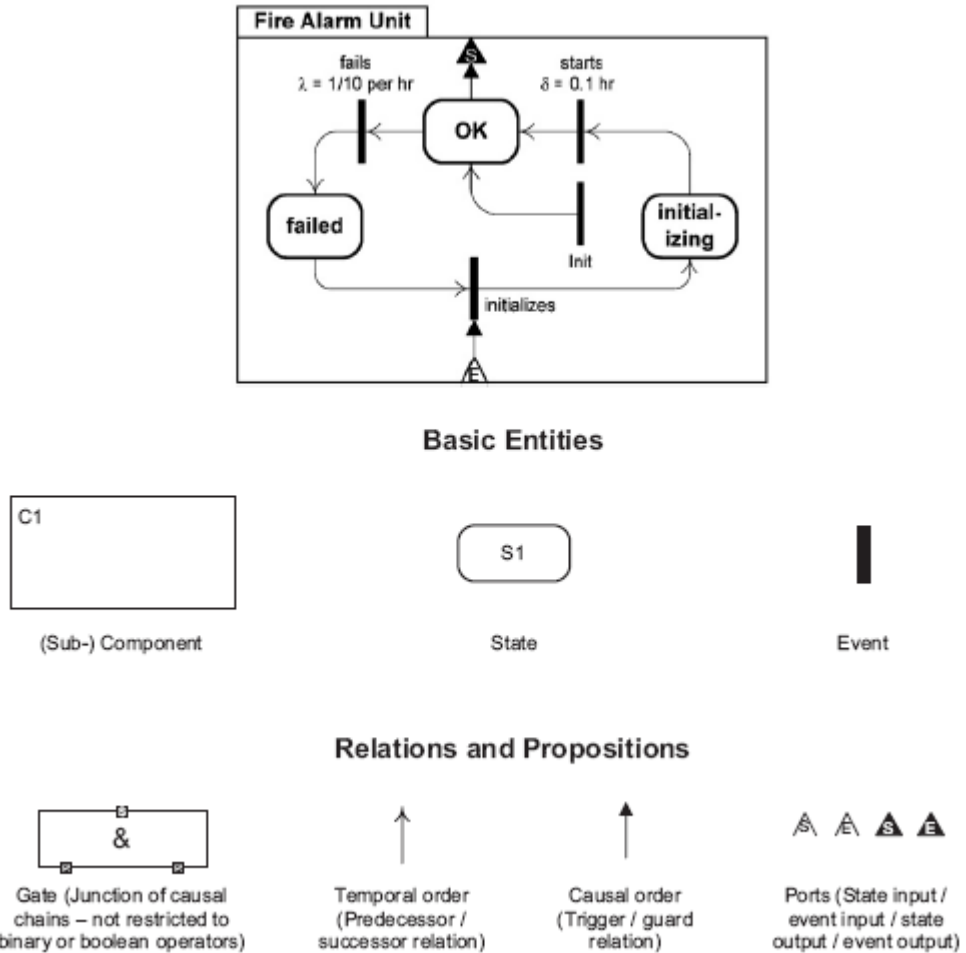
Figure 8 - SEFT notation (from [19])

Each component has its own state (and thus can only be in one state at any given time). As with the other compositional approaches, they also allow decomposition: each system or component can be decomposed into subcomponents, each in turn with their own state-transition mechanisms. Interfaces of components are defined via ports, which are typed according to input/output and whether they are state ports or event ports.

Probabilities can be assigned to states as well as events, indicating the likelihood of a given state being the active state for that component. Gates can also apply to states, thus enabling the modelling of scenarios where e.g. two components must be in given states in order for some other event to occur.

In addition to basic Boolean AND and OR operators, more complicated timing scenarios are handled with a variety of dynamic and temporal operators. Delay Gates, for instance, can be used to model deterministic or probabilistic delays between events. History-AND gates check whether an event has previously occurred, while Priority-AND gates check which order a set of events occurred. Duration gates can be used to test whether a state has been active for a given duration.

Given the breadth of its modelling entities, SEFTs effectively act as a kind of superset of both fault trees, state machines, and Markov chains, capable of combining them all and integrating them into a representation of the system architecture.

While the process of creating a SEFT model is primarily component-based, where the behaviour of each component is considered in turn, SEFTs are not an inductive technique. Unlike FPTC, SEFTs are intended to function in a deductive FTA-like fashion, where system-level failures are traced back through the system from one component to the next. This ensures that failures with combinations (or sequences) of causes can be handled correctly. The inclusion of states also means that more complex scenarios and triggering conditions are modelled, e.g. in terms of considering what state a component must be in for a given event to occur.

Analysis of SEFTs is typically achieved via conversion to Deterministic Stochastic Petri Nets (DPSNs), which are an extension to general Stochastic Petri Nets with added functionality for modelling deterministic delay [21]. The DSPNs are then evaluated separately, e.g. via the TimeNET tool [22]. Conversion from SEFT to DSPN is achieved using the ESSaRel (Embedded Systems Safety and Reliability Analyser) tool, which sadly now appears to be defunct.

The separation between the model and the DSPN analysis tool is not ideal, since some of the traceability between the model and the analysis results is lost and troubleshooting errors is made more problematic. And as with most state-based approaches, the state-space explosion problem can be an issue for larger models with many states. While SEFTs offer compositionality and the states are per-component, when converted to the resulting DSPNs, these nuances are lost as the model gets "flattened". Thus there is a trade-off between the wide range of capabilities SEFTs offer and the limitations in the analysis support available.

2.2.1.5 *HiP-HOPS*

HiP-HOPS, or "Hierarchically Performed Hazard Origin & Propagation Studies" to give it its full title, is a comprehensive model-based safety analysis methodology with a tool of the same name. Originally developed in the late 1990s [23], it has been the focus of continuous development over the ensuing 20+ years and its initial foundation has since played host to a wide range of advancements and additional functionalities [24],[25].

Like CFTs and SEFTs, HiP-HOPS uses fault trees as a foundation. The central principle is the integration of data about component failure behaviour into a system architecture model, which facilitates the synthesis of fault tree-based failure propagation models that can then be further evaluated to produce FTA and FMEA results.

The core HiP-HOPS methodology consists of four main phases:

- System modelling

- Failure annotation

- Synthesis of fault propagation models

- Fault tree analysis & FMEA synthesis

Phase 1: System Modelling

System modelling consists of developing a system architecture modelling along the lines of a block diagram, featuring components (which may be hardware, software,

purely functional, or just about anything else) and the connections between them (again, these may be hydraulic, electrical, electronic, mechanical, data-based, or other). Components may have subcomponents, allowing a compositional system hierarchy to be built up. Various system modelling tools are compatible with HiP-HOPS, including Matlab Simulink[16], SimulationX[17], and MetaEdit+ (with EAST-ADL)[18]. Export and model transformation to HiP-HOPS format is also possible for various other modelling languages, e.g. AADL[26].

Interfaces for components are defined via ports. Unlike SEFTs, ports do not have a type, nor are they necessarily restricted to either input or output only. Instead, input and output flow is defined via connections, as will be explained shortly.

Support also exists in HiP-HOPS for "multi-perspective" modelling. This is intended to facilitate modelling of multiple interconnected layers (or perspectives), e.g. different software and hardware layers with allocation from one to the other, as found in languages such as EAST-ADL[12]. These allocations present new lines of possible failure propagation, similar to how FPTC allows modelling of the effects of hardware processor failure on software it is executing.

Phase 2: Failure data annotation

The next phase is the annotation of components and connections with logical expressions that describe local failure behaviour, similar in principle to FPTC or CFTs. These expressions indicate how deviations at component outputs (*output deviations*) are caused by some combination of corresponding *input deviations* or internal failures of that component. Annotation is possible at all levels of the system hierarchy; output deviations at a component's output may be caused by inputs or internal failures of that component, or by output deviations deriving from subcomponents within that component.

Input and output deviations are also assigned a *failure class*. This facilitates matching of deviations at different ports. Typical classes include omissions (i.e., no input/output when it was intended), commissions (i.e., unexpected input/output), timing errors (e.g. late, early), and value errors (e.g. high, low, or just a wrong value), but classes are user-defined and additional failure classes can be added as required. As well as the failure class, deviations also refer to the component and port at which they occur, e.g.:

- `omission-sensor.out` (omission at port 'out' of component 'sensor')

- `value-brake.in2` (value error at port 'in2' of component 'brake')

Internal failures are given a name that is unique within the component, though to avoid ambiguity are often referred to with their parent component's name, e.g.:

- `valve.blocked`

- `switch.stuckOpen`

Together, deviations and internal failures can be combined into logical expressions that define the causes of output deviations, e.g.:

- `omission-valve.out = valve.blocked OR omission-valve.in`

As with FPTC, there is no requirement for the same failure class to be used on both sides of the expression: it is possible (and indeed not uncommon) for a failure class to transform as it is propagated from input to output. For instance, a controller may be able to mitigate sensor failures by failing silent when erroneous input is detected, omitting its output instead of passing along incorrect data:

- `omission-controller.out = (value-in1 AND value-in2) OR (value-in1 AND value-in3) OR (value-in2 AND value-in3)`

Boolean operators AND and OR are most typical, but other gate types are available with varying degrees of support, from non-coherent gates (NOT, XOR) to temporal gates like Priority-AND, Priority-OR, and Simultaneous-AND. As mentioned earlier, it is also possible for an output deviation to be caused by e.g. a failure propagating along an allocation relationship, or via a common cause failure (CCF) that is not specific to any given component.

Potential CCFs are defined per component and then explicitly linked to actual CCFs defined at the system level; for example, multiple components in the same physical compartment of a ship may refer to a potential CCF "Flood", but this only becomes a valid cause once connected to an actual CCF. This helps support reuse, since a component can be used in environments where its potential CCFs are inappropriate (e.g. in a situation where flooding is not possible).

In addition to these logical descriptions of cause and effect, probabilistic failure data can be added to failure events such as internal failure modes and CCFs. A variety of different failure distributions are supported, from simple exponential failure rates or MTTF/MTTR to Poisson, Binomial, and Weibull models.

The process of annotating the model with failure data is typically performed component by component, in which all possible output deviations are considered for each output port, their causes evaluated, and the necessary input deviations and internal failures defined accordingly. In doing so, the failure behaviour of the components is defined, illustrating how they generate, mitigate, propagate, or transform failures across their inputs and outputs. The result is a virtual table like the following:

control		
Valve Malfunctions		
Failure mode	*Description*	*Failure rate*
blocked	e.g. by debris	1e-6
partiallyBlocked	e.g. by debris	5e-5
stuckClosed	Mechanically stuck	1.5e-6
stuckOpen	Mechanically stuck	1.5e-5

Deviations of Flow at Valve Output		
Output Deviation	*Description*	*Causes*
Omission-b	Omission of flow	blocked OR stuckClosed OR Omission-a OR Low-control
Commission-b	Commission of flow	stuckOpen OR Commission-a OR Hi-control
Low-b	Low flow	partiallyBlocked OR Low-a

Figure 9 - Example of a HiP-HOPS component failure annotation

This table shows the various data annotated for a simple valve. Failure modes are defined, along with descriptions and failure rates (using an exponential distribution with no repair rates, in this case). And three possible output deviations are defined, each with a different logical expression describing the cause.

Note that these annotations are generic in the sense that they do not reference the surrounding context in which the valve operates: no knowledge of the immediate environment is needed beyond the component's interface ports. This helps with reuse of failure data, in the sense that the same logic can be used for any other valve of similar design.

Although annotation is done per component, at its heart HiP-HOPS is a deductive technique, and as such it is best employed when starting with top-level system failures (annotated as hazards) and working back to determine which high-level output deviations may cause them. The causes of those output deviations may then be investigated further in turn.

In addition to components, connections can also be annotated with failure behaviour. Unlike components, however, connections are assumed not to *originate* failures: they have no internal failure modes. If such behaviour is required, they should be modelled as components in their own right (e.g. a network bus or a pipeline subject to blockage). Connections can, however, propagate different failure classes in different ways. For example, a 2-to-1 connection joining two outputs to a single input may use OR logic for commission failures (a commission at either output will be received at the input) but AND logic for omissions (only an omission at both outputs will result in an omission at the receiving input port).

This custom propagation logic also provides better support for different types of connections; mono-directional, bi-directional, and many-to-many connection types are all possible in HiP-HOPS.

Annotation of the system models can be performed via interfaces with supported modelling tools. Alternatively, an annotated model file in XML format can be generated directly and provided to HiP-HOPS.

Phase 3: Synthesis of fault propagation models

Although phases 1 & 2 must be performed manually (albeit with tool support), phase 3 is entirely automatic. Once the system model has been suitably annotated with component failure behaviour, HiP-HOPS can perform a deductive investigation to create a network of interconnected fault trees that relates top-level hazards to their root causes, with the path of propagation being maintained via intermediate nodes.
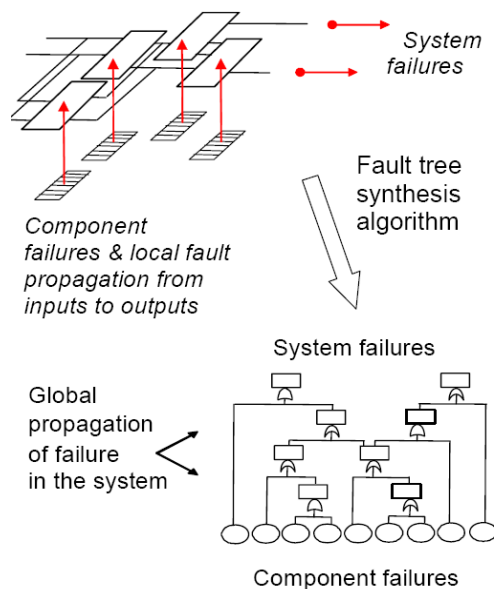
Figure 10 - HiP-HOPS synthesis phase

The process is as follows:

1. For each hazard, link to the top-level output deviations that cause it.

2. Traverse the output deviation backwards to find any input deviations.

3. Using the connection logic, connect the leaf nodes (input deviations) with the corresponding output deviations of the matching failure class at the other end of the connections.

4. Continue from step 2 until no further input deviations are found.

5. If at any stage output deviations of the same failure class are found on multiple levels of the system hierarchy, connect them with an OR gate.

6. Allocation propagations are traced backwards to their originating model perspective, where synthesis continues as normal.

As an example, consider the following system:

Confidentiality: Public Distribution
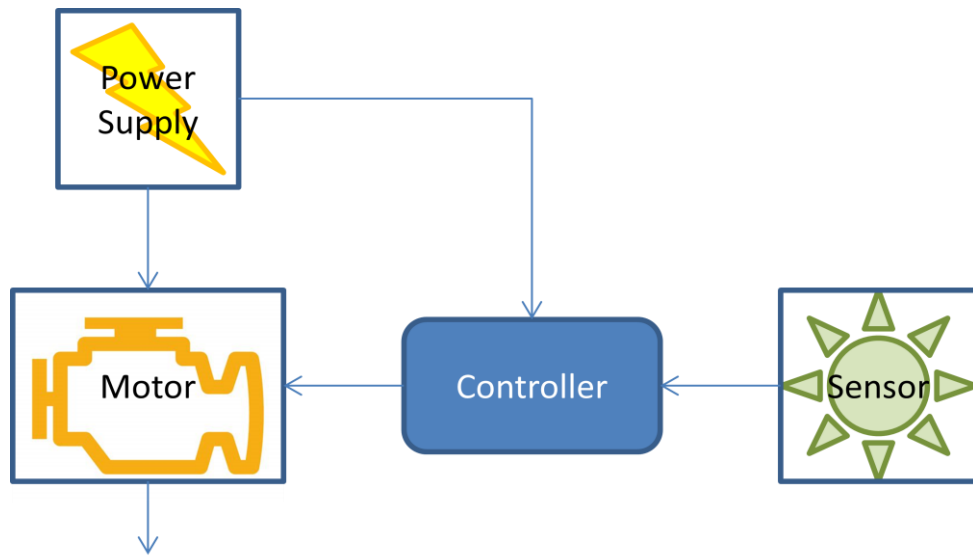
Figure 11 - Example system

The component failure logic for the system can be found in the table below, where "O" is shorthand for an Omission failure and "V" for a value failure.

Component	Output Deviation	Cause
Power supply	O-motorPower	PSU_failure OR mains_supply_failure
	O-controllerPower	PSU_failure OR mains_supply_failure
Motor	O-motivePower (system output)	Motor_failure OR O-power OR O-control
Controller	O-output	Controller_failure OR O-power OR V-sensor OR O-sensor
Sensor	O-data	Sensor_failure
	V-data	Sensor_misalignment

If the system-level hazard is defined as "No motive power from motor upon demand", then we start by investigating the omission of output from the motor:

- O-motivePower

We can then substitute this for its expression:

- Motor_failure OR O-power OR O-control

And expand each input deviation until none are left:

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR O-control

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR (Controller_failure OR O-power OR V-sensor OR O-sensor)

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR (Controller_failure OR O-power OR V-sensor OR O-sensor)

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR (Controller_failure OR (PSU_failure OR mains_supply_failure) OR V-sensor OR O-sensor)

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR (Controller_failure OR (PSU_failure OR mains_supply_failure) OR Sensor_misalignment OR O-sensor)

- Motor_failure OR (PSU_failure OR mains_supply_failure) OR (Controller_failure OR (PSU_failure OR mains_supply_failure) OR Sensor_misalignment OR Sensor_failure)

This results in a simple fault tree showing all possible causes of the hazard "No motive power from the motor upon demand":
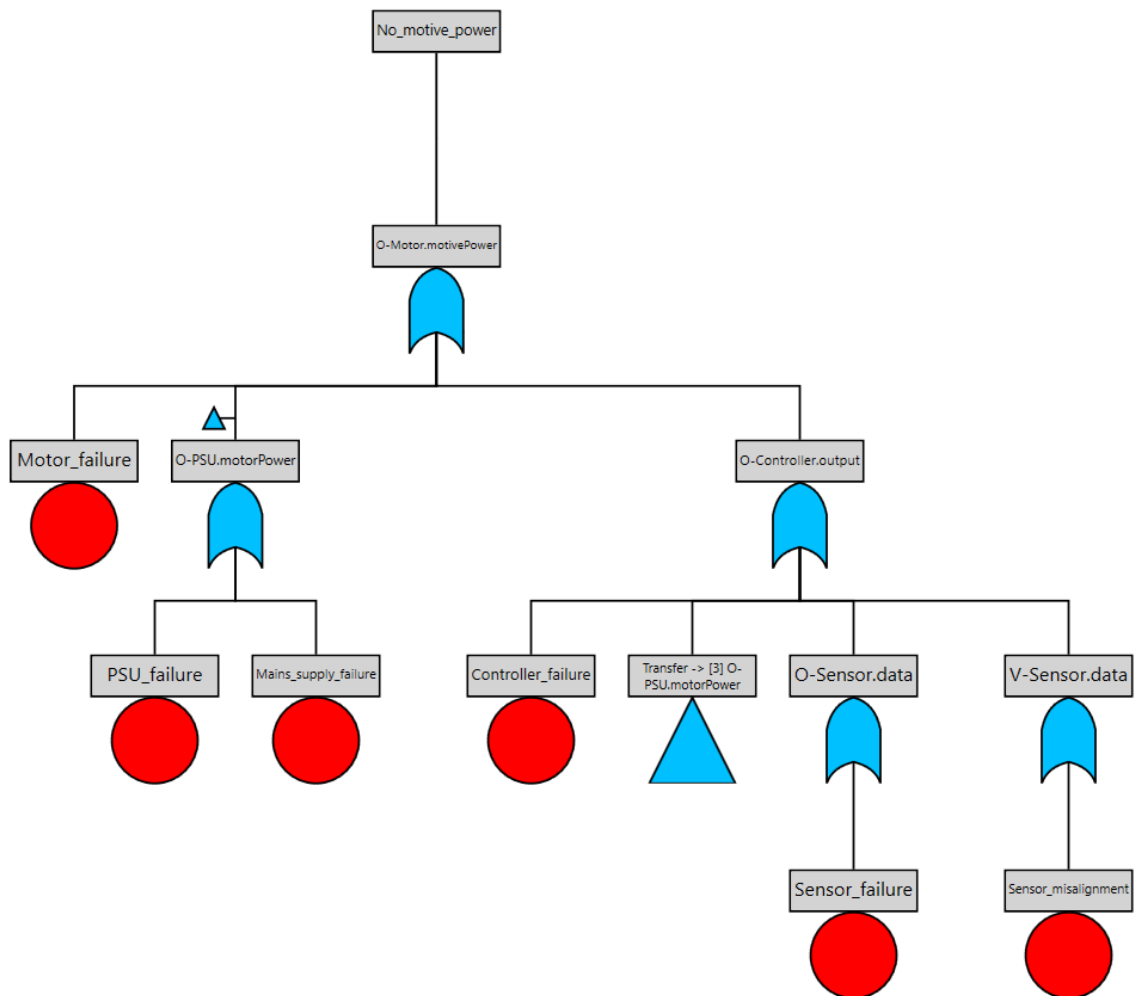
Figure 12 - Example synthesised fault tree

Although the fault tree above is simplified to save space, it still maintains the path of propagation through the system model. For example, we can see how the failure mode

"sensor misalignment" propagates through the sensor, then the controller, and finally the motor in order to cause the hazard.

Although the interconnected fault trees generated during the synthesis phase can provide useful information about the global failure behaviour of the system in themselves, to obtain the full benefit, the resulting fault trees must be analysed.

Phase 4: Fault tree analysis & FMEA generation

HiP-HOPS employs various FTA algorithms to analyse the generated fault trees. Classical algorithms are not directly applicable due to the nature of the trees; loops are possible, repeated events are common, and there may be dead end branches in the trees (e.g. if there is no corresponding output deviation to provide input to an input deviation). Therefore analysis includes several pre-processing steps before the fault trees can be fully analysed, whether qualitatively or quantitatively. Qualitative analysis involves obtaining the minimal cut sets, while for quantitative analysis, an estimate of the probability of the top event is obtained (along with probabilities for all the cut sets).

For the example fault tree above, the cut sets are very simple since there are no AND gates. However, two basic events — PSU_failure and Mains_supply_failure — each occur twice, resulting in a redundancy that must be addressed. The minimal cut sets are thus:

- Motor failure

- PSU failure

- Mains supply failure

- Controller failure

- Sensor failure

- Sensor misalignment

Had we provided failure rates for these basic events, an estimate of top event probability would also have been calculated.

An example of HiP-HOPS output for FTA is shown below:
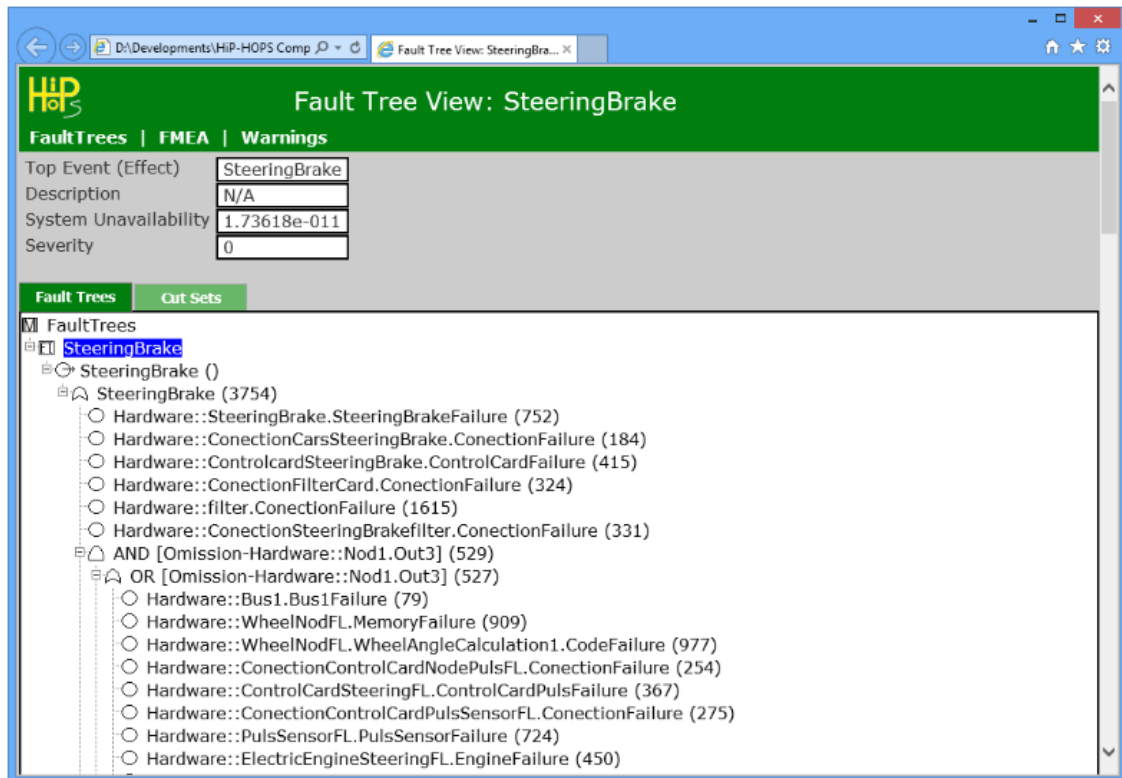
Figure 13 - HiP-HOPS FTA results

Minimal cut sets are useful because they show the necessary and sufficient causes of the hazard. However, sometimes it can also be useful to obtain inductive information as well. HiP-HOPS achieves the best of both FTA and FMEA by generating the latter from the analysis results. Unlike a regular FMEA, a HiP-HOPS FMEA also considers the effects of combinations of failures. For each failure mode of each component, the hazards it causes by itself are displayed (referred to as "direct effects") as well as any hazards caused in conjunction with other failures (referred to as "further effects").

This is especially useful in that it highlights which failures only have further effects (i.e., they never cause system hazards by themselves) and which failures can cause *multiple* system hazards, as in the case of C5 in Figure 14 below.
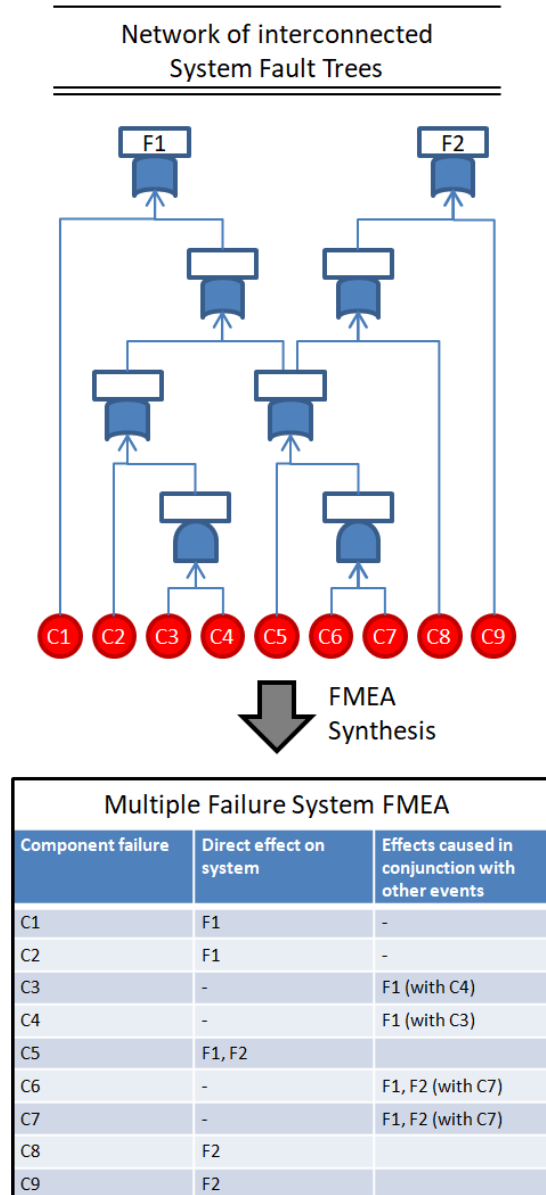
Figure 14 - FMEA generation in HiP-HOPS

Other features

HiP-HOPS has been extended many times with a wide range of additional functionality, including:

- Architectural optimisation [24], allowing optionality to be defined within the system architecture (by defining alternative implementations of components with different cost and reliability characteristics) that can then be subjected to a multi-objective optimisation process to achieve an optimal balance of cost and reliability.

- Automatic allocation of safety requirements using the propagation logic as a basis. If a strong requirement is assigned to a system function, HiP-HOPS can determine how that requirement can be decomposed amongst the (sub)components of the system that fulfil that function. Prototype support for decomposition of ASILs [27] and DALs [28] exists.

- Integration of HiP-HOPS with model-checking technologies, thereby combining HiP-HOPS's compositional safety analysis capabilities with the behavioural simulation capabilities of model checkers [29].

- Incorporation of various approaches for dynamic safety analysis via dynamic fault trees [30].

- Experiments with using HiP-HOPS to perform security analysis with attack trees, thus introducing another pillar of dependability [31].

2.2.1.6 safeTbox

safeTbox[19] (Safety Toolbox) is an extensive model-based safety analysis approach (and tool) that combines architecture design, hazard analysis, fault analysis, and safety argumentation all in a single package. It places a strong emphasis on reusability, traceability, and maintainability of safety-related artefacts. Integrating with existing UML and SysML modelling tools like Enterprise Architect and MagicDraw, it enables integration of dependability data with system architecture information.

safeTbox has evolved over time, growing out of earlier projects such as I-SafE [32]. System modelling is based on UML/SysML block diagrams in which component interfaces are defined using ports which can be connected together. This is then followed by a model-based hazard and risk assessment (HARA) along the lines of that mandated by ISO 26262, and which benefits from integrated traceability by being able to link to pertinent architectural elements.
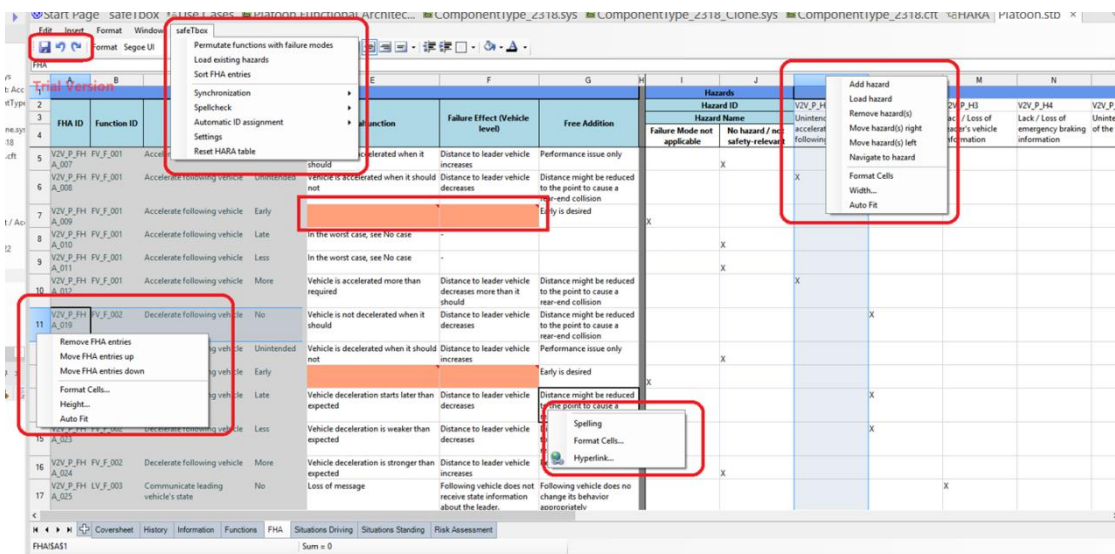
Figure 15 - HARA in safeTbox

---

[19] https://www.safetbox.de/

A process of annotating the model with more detailed safety information then follows. The nature of the failure data added to the architecture depends on the desired form. For example, safeTbox supports Component Fault Trees, enabling the representation of component-level fault propagation and failure generation.
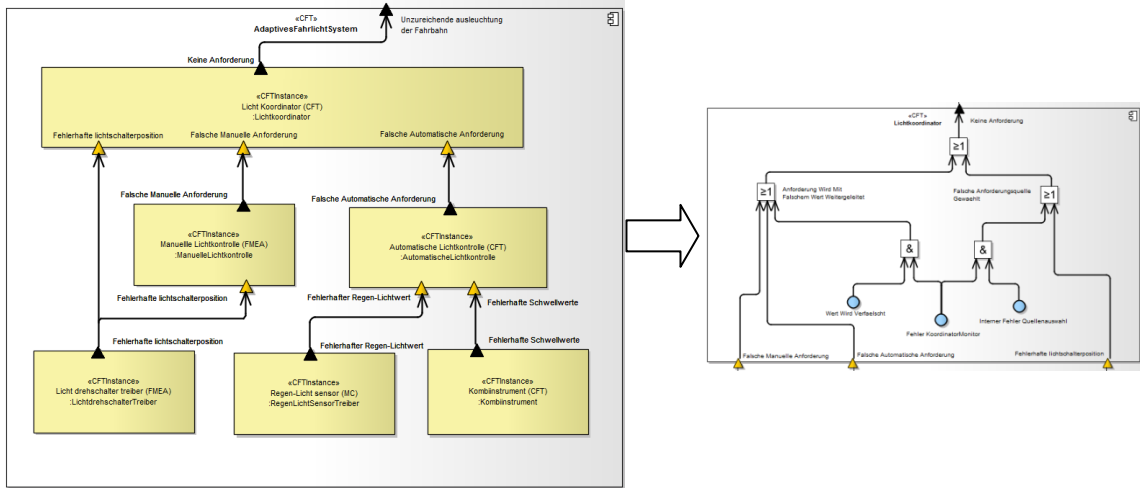
Figure 16 - Generating CFTs from an annotated system architecture in safeTbox

Both qualitative and quantitative analyses are supported. The former produces minimal cut sets on the basis of a system-wide FTA, while the latter evaluates the top-event probability for high-level system failures. Integration with widely used FTA tools such as Isograph's Fault Tree+ (now part of Reliability Workbench[20]) enable rapid evaluation with powerful capabilities.

Figure 17 - Results of a Component FTA in safeTbox (from https://www.safetbox.de/blog)

---

[20] https://www.isograph.com/software/reliability-workbench/fault-tree-analysis-software/

The focus on traceability and maintainability is important because it also supports the generation of safety argumentation such as safety cases that can be used as part of the certification process. Rather than using separate documents, in SafeTbox the safety cases are integrated with the rest of the model, reducing the risk that the documentation becomes out of date and ensuring that the various concepts are interlinked for ease of use. SafeTbox further supports generation of safety cases by directly integrating Goal Structuring Notation[21], widely used for safety cases (see Section 2.2.4.1 below).

2.2.1.7 *Dymodia*

The Dymodia tool[22] is an MBSA software tool that allows architectural, behavioural, and failure modelling and analysis to be combined as part of a single platform. As with other compositional approaches, it is built on the concept of being able to annotate system components with logic that describes their failure behaviour. However, rather than try to embed behavioural models (in the form of state machines) as part of this component-level logic, as in SEFTs or GHCFTs, in Dymodia the state machines are separate models that link to a given system model. Then either different failure logic can be defined per state or generic failure logic can be used to describe state-independent behaviour instead. Standalone fault trees can also be defined to model failure-related behaviour that does correspond directly to system architecture elements or system states.
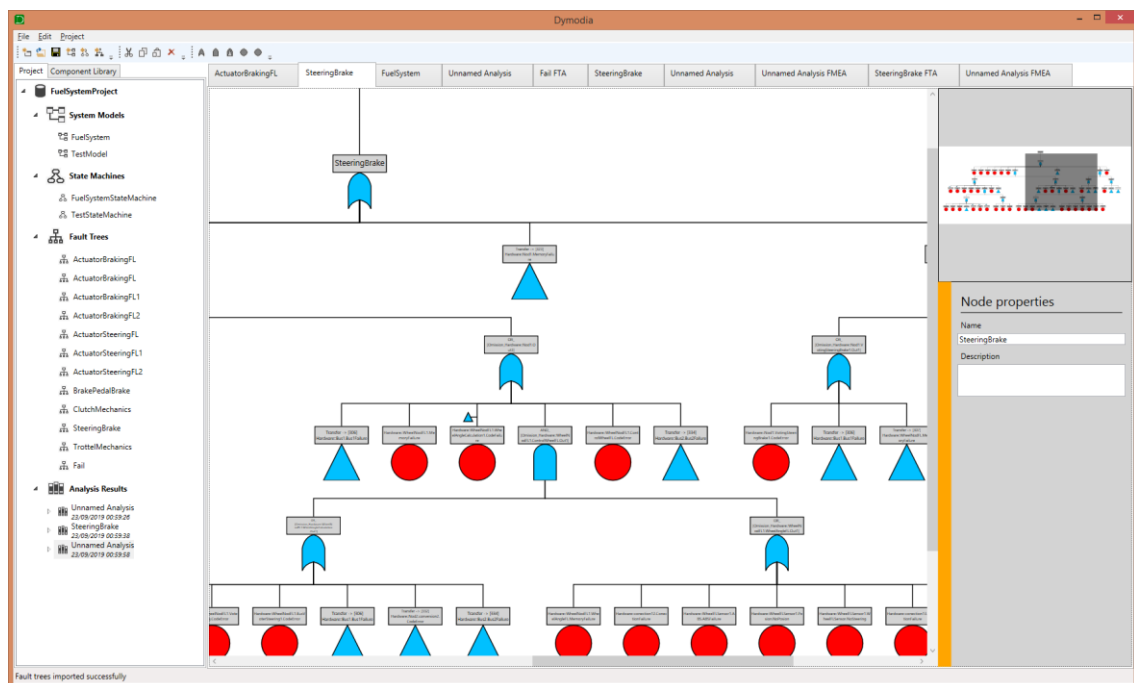
Figure 18 - Dymodia

Links and cross references are possible across multiple models. For example, a state machine transition could be triggered by the deviation of a component output, or by the top event of a standalone fault tree. Leaf nodes in a fault tree can be standard basic events, or they can be references to events in a system model, allowing e.g. a single fault tree to combine output from multiple system architecture models.

---

[21] https://scsc.uk/gsn?page=gsn%20standard
[22] https://dymodiansystems.com/dymodia/

Once models have been created and suitably annotated with failure behaviour, Dymodia can automatically synthesise state-aware failure propagation models (in the form of dynamic fault trees) to describe the dynamic failure behaviour of the system and produce a global view of system dependability.

Both FTA and FMEA are supported. FTA results track changes in state, so that not only combinations but also sequences of events that cause failures can be included. Because all of the models are interlinked as part of an all-in-one platform, the results also maintain links to the relevant model entities (such as the component where a failure mode occurs, or the node in a standalone fault tree).

2.2.2 Behavioural simulation safety analysis approaches

Unlike compositional MBSA approaches, behavioural approaches tend to be inductive in nature and generate analyses by hypothesising a failure and evaluating the effects via simulation. Typically this is done by first creating a formal specification of the system, simulating the nominal behaviour of the system, injecting a fault to determine its effects, and then verifying the system safety properties (i.e. determining whether safety requirements are still met).

Most approaches in the behavioural simulation paradigm introduce their own formal specification language to formally model the system behaviour. This often leads to a greater level of behavioural detail than compositional approaches; for example, most behavioural approaches can differentiate between transient and permanent failure events, as well as imposing more complex timing constraints. The analysis itself is typically carried out by a model checker tool.

Although it is possible to obtain more detailed information via behavioural approaches than compositional approaches, they also share the same general drawbacks: they are more complex and time-consuming (both in terms of modelling and the computational cost for analysis), and the level of detail required typically means that they can only be applied later in the development process, when the in-depth data required is available.

Examples of behavioural simulation approaches include AltaRica, FSAP/NuSMV and its successor xSAP, FPTA, and SAML.

2.2.2.1 AltaRica

AltaRica[23] is a modelling language, methodology, and supporting tools for the formal description and verification of complex systems[33]. In AltaRica, a system model has two dimensions: like the various compositional techniques, AltaRica models a system architecture as a hierarchy of components and subcomponents, but it also models the system behaviour using a combination of states and events.

Component behaviour is defined in AltaRica using *state variables* and *flow variables*. The former represent the internal state while the latter define the external interactions. For example, a switch would have one state variable (on/off) and two flow variables, one for input and one for output. Both types of variables are discrete rather than continuous, meaning that continuous properties such as voltage etc. must be divided up

---

[23] https://altarica.labri.fr/wp/

into intervals (e.g. low voltage, normal voltage, high voltage). A *transition* occurs when a variable changes from one value to another and must be caused by an *event*.

Two types of event are available in AltaRica: *local events* that are internal or local to the component, and *invisible events*, which are external and not directly connected to the component — and thus not visible to it. Keeping the switch example, toggling the switch would be an example of a local event: it changes the state variable and is integral to the switch component itself. On the other hand, a failure of the power supply could be an invisible event: it changes the flow variables to e.g. "low voltage" but the source of the event is beyond the scope of the component. Priority values can also be assigned to events, allowing high-priority events to supersede low-priority events.

All of this information is captured for each basic component in an *interfaced transition system*. This contains descriptions of the various events, possible observations, available configurations (i.e., possible combinations of state and flow variables), mappings from observations to configurations, and the transitions that exist for each configuration.

Basic components can then in turn be composed into *nodes*, which are sets of components that act together under a *controller*. The system itself is the top-level node. Like components, nodes are also defined as interfaced transition systems. Unlike components, nodes also have two additional fields: a description of the node's controller and a set of *broadcast synchronisation vectors*.

As the name implies, broadcast synchronisation vectors are a form of synchronisation mechanism and allow events from one component or node to be synchronised with another. Other information can be included, e.g. indications of whether or not an event is mandatory or constraints upon which combinations of events must or must not occur together. In this way, the mechanisms can be used as Boolean operators or to define e.g. k-out-of-n constraints.

Controllers are responsible for coordinating the configuration and behaviour of their subordinate nodes. For example, a controller may ensure that an actuator is only active when the switch is on and power is flowing. Because they act as parent components for their subcomponents, a transition in a controller may also affect variables in the nodes beneath it.

This knowledge of the behaviour of its subordinate nodes also enables a controller to respond to failures. For example, a controller monitoring a primary component may detect the transition of its operating state variable from "Active" to "Failed" and activate a standby component in response by switching its state variable from "Dormant" to "Active".

An example AltaRica specification for a single-state node is provided below:

```
node block
    flow
        O : bool : out ;
        I, A : bool : in ;
    state
        S : bool ;
    event
```

D4.1 Safety Analysis Concept and Methodology for EDDI development

```
                failure ;
        trans
                S |- failure -> S := false ;
        assert
                O = (I and A and S) ;
        extern initial_state = S = true ;
    edon
```

This defines three flow variables (one output, O, and two inputs, I and A), a single state variable S, and one event (failure). The failure event causes the transition of the state variable S from true to false (i.e., it becomes inoperative). The last two elements define an assertion (output O is true only if there is input at I, the control signal A is active, and the operating state S is true) and the initial value of S (true, i.e., operational).

More complex specifications are possible, e.g. using temporal logic to define transitions or assertions. Once defined, specifications for nodes can be stored in libraries for later reuse.

Once the system model is complete, it can be simulated and analysed in a variety of ways. Analysis involves transformation of the model into some other form first. Simplified fault trees can be generated for analysis of non-dynamic failures, Petri nets or Markov chains might be used to analyse dynamic failures, and transformation into a model-checking language like SMV can be used to verify whether requirements are met.

AltaRica provides a powerful and expressive language for model-based analysis of systems. It grants a lot of flexibility, allowing transformation of the model into a variety of other forms that can then be processed or analysed in different ways. However, as mentioned earlier, the level of detail required can be burdensome and generally restricts its use to later stages of development. Another issue is the problem posed by loops, bi-directional signals, or circular propagations in the model, as might be found in electrical networks or hydraulic systems — this can result in circular logic, i.e., something causing itself, which will be rejected by AltaRica.

2.2.2.2 FSAP/NuSMV-SA

FSAP/NuSMV-SA[24] is a tool consisting of two main parts: the Formal Safety Analysis Platform, which is the graphical user interface, and NuSMV-SA, an extension to the NuSMV model-checking engine with safety analysis capabilities [34]. It incorporates a variety of features, from formal specification, model checking, automatic synthesis and analysis of fault trees, and more.

The FSAP/NuSMV-SA process is generally as follows:

- *System modelling*, in which the system model is formally specified in the NuSMV language. It can be a nominal system model, a dependability model, or a combination of both.

---

[24] https://fsap.fbk.eu/

- *Failure mode identification and model annotation*, in which the failure behaviour of the system components is specified. This can be achieved with fault injection to obtain an extended system model, which describes the failure modes of the constituent components. As with AltaRica, definitions can be saved in libraries and reused in other models — or in the case of failure modes, injected later for the purposes of simulation.

- *Requirements capture*, in which the functional and safety requirements are defined. The system is then later evaluated and verified against these requirements.

- *Model simulation and analysis*, which primarily involves formal verification via model-checking but may also involve e.g. generation of fault trees for FTA purposes.

The last stage is where the properties of the model are validated and verified, both for nominal functional requirements and for safety requirements. Faults can be injected to observe the effects and counter-examples are generated to document cases where properties fail validation or verification. While model-checking is the primary aim, FSAP/NuSMV-SA is also capable of conducting safety analysis directly via a modified form of fault tree analysis. NOT gates can be included (imposing constraints on the occurrence of events, e.g. that a failure only occurs if another event does not occur), as can ordering information to cover cases where the sequence of events is important.

The NuSMV language itself is built on the concept of finite state automata (a form of state machine). Component hierarchies can be defined and their behaviour is specified using a form of temporal logic. NuSMV expressions can then be parsed and checked against specified properties to see whether they hold true.

An example NuSMV description for a two-bit adder is provided below:

```
MODULE bit(input)
VAR
     output : {0,1};
ASSIGN
     output := input;

MODULE adder(bit1, bit2)
VAR
     output : {0,1};
ASSIGN
     output := (bit1 + bit2) mod 2;

MODULE main
VAR
     random1     : {0,1};
     random2     : {0,1};
     bit1        : bit(random1);
     bit2        : bit(random2);
     adder       : adder(bit1.output, bit2.output);
```

First a bit is defined, capable of holding a value of either 0 or 1. The adder is then defined as a function taking two bits. The main component then sets up two random variables, two bits taking values from those variables, and the adder.

We can then inject a fault into the bit module:

```
VAR   output_nominal        : {0,1};
      output_FailureMode     : {no_failure, inverted};
ASSIGN output_nominal := input;
DEFINE output_inverted := !output_nominal;
DEFINE output := case
          output_FailureMode = no_failure : output_nominal
          output_FailureMode = inverted : output_inverted
      esac
ASSIGN next(output_FailureMode) := case
      output_FailureMode   =   no_failure   :   {no_failure,
inverted};
      output_FailureMode = inverted : inverted;
  esac
```

Here we see that in addition to the nominal output (0 or 1), the bit component now has a failure state (no_failure or inverted, i.e. the bit is flipped). The behaviour is then defined (i.e., inverted output is the inverse of nominal output, the no_failure state produces nominal output, and the inverted state produces inverted output), and finally we introduce the possibility of changing state from no_failure to inverted (and once inverted, we cannot change back). In practice, this would be defined via the FSAP interface rather than textually — and once defined, can be stored in a library for later reuse (or injection).

Annotations of failure behaviour like this can be used in multiple ways:

- They can form the basis of a behavioural fault simulation, which results in a trace showing the effects of the fault and the system failures (if any) it causes.

- They can serve as input to a property validation, proving whether or not the property holds (and if not, forming part of the counterexample demonstrating why not).

- They can be used to automatically synthesise fault trees, illustrating the root causes of a given failure event.

- And they can be used as part of a failure order analysis.

Fault trees are generated via a reachability analysis. This begins with the initial state(s) of the system and produces successor states iteratively until a top-level system failure state is reached. The result is the set of all states in which that failure event occurs, with all nominal information removed and leaving only the failure modes. These can then be simplified to produce minimal cut sets. One may note that this is the opposite of how

FTA is normally conducted: it is an inductive process rather than a deductive one. This increases the computational expense considerably, since every single variable needs to be tested in every possible combination to arrive at all possible causes of system failure. Unlike FMEA, however, which is similarly inductive, effects of multiple combinations of failures are considered (and is the cause of the combinatorial explosion).

It should also be noted that the fault trees obtained are "flattened", i.e., they directly link root causes to the system failure top event. The information about the propagation of failure through the system is lost in the process. They also do not include any temporal or dynamic information. Instead, if such effects are to be considered, a failure order analysis is needed. This is based on the same process (and thus the two can be run together) except additional timing and ordering constraints are applied to cut sets. The failure order analysis results in a precedence graph displaying the order of events necessary to cause the system failure event.

One of the major advantages of FSAP/NuSMV-SA is the wide range of capabilities all contained within a single package. The same model is used for multiple purposes: system architecture description, validation & verification of functional properties, verification of safety requirements, model-checking and behavioural simulation, and safety analysis. This contrasts with e.g. AltaRica, in which model-checking and analysis is done separately after a model transformation (to NuSMV itself, for instance). The all-in-one nature of FSAP means that everything is more tightly integrated and the dependability information evolves along with the nominal system design, helping to prevent errors and discrepancies from slipping in.

In addition to the downsides shared by other behavioural approaches, FSAP/NuSMV-SA also suffers from the state-space explosion problem for larger models, where the computational expense of analysis can become prohibitive. Due to the inductive way it generates fault trees, even a purely static fault tree analysis is costly.

2.2.2.3 xSAP

To address some of the problems with FSAP/NuSMV-SA, a successor project emerged through collaboration between the FBK (Fondazione Bruno Kessler) Group and Boeing: xSAP[25]. Like its predecessor, it consists of two parts: the user interface package xSAP and the underlying model-checking engine, nuXMV, itself an extension to NuSMV.

xSAP extends FSAP with a variety of new features, including:

- Library-based specification of faults and fault dynamics;

- Automatic model extension with fault specifications via fault injection;

- FTA and minimal cut set generation for dynamic systems

- FMEA generation

- Mode Transition Cut Set analysis

- Common Cause Analysis

---

[25] https://xsap.fbk.eu/

- Fault propagation analysis based on Timed Failure Propagation Graphs

- Fault detection and isolation

xSAP extends support to infinite state automata (as well as finite state automata) and has expanded capabilities with regard to generation of counter-traces during property validation and verification, including SAT-based bounded model checking. To support safety certification processes required by newer safety standards, probabilistic FTA support has been improved over FSAP and common cause analysis has been introduced. xSAP can also be used to automatically synthesise Timed Failure Propagation Graphs (TFPGs), which model Boolean derive of events, like fault trees, but can also model quantitative timing delays between them [35]. A TFPG consists of nodes representing failure modes or discrepancies (deviations from nominal behaviour) and edges that represent the temporal dependency between nodes.

The xSAP methodology is shown below:
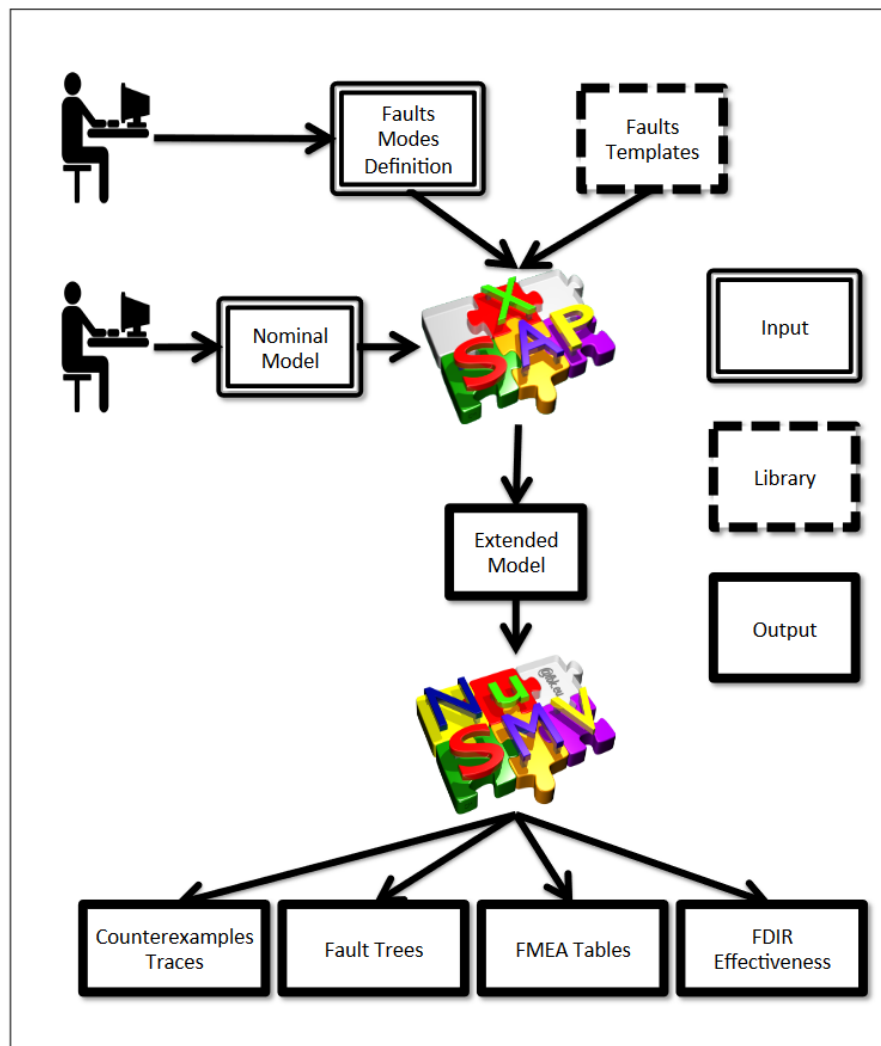
Figure 19 - xSAP methodology (from the xSAP manual)

As with FSAP, the system is formally modelled in two parts: the nominal functional behaviour and the failure behaviour. The latter is part of the extended system model and

can be derived manually or with automatic support via fault injection (once suitable failure modes have been defined).

Once modelled, requirements can be validated and functional properties can be verified. Validation includes property consistency (checking that requirements are not mutually exclusive), property assertion (i.e., that a property is a logical consequence of the requirements), and property possibility (checking that a property is logically compatible with the requirements). Model-checking for functional verification purposes supports deadlock checking (ensuring the system does not cause terminating conditions). xSAP also supports interactive simulation of system execution.

In addition to the variety of safety analysis techniques supported (FTA, FMEA, CCA, TFPGs etc.), xSAP also supports analysis of fault detection/monitoring functionality, including requirements based on inference of faults via indirect observations. It should be noted that this is a design-time analysis, however, not runtime.

Like FSAP before it, xSAP has been integrated as an engine by other tools, such as COMPASS [36] and AUTOGEF[37]. It provides a wide range of features and capabilities presented as part of an all-in-one package. However, as with all behavioural simulation approaches, xSAP requires detailed knowledge of the system to function and tends to be more computationally expensive than compositional approaches.

2.2.2.4 *FPTA*

Fault Propagation & Transformation Analysis (FPTA) is a development of FPTC [38]. FPTA aims to address some of the limitations of FPTC, described earlier, by introducing probabilistic model checking. The PRISM model checker is used to perform the checking [39].

FPTA defines a system (the top-level element) as a set of components linked with connectors. As with most compositional approaches, components possess an interface defined by a set of input/output ports. Similar to SEFTs, however, components in FPTA also possess one or more states known as 'modes', including both normal operating modes and any failure modes. These form Markov chain-esque state machines, and each component therefore needs annotating with a set of possible transitions of the format:

```
input_port.failuremode --> output_port.failuremode, probability
```

This both represents the probabilistic propagation of faults from inputs to outputs but can also be used to encapsulate the generation of failures by giving a probability for a 'normal' input to yield an output failure mode, e.g.:

```
input.normal --> output.normal, 0.9999
```

```
input.normal --> output.omission, 0.0001
```

As with the various compositional approaches, the component-level state machines defined by these expressions can be connected together across the system to provide a system-level view of failure. This is then passed to the PRISM model checker as a form of Markov model and analysed accordingly.

The downside of creating a system-wide probabilistic model in this way is that it is prone to the state-space explosion problem. Unlike GHCFTs or even SEFTs, which attempt to modularise the model and limit the scope of the Markov analysis, in FPTA the entire system must be analysed whole.

2.2.2.5 SAML

SAML — the Safety Analysis Modelling Language — is a tool-independent modelling framework for the creation of system models with both probabilistic and non-deterministic behaviour [40]. It acts as an intermediate layer between graphical modelling tools (like Matlab Simulink or SCADE[26]) and model-checkers or other verification tools (like NuSMV or PRISM). Model transformation is used to switch between the input and output languages of different tools, with the intention of allowing access to multiple tools and thereby taking advantage of the best features of each of them (at the cost of complexity and the necessary transformations).

SAML takes the form of textual annotations to existing nominal system elements. The annotations take the form of a formal representation of finite state automata, not unlike FSAP or xSAP. These automata are executed synchronously in discrete time-steps and can feature both non-deterministic and probabilistic transitions.

```
constant double P_A := 0.1;
constant double P_B1 := 0.2;
constant double P_B2 := 0.3;
constant double P_B3 := 0.5;
formula CASE_3:=V_A=0& !(V_B1=0&V_B2=0|V_B1=1&V_B2=1);


module A
V_A:[0..2] init 0;
    V_A=0&V_B1=0&V_B2=0 ->
     choice (P_A:(V_A'=0)+(1-P_A):(V_A'=1));
    V_A=0&V_B1=1&V_B2=1 -> choice (1:(V_A'=2));
    CASE_3 -> choice (1:(V_A'=1));
    V_A=1 -> choice (1:(V_A'=1));
    V_A=2 -> choice (1:(V_A'=2));
endmodule


module B
V_B1:[0..1] init 0;
V_B2:[0..1] init 0;
true -> choice (P_B1:(V_B1'=0)&(V_B2'=0) +
        P_B2:(V_B1'=1)&(V_B2'=0) +
        P_B3:(V_B1'=1)&(V_B2'=1)) +
     choice (1:(V_B1'=1)&(V_VB2'=1));
endmodule
```

Figure 20 - Example SAML model (from [40])

In the example above, two components (or modules) are defined, A and B. A has one state variable (V_A) while B has two (V_B1, V_B2); V_A can have values 0, 1, 2, while the variables in B are binary. Both modules then define a set of "updates", which are roughly equivalent to transitions in AltaRica and represent change in values of state

---

[26] https://www.ansys.com/products/embedded-software/ansys-scade-suite

variables. The first update in A, for example, indicates a non-deterministic choice between retaining the value 0 (with probability P_A, defined at the top as 0.1) or moving to a value of 1 (with probability $1 - P\_A$). The update in B, by contrast, is solely non-deterministic with no probability.

For specifying properties in a SAML model, a temporal logic is generally used. This is usually CTL for qualitative properties or PCTL (a probabilistic extension of CTL) for quantitative properties. These properties form part of the *extended system model*, which (like FSAP) indicates the additions to the nominal model containing the failure behaviour and other pertinent information, e.g. about the environment.

Verification and analysis of a SAML model requires transformation into a format suitable for the target analysis tool. This can be done as part of the S^3E SAML workbench, based on the Eclipse platform [41]. Alternatively, or in addition, Deductive Cause Consequence Analysis [42] can be used to perform a qualitative analysis analogous to FTA, in which the minimal cut sets are computed. Probabilities can then be calculated subsequently if required.

This transformation and reliance on other tools to perform analysis represents the key disadvantage of the SAML approach. While in principle it offers more flexibility, in practice differences in the semantics between the languages and tools involved can present difficulties. NuSMV, for instance, does not support the parallel, synchronised assignment of values that SAML relies on. Versus AltaRica, the model of time used is different: SAML uses a synchronous yet discrete time model in which multiple automata update simultaneously, whereas AltaRica uses continuous time with discrete events in which only one transition is fired at a time [43].

2.2.3 Allocation of safety requirements

Analysing the causes and effects of failures on a system is only one part of the wider dependability design process. As described above, several behavioural simulation approaches also support validation and verification (V&V) of system properties, including safety requirements (and often functional requirements too). Fault and failure analysis is complementary to these activities, as identifying the possible failures and evaluating their effects is necessary to determine whether the requirements are violated or not.

This duality is captured by modern safety standards such as ISO 26262, where safety requirements are established on the basis of an initial HARA and then subsequently verified using safety analysis.

In ISO 26262, once hazards have been identified, a risk analysis takes place as part of a HARA. Risks are evaluated along three axes: controllability (how easy it is to contain and mitigate the hazard), severity (the degree of potential harm posed by the hazard), and exposure (how frequently or how long the system may be vulnerable to the hazard, or in other words, the likelihood of the hazard occurring). Each of these is scored along a 4-point scale (either 0–3 for controllability and severity, or 1–4 for exposure). For example, severity:

- S0: No injuries

- S1: Light or moderate injuries

- S2: Severe or life-threatening injuries (but survival probable)

- S3: Life-threatening injuries (survival uncertain), fatal injuries

On the basis of these scores, an ASIL — Automotive Safety Integrity Level — is determined for each hazardous event according to a table:

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 2 - ASIL determination according to ISO 26262

ASILs are effectively a qualitative measure of risk. QM means there is no need for any special requirements and ordinary quality management controls are sufficient. A represents a low risk, requiring only minimal measures to ensure safety, while D represents the highest risk, requiring much stricter measures.

With ASILs determined, the next step is the definition of *safety goals* — the top-level safety requirements that define the measures needed to avert or mitigate hazards and reduce risk to acceptable levels. Each safety goal therefore inherits the ASIL assigned to its relevant hazard and this serves as a kind of benchmark of the level of evidence needed to show that the requirement is met. If different ASILs are assigned to similar safety goals, the highest ASIL is the one used to represent the combined safety goal.

From safety goals, several stages of iteration follow in which functional safety requirements are derived from the safety goals and then in turn technical safety requirements — indicating how to fulfil the safety measures in the functional safety requirements — are derived. Along the way, the critical parts of the system are identified, indicating those components that are responsible for meeting the various requirements. Those components then inherit ASIL values indicating how critical they are to the overall safety of the vehicle and thus how stringent their development needs to be. If multiple independent components are collectively subject to a requirement, the responsibility of meeting that requirement can be shared amongst them according to a

simple algebra in which A = 1, B = 2, C = 3, D = 4, and an ASIL requirement can be met by summing the ASILs of its components (e.g. A (1) + B (2) = C (3)).

This latter process is known as ASIL decomposition. It is particularly important because a component's ASIL can significantly impact its development and production cost, and costs tend to rise exponentially with higher ASIL ratings. Thus it may be more cost effective — and potentially more reliable — to e.g. use two ASIL B components rather than a single ASIL D component. The figure below illustrates this idea:
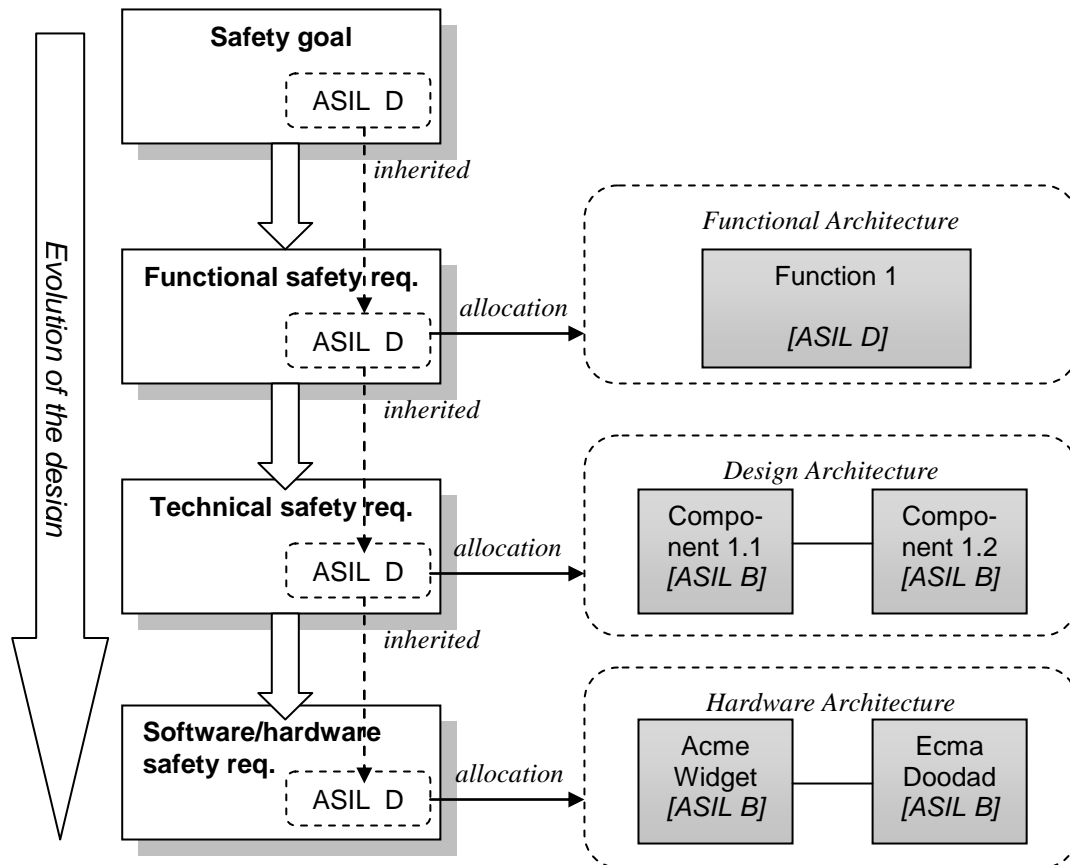
Figure 21 - Inheritance and allocation of safety requirements throughout development

The concept may be straightforward enough — ASILs are allocated based on a component's contribution to the overall safety requirements. But how is that determined exactly? In order to allocate the right ASIL, we first need to understand how critical a component is, how it responds to failure, which hazards it contributes to (and thus which safety requirements it is subject to), and which other components it shares responsibility with. All of this requires detailed knowledge of how the components are connected and how failures propagate between them. And while the discussion above has focused on ISO 26262, similar processes exist in other related standards, like ARP4754-A.

This is where MBSA techniques and compositional safety analysis approaches in particular come in useful, because most of them function by analysing a system to determine failure propagation and in doing so linking component-level failures to the system-level hazards they cause. If a safety analysis shows that a given component failure can lead directly to a hazard, then it must be allocated the full integrity level

necessary to meet the safety requirement associated with that hazard. If, on the other hand, the analysis shows that a component can never cause that hazard, or does so only in conjunction with other failure events, then a lower integrity level may be acceptable [44].

As an example, consider the following system:
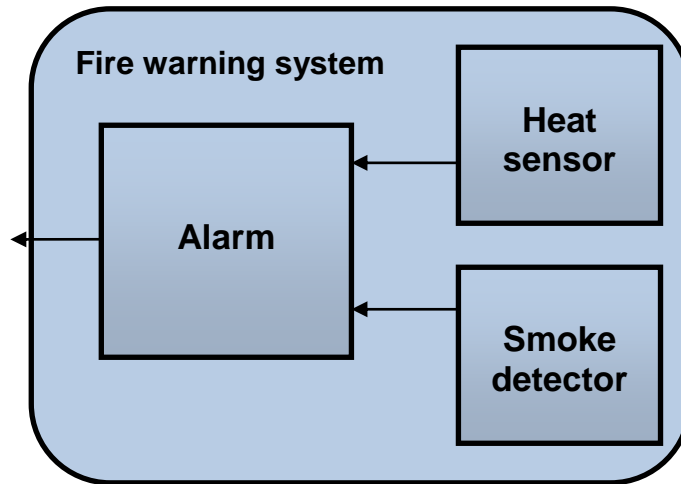
Figure 22 - Example system for ASIL allocation

Assume an ASIL of C has been assigned to the fire warning subsystem, on the basis that its failure leads to the severe hazard "No warning of fire". During development, the design of this subsystem led to three subcomponents: the alarm and two different sensors, either of which alone can trigger the alarm if fire or smoke is detected. We could simply allocate ASIL C to all three subcomponents, but instead a preliminary qualitative safety analysis has produced a fault tree showing that both sensors must fail to cause the hazard (see below).
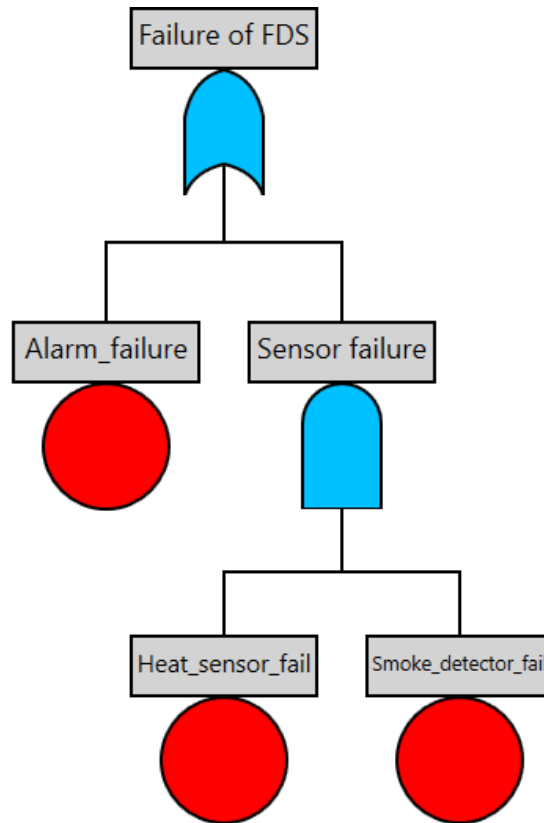
Figure 23 - Fault tree for ASIL allocation

In this case, we know we must assign ASIL C to the alarm, since it is a single point of failure for the subsystem — failure of the alarm directly causes the hazard. However, the two sensors together share responsibility for detecting the fire and thus must *jointly* meet ASIL C. This presents us with four options for decomposition:

Heat sensor	Smoke detector
QM	C
A	B
B	A
C	QM

Other combinations are of course possible, like allocating ASIL C to both sensors, but this would not be cost effective. This can be seen more readily if we assign cost values to each ASIL, e.g.:

- QM = 0

- A = 1

- B = 10

- C = 100

- D = 1000

We can then determine the total cost of each possible allocation in the table:

Heat sensor	Smoke detector	Cost
QM	C	0 + 100 = 100
A	B	1 + 10 = 11
B	A	10 + 1 = 11
C	QM	100 + 0 = 100

As can be seen, the most cost-effective allocation here would be to assign ASIL A to one sensor and ASIL B to the other.

More complex scenarios exist where requirements overlap. Consider a situation where the heat sensor somehow contributes to another hazard (and thus is subject to another safety requirement) but the smoke detector is not. In such a case, the second requirement may impose a minimum ASIL requirement on the detector — e.g. ASIL C — which rules out all but the last row in the table above. In other cases, the decomposition might produce too many options, making it difficult to manually choose an allocation or even for a tool to deterministically derive an optimal solution; in such situations, an automatic optimisation process can be used to rapidly determine optimal solutions. The HiP-HOPS approach is capable of such a process and has been applied to both ASILs for ISO 26262 [27] and DALs for ARP4754-A [28].

Beyond supporting allocation of decomposed SILs, safety analysis is also vital for demonstrating that the requirements have been met to the desired standard. Qualitative analyses indicate failure cause and effect, potentially highlighting places where e.g. components contribute to hazards but they have not been subjected to the necessary requirements due to an oversight. And quantitative analysis produces probability estimates that can be used to show whether requirements have been met according to guidelines that govern the likelihood of hazards (since reducing the likelihood is one way to reduce risk). For instance, a high integrity level might mandate a maximum probability of no higher than 1e-6, while a low level might only impose a threshold of 1e-4. Analysis can then demonstrate whether these thresholds are met.

2.2.4 Safety argumentation

In addition to failure analysis and support for the allocation of safety requirements, another aspect of the dependability-driven development process that can benefit from MBSA is in the generation of safety argumentation. Standards do not just impose a methodology to follow: they typically also require the documentation of evidence of any actions taken to argue that a system is safe. Such documentation is often referred to as a **safety case**.

Although there is no single formal definition of what a safety case should contain or how it should be specified, the definition in [45] is a good indicator: "A safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context." Safety cases have been adopted by various standards across various industries, including ARP4754-A. Each has their own

slightly different definition of what a safety case is; for example, the UK Ministry of Defence defines a safety case as:

> "*...* a comprehensive and structured set of safety documentation which is aimed to ensure that the safety of a specific vessel or equipment can be demonstrated by reference to safety arrangements and organisation, safety analyses, compliance with the standards and best practice, acceptance tests, audits, inspections, feedback and provision made for safe use including emergency arrangements." [46]

Generally speaking, common aspects covered by a safety case may include:

- Definition and/or description of the system, including its scope.

- The hazards identified.

- The safety requirements.

- A risk assessment.

- Risk reduction measures and safety mechanisms etc. applied.

- Safety analysis results to provide evidence of the efficacy of such measures and to show that the requirements have been met.

- Information about the development process undertaken.

However, the primary aim of a safety case is to argue that a system is sufficiently safe, and in that respect the most important three elements are the safety *requirements*, *arguments* as to how the requirements have been addressed, and *evidence* to prove the claims (e.g. in the form of safety analysis results).

2.2.4.1 Goal Structuring Notation

Due to the informality in the definitions and the ensuing lack of clarity involved in producing safety cases, a variety of different safety argumentation notations have been proposed to try to formalise and structure the process more clearly. Perhaps the most prominent is the Goal Structuring Notation (GSN), developed at the University of York in the 1990s and now maintained by the Assurance Case Working Group (ACWG) [47].
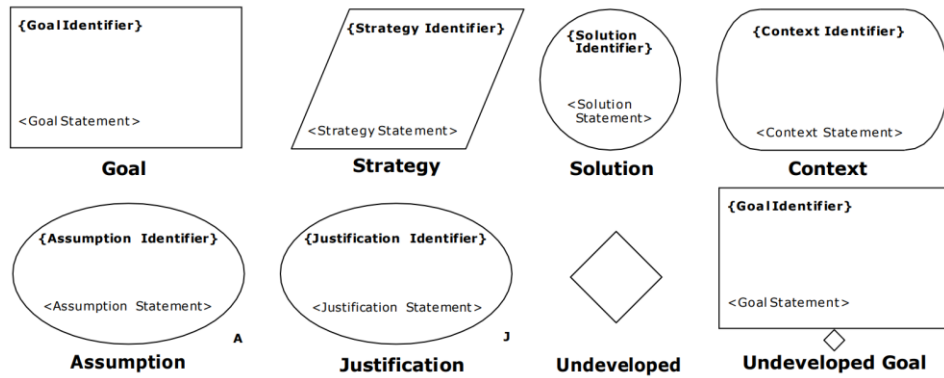
Figure 24 - GSN Elements (from [48])

The core elements of GSN include:

- *Goals* represent safety claims made;

- *Strategies* represent the inference between a goal and any supporting goals;

- *Solutions* provide references to evidence to support a claim;

- A *Context* can be a statement or a reference to some contextual information;

- An *Assumption* is a hypothesis made as part of the argument;

- A *Justification* is a rationale behind the choice of a strategy;

- And *undeveloped* elements act as placeholders for future argumentation.

Together, these elements form an interconnected argument structure. For example:
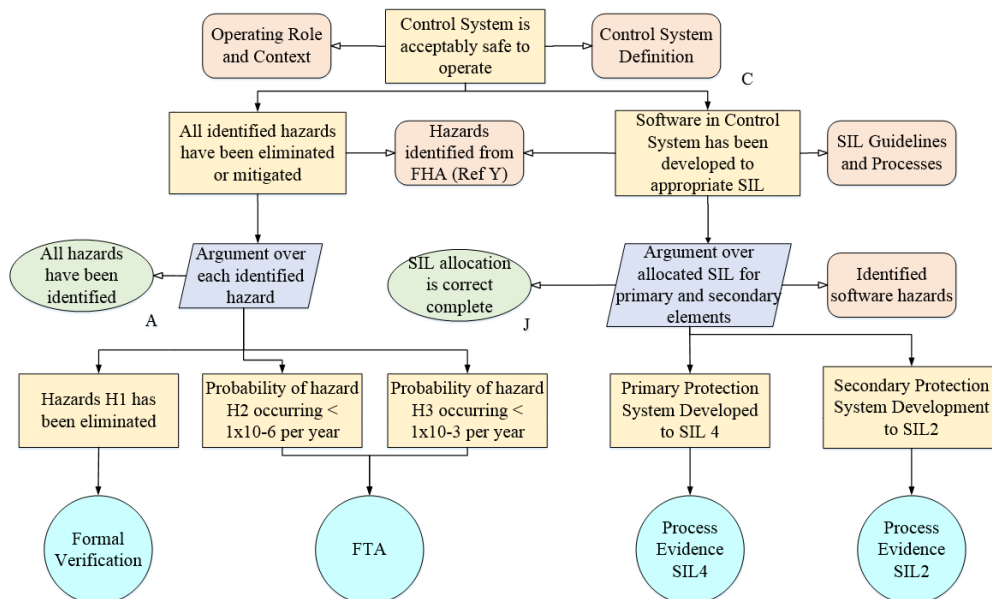
Figure 25 - Example GSN argument structure (from [49])

Here, the top-level argues that the "control system is acceptably safe to operate". Two context elements are linked, and two supporting goals are defined: one stating that all hazards have been eliminated or mitigated and the other stating that the control software has been developed to the appropriate SIL. Further context elements provide support for these, including a reference to hazards from a hazard analysis.

Each of these subgoals is elaborated further via two strategies. The first deals with each hazard, claiming that some hazards have been eliminated and that the probability of others has been reduced to a given level; the former is argued on the basis of a formal verification solution, while the latter is determined by an FTA. The second strategy, referring to the SIL, likewise has solutions providing evidence to support the claims — in this case, documentation from the development process.

The structured nature and graphical layout of GSN are a significant improvement over informal, ad-hoc safety cases, particularly in terms of making the logic of the safety argument clear. Over time, GSN has matured and been standardised, allowing production of supporting materials to flourish.

However, just like early safety analysis approaches, one of the drawbacks of GSN is that it is separate to the development models used during the design process — and also to the safety models and analysis artefacts.

2.2.4.2 CAE – Claims, Arguments, Evidence framework

GSN is not the only notation used for safety argumentation. CAE — the Claims, Arguments, Evidence Framework — is another. Developed by Adelard LLP, it was originally intended for reasoning about the safety and trustworthiness of systems, as in a safety or assurance case[27]. Over time, its uses have broadened to reasoning more generally about complex systems and across the system lifecycle.

As the name implies, there are three key elements to CAE:

- *Claims*, which are true/false statements about a property of an item (e.g. "the train is safe");

- *Arguments*, which are rules that link evidence or assumptions with the claim being investigated;

- *Evidence*, which consists of artefacts that establish trusted facts and thus support a claim.

A variety of building blocks are used to help assemble assurance cases, acting like templates that address particular types of problems. For example, decomposition breaks up a claim into multiple sub-claims: if it can be established that a property holds for all subcomponents of an object, then that property holds for the object itself too (for a given context, at least). Substitution indicates that if a property holds for one item, then it also holds for an equivalent item. Others include 'concretion' (making a claim more precise), 'calculation' (computing a property from other related properties), and evidence incorporation.

---

[27] https://claimsargumentsevidence.org/

These building blocks can be illustrated by the 'helping hand':
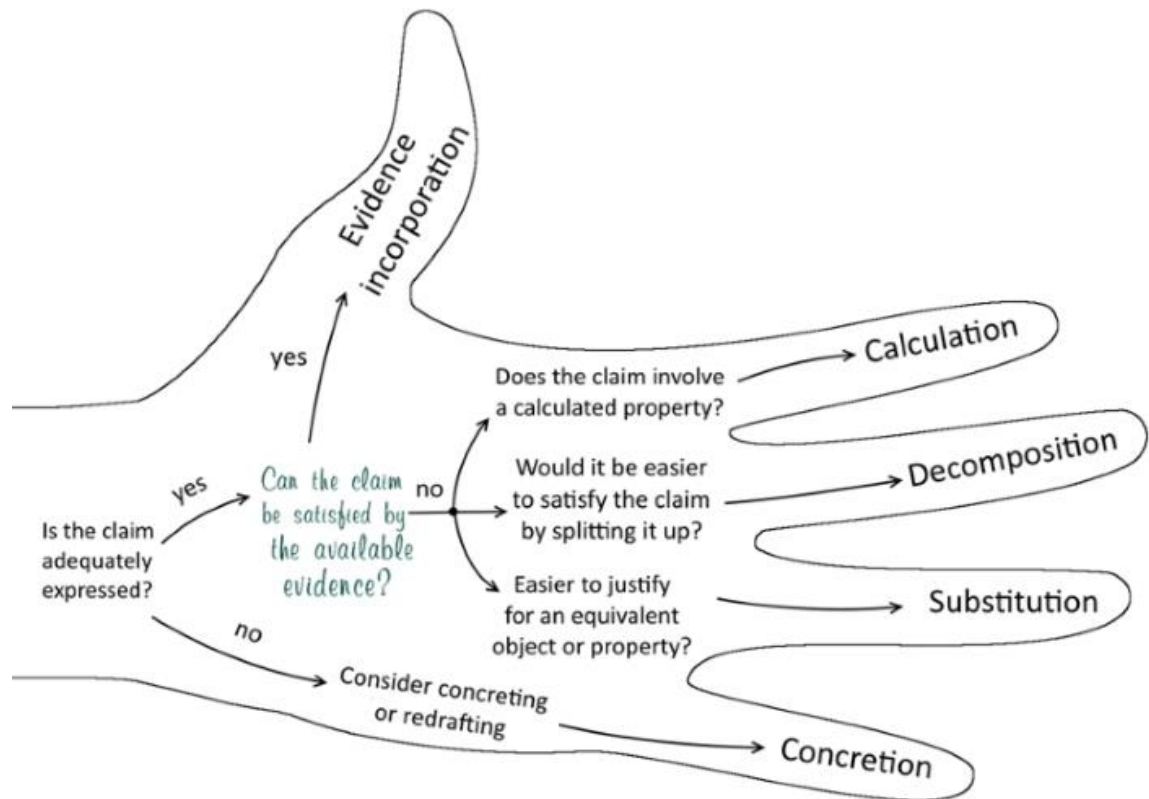
Figure 26 - The CAE 'Helping Hand'[28]

Although there are plenty of cosmetic differences, CAE and GSN are generally comparable in what they express and have both been successfully used to create safety cases. Like GSN, however, the information recorded by the CAE model is largely separate to those of the development models used during the design process (though these models could be included as evidence artefacts).

2.2.4.3 SACM

The Structured Assurance Case Metamodel (SACM)[29] is one attempt to address this shortcoming. Developed by the Object Management Group, it is a metamodel intended for the representation of structured arguments such as safety cases/assurance cases. An assurance case is defined in SACM as a set of claims, arguments, and supporting evidence to justify a claim that a system satisfies some set of requirements.

The goal behind SACM is to provide a model-based approach to system assurance and supports existing graphical notations like GSN and the Claims, Argument, Evidence approach. It is a package consisting of five main elements:

---

[28] https://claimsargumentsevidence.org/wp-content/uploads/2018/01/helping_hand.jpg
[29] https://www.omg.org/spec/SACM/2.2/About-SACM/
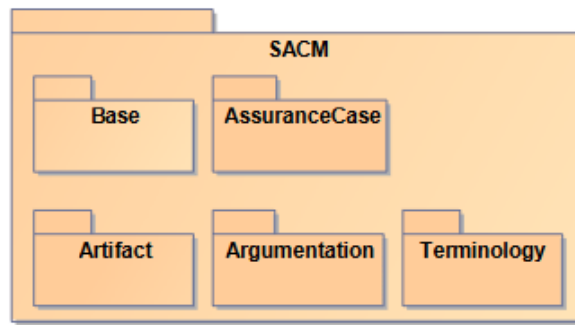
Figure 27 - SACM elements (from [48])

The AssuranceCase element, as the name would suggest, encapsulates all the concepts necessary to produce an assurance case, like a top-level container. This is further supported by the Argumentation component (for making claims and arguments), the Artifact component (representing concepts used in defining evidence for the arguments), and the Terminology component (which helps to define the vocabulary used to express system properties and characteristics). The Base component, meanwhile, encapsulates all basic entities such as multi-language strings.

Compound assurance cases can be constructed via bindings to link subordinate assurance cases together (along with structured information that justifies why such a link between the systems being represented is possible in terms of compatibility and trustworthiness).

SACM also provides an "Interface" concept for the purposes of model exchange etc. By means of an AssuranceCasePackageInterface, some part of the assurance case can be exposed, e.g. to make it queryable or exchangeable at runtime. The Base component can also define the necessary formats to support machine readability as well as human readability.
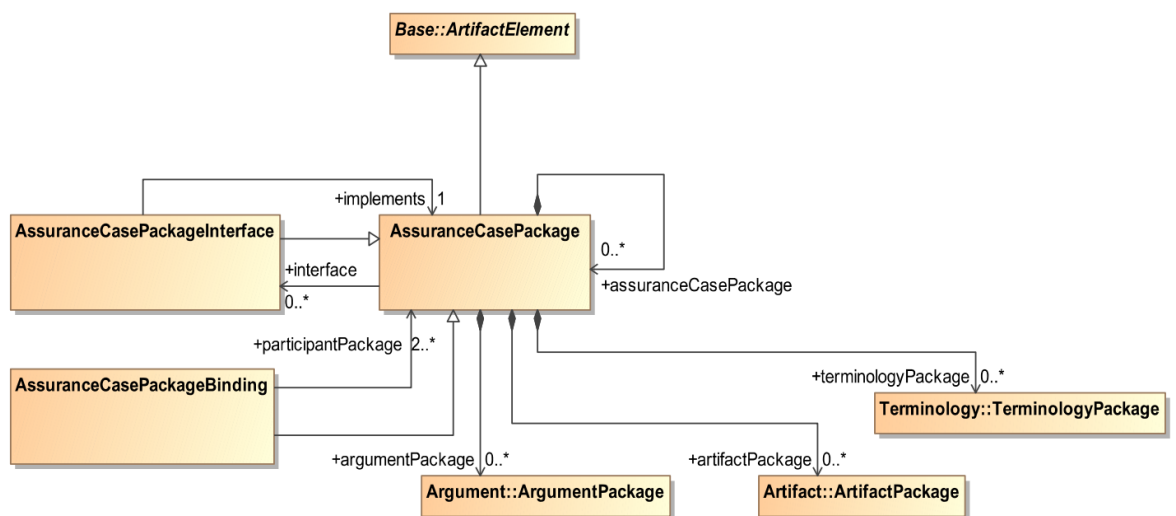
Figure 28 - Overall SACM metamodel[30]

---

[30] https://www.omg.org/spec/SACM/2.2/PDF

These features for compositionality and modularity make it easier to divide up an assurance case into smaller, more manageable (and understandable) components, particularly when dealing with different but related concerns, such as security and safety. It also makes it more feasible for assurance cases to be used at runtime, e.g. in multi-agent or cooperative systems of systems, where different agents must negotiate their demanded safety requirements and provide safety assurances to attempt to guarantee collective safe operation.

Newer versions of SACM also define its own graphical notation (in addition to supporting multiple existing approaches, like GSN and CAE) and there are tools that support this, as illustrated by the ACME tool (Assurance Case Modelling Environment) [50].

2.2.4.4 *Automatic generation of Safety Cases*

With the aid of model-based dependability approaches, it is possible to both combine safety cases with dependability models (as in the case of safeTbox, for example) and in some cases *generate* safety cases semi-automatically from dependability models.

In the context of model-based safety analysis, the advantages offered by such a capability are obvious. Just as it can be a challenge to keep a safety analysis current with an evolving design, so can it be a challenge to keep the safety argumentation up to date (an even greater challenge, arguably, since the latter depends on the safety analysis too). If all of these could be linked together as part of a cohesive, over-arching model framework, then they can all move in lockstep and be updated together. As the design changes, the safety analyses can be re-run on the same model to ensure the safety requirements still hold (or if not, why not). This means any safety implications introduced by the design change can be identified immediately and any conclusions reached can thus be fed into the next design iteration. If the safety argumentation can then also be generated, this means that the safety argumentation for the model is automatically updated so that it always reflects the current state of the design without the need for extensive (and potentially costly) manual revisions.

Given the potential benefits it offers, there has been a range of work in this area in recent years. The FLAR2SAF (Failure Logic Analysis Results to Safety Case Argument Fragments) approach[51] is one such work, which makes use of FPTC and the CHESS toolset[52]. FPTC expresses how a component can propagate, transform, or even absorb a failure received at its input and transmitted from its output, and from this a type of safety contract can be defined. Such contracts indicate safety properties that a component either requires or guarantees during its operation, and may be either 'weak' (context dependent) or 'strong' (context independent). The output of the FPTC analysis itself can then be used as evidence in support of the safety contract, forming a kind of component-level safety case (or fragment).

Another approach is the THRUST methodology [53]. THRUST uses the concept of a process line, which are sequences of tasks that can potentially vary or branch out. Tasks are assigned to specific people (or rather, people serving specific roles) and specific tools that are used to create products and complete those tasks. An argumentation line uses this same concept to create a safety argument, supporting a given process line. Although different languages are used (SPEM or vSPEM for the process line, S-TunExSPEM for the argumentation line), both are modelled in parallel and the

process line model can also be transformed to help serve as the basis for the argumentation line. This helps ensure that both are kept in sync while also enabling information in the process line model to be reused directly as part of the argumentation line, although there is no formal metamodel encompassing both lines.

Other works have adopted a verification-oriented approach using formal verification. The technique in [54] uses code along with formal specification, the safety requirements, and FTA to identify flaws in either the specification or the code by means of path and coverage analyses and verifying whether the formal specification holds. In [55], formal software verification is used to produce arguments from argument patterns. Safety properties are formally specified, defining the goals of the safety case, and then analysis evidence is collected to support these goals and verify the requirement over the course of multiple phases, beginning with high-level hazards and moving on to low-level failures. Here the claims of the safety cases are manually created but then backed with evidence generated automatically by the verification process.

Finally, there have also been several approaches built on the HiP-HOPS tool. [56] and [57] present an approach for automatically generating safety arguments for software product lines by combining variability management, MBSA (via HiP-HOPS), and assurance cases (via SACM). An integrated metamodel encompassing functional, architectural, and failure models is used to model the software product lines; HiP-HOPS's capability for automatically allocating SILs is used to guide the argumentation (in the form of SACM), and evidence is provided by safety analysis techniques such as the AADL error annex and HiP-HOPS. A more generic approach is presented in [58], in which argument patterns are used to algorithmically control the generation of safety argumentation when combined with the ability to automatically allocate DALs. This process is illustrated by the diagram below:
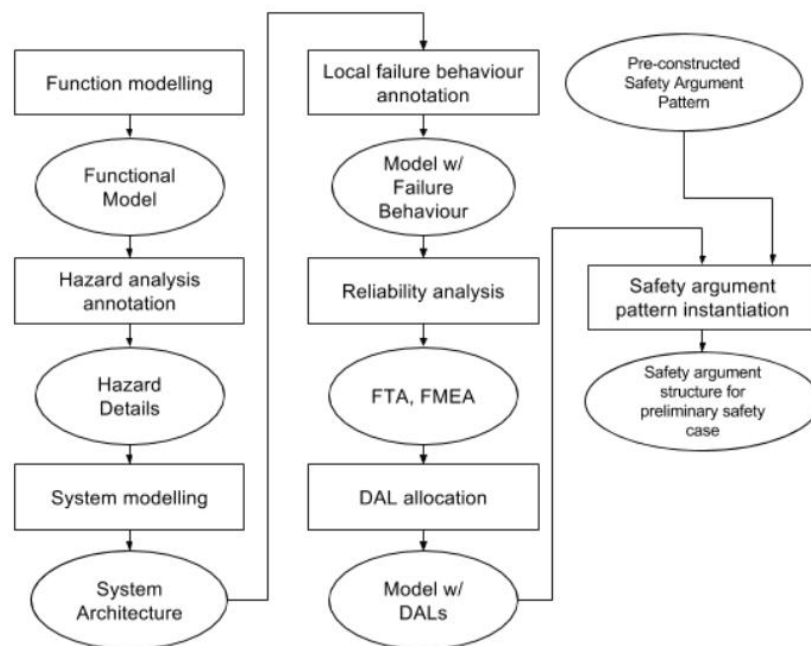
Figure 29 - Process of automatically generating safety arguments (from [58])

Matlab Simulink and HiP-HOPS are used to illustrate the process. Functional and system modelling is performed in the former, while hazard analysis and local failure behaviour annotation is performed via the HiP-HOPS plugin. Reliability analysis (in the form of FTA and FMEA) is performed on this model via HiP-HOPS. On this basis, DALs (or other forms of integrity levels) can be allocated across the subcomponents, representing how the safety requirements are distributed over the system. Finally, the model, analysis results, and decomposed requirements are used in conjunction with a given argument pattern to instantiate the (preliminary) safety case argumentation structures.

Regardless of the exact technique employed, the ability to automatically generate safety cases (or parts of them) from joint nominal/dependability models offers significant advantages in ensuring that safety argumentation is comprehensive, up-to-date, and accurate with respect to the underlying design models. Furthermore, by significantly reducing the effort involved, it means that the safety argumentation can be generated much earlier and kept in sync with the evolving design during development, rather than only being created at the end.

2.2.5   Digital Dependability Identities: a comprehensive approach to model-based safety

Digital Dependability Identities, or DDIs, are self-contained, analysable, and composable models that combine all the information necessary to uniquely describe the dependability characteristics of a component or system [59]. Originally developed as part of the H2020 DEIS project[31], the motivation behind the DDI is to encapsulate model-based descriptions of all required dependability artefacts — hazard and risk analyses, fault trees, FMEAs, Markov models, system architectures, safety argumentation etc. — in a standardised, machine-readable form. A DDI can then be composed hierarchically or otherwise connected to other DDIs to form a description of an entire system (or even system of systems).
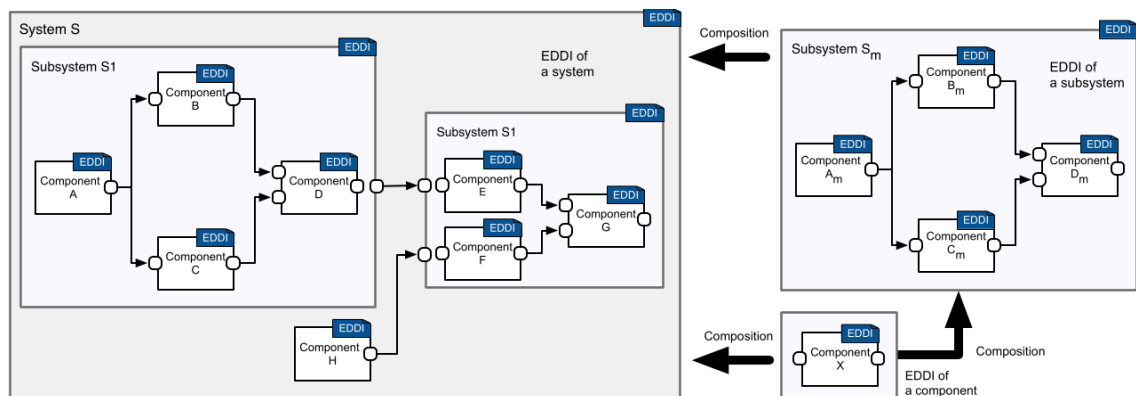
Figure 30 - Composition of DDIs

Because they capture all the necessary dependability properties into a single standardised entity, DDIs can streamline the exchange and evaluation of information, both within a developer organisation and outside the organisation (by providing a limited interface that exposes pertinent data while hiding implementation detail). They also serve as living dependability assurance cases, including both the dependability requirements for the component, arguments for how they are met, and evidence (in the

---

[31] https://www.deis-project.eu/

form of safety analysis artefacts) to support the arguments. An example of this can be seen below.
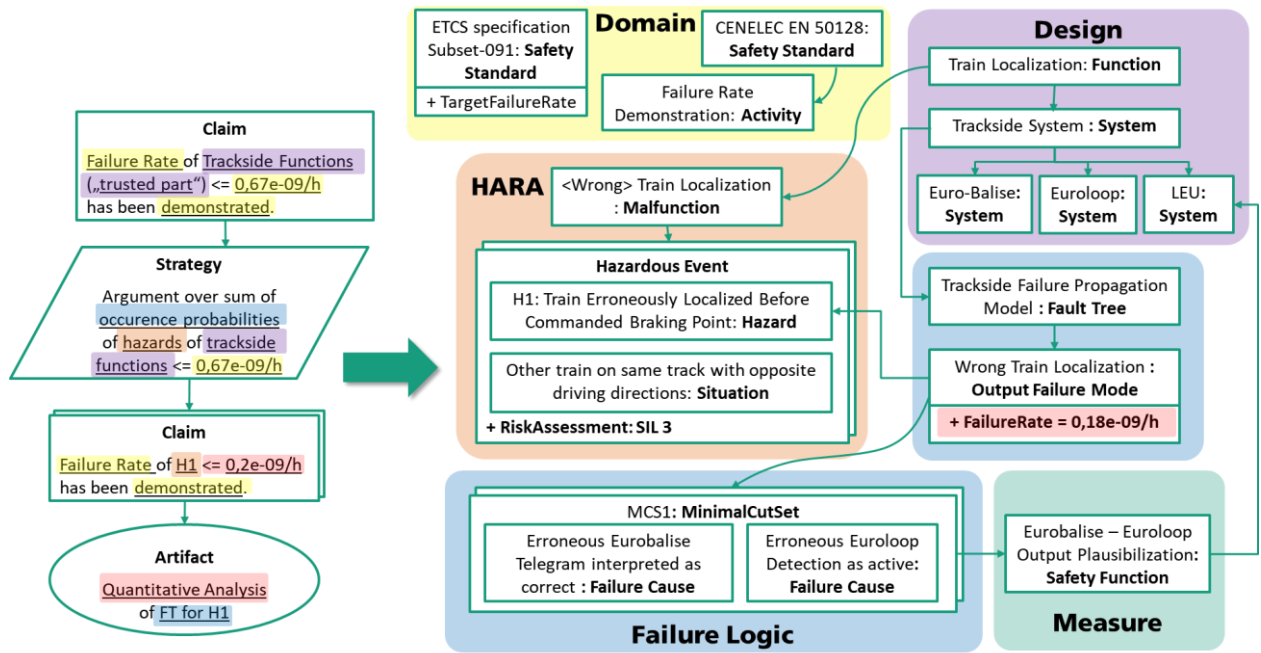
Figure 31 - Illustrative DDI for a dependability assurance case

It is important to note that DDIs are intended to be evolving artefacts and may assume different forms and contain varying content depending on the intended recipient and the current stage of the system lifecycle. Early DDIs may be more abstract, for instance, becoming more detailed as the design evolves. Alternatively, DDIs could be divided according to exposure: white box DDIs would be used for internal development, containing (and exposing) all information; grey box DDIs could then be provided to integrators/suppliers and would contain the same range of information but limit some of the implementation details for the purposes of IP protection; and black box DDIs that contain only the dependability guarantees could be employed in the field.

Experimental work has also been undertaken to apply DDIs at runtime as part of a cooperative vehicle platooning system using ConSerts (c.f. 4.2.2) [60]. Here, the information about the vehicles stored in the DDIs is combined with ConSerts' ability to exchange conditional safety guarantees and a runtime monitoring architecture to show how systems can react dynamically to changing environmental conditions to maintain safe operation. This paves the way towards the EDDI, or Executable DDI, as discussed in Section 4.

Prototype tool support for DDIs already exists within the safeTbox and HiP-HOPS tools, achieved via model transformation using the Apache Thrift[32] framework.

2.2.5.1 The Open Dependability Exchange metamodel

Specification and standardisation of DDIs is achieved via the Open Dependability Exchange (ODE) metamodel[33]. The ODE comprises two main parts: the assurance case

---

[32] https://thrift.apache.org/
[33] https://github.com/Digital-Dependability-Identities/ODE

metamodel (a version of SACM, which has already been discussed in Section 2.2.4.3) and the product metamodel (which models the system dependability information). In effect, the ODE is therefore a superset of SACM with added features for system and failure modelling.

Like SACM, the ODE's product metamodel consists of several main packages. The first is the Base package, which covers fundamental entities like IDs and time units as well as an extensible key-value map structure that can be employed to create custom properties to store information not otherwise catered for by the rest of the metamodel.
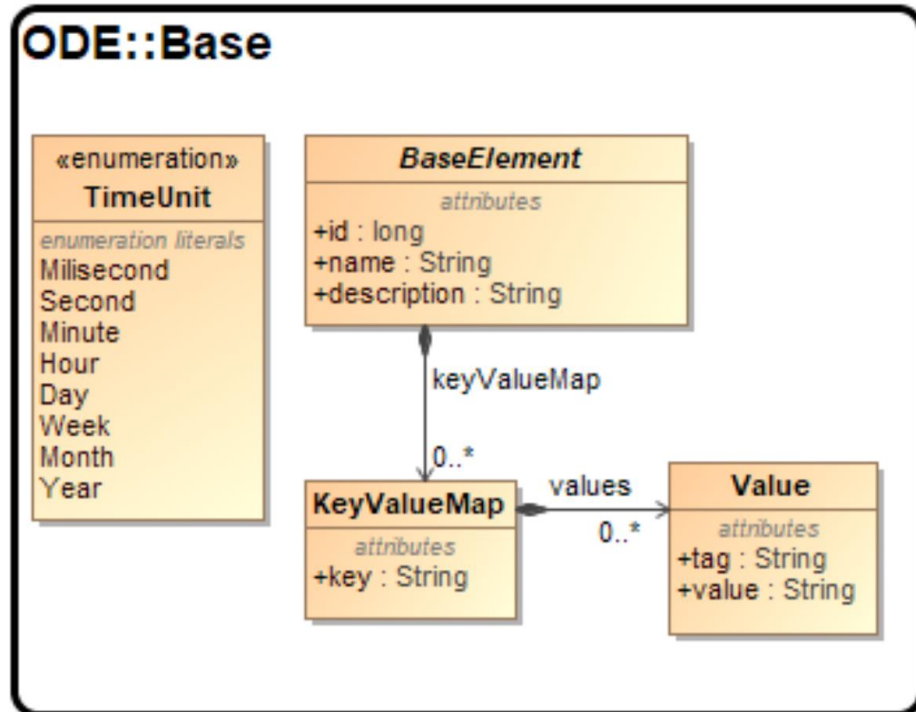
Figure 32 - ODE::Base package

The Design package is intended for system architecture modelling. It defines generic entities to describe systems, functions, components, ports to act as interfaces, and connections between them:

Confidentiality: Public Distribution
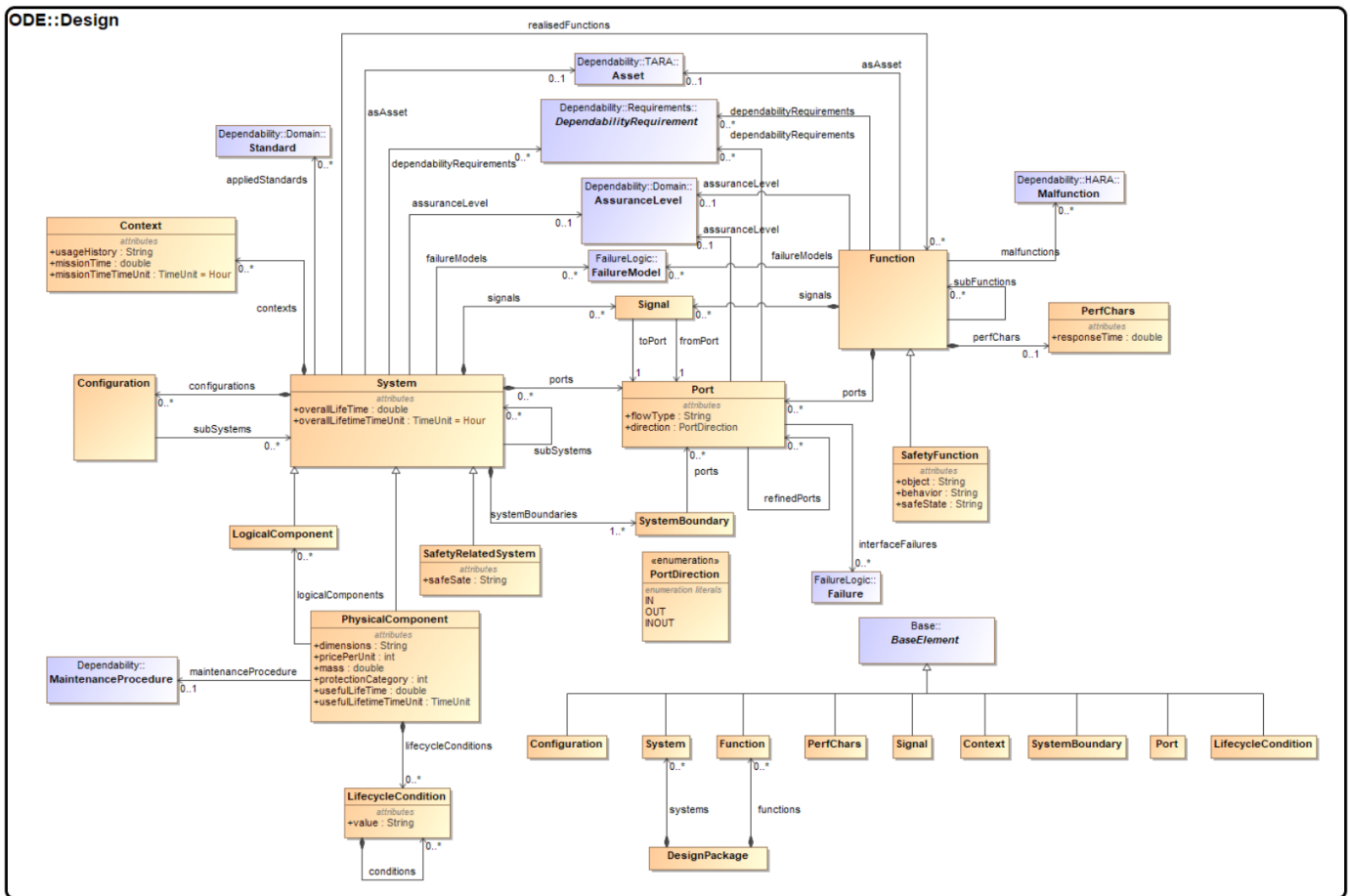
Figure 33 - ODE::Design package

Key elements include:

- *System*, representing a generic system and serving as the top-level entity for most of the other parts of the package. A System can have Ports and Signals (representing its external interface and communication with the environment/other systems), as well as subsystems (to create a hierarchy). A System also maintains references to the failure models and possible dependability standards associated with the system as well as the functions it realises and requirements it fulfils.

- *Context*, which describes the operational context of the System (particularly its expected mission time).

- *LogicalComponent* and *PhysicalComponent*s; LogicalComponents can represent either conceptual or software components, while PhysicalComponents represent implementation in hardware. Both are Systems in themselves, allowing the system hierarchy to be constructed. PhysicalComponents may also contain information on e.g. maintenance procedures and component lifetimes/lifecycle stages.

- *Ports* define the interface of a System or Component. They can have a variety of types (data, energy, fluid etc.) as well as a direction (input / output / both).

Confidentiality: Public Distribution

Assurance levels (e.g. SILs) can be assigned to Ports as can dependability requirements and potential failures at that port.

- *Function* represents a conceptual or logical operation, thus supporting an earlier, more abstract functional architecture before a more detailed implementation becomes available. They too can contain sub-functions, Ports, requirements, failure models, and most of the other paraphernalia of a System.

- *Signals* model links between two ports and typically represent some form of communication or transfer, though the actual role depends on the context. For example, a Signal between two Components could mean a service request / provision pair, but between a Component and a Function it could represent an implementation (i.e., the component implements the function).
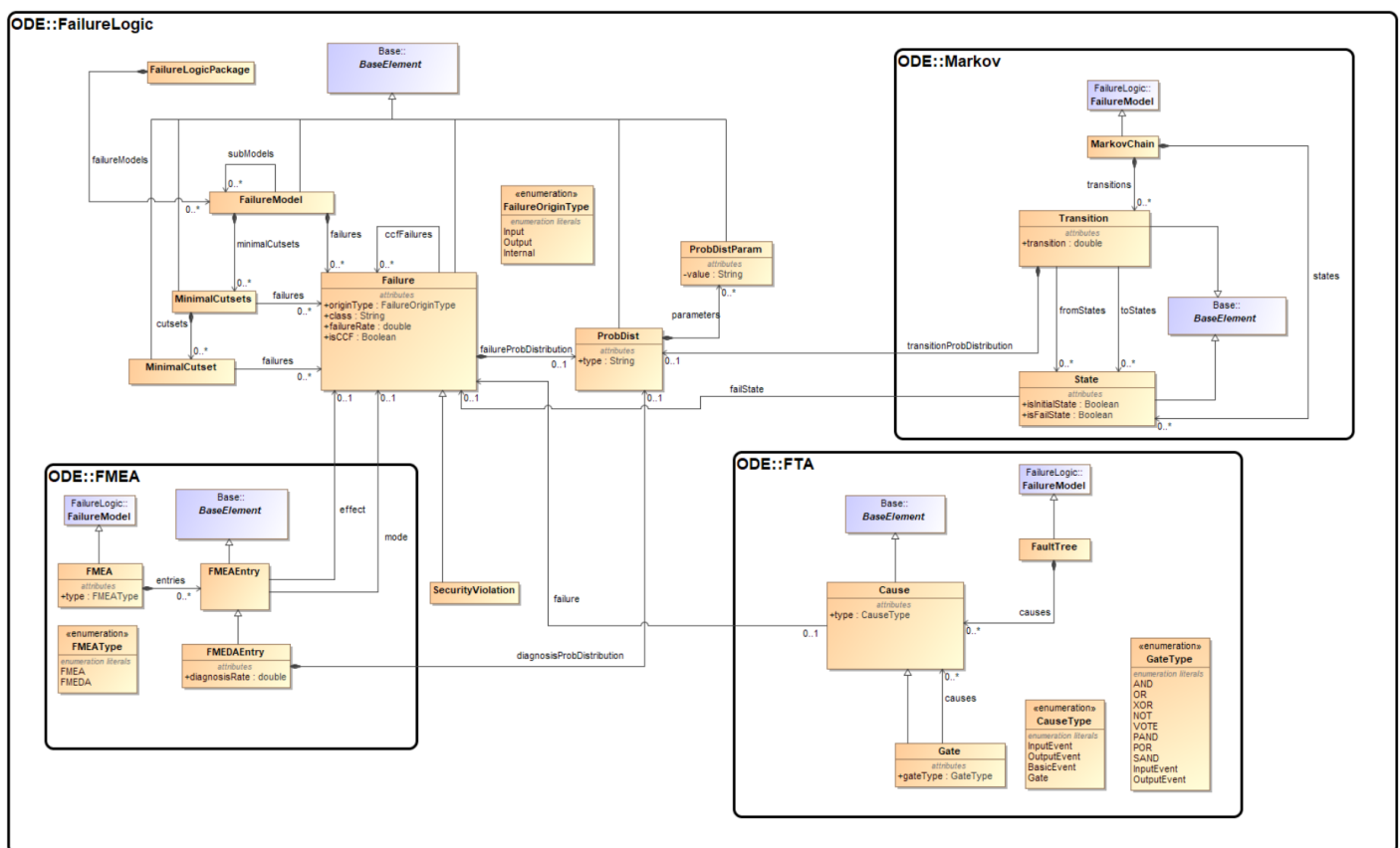
Figure 34 - ODE::FailureLogic package

The FailureLogic package supports a variety of failure logic analysis approaches. FTA, FME(D)A, and Markov analysis are directly supported with their own sub-packages, but the entities could also be used to describe similar methods too (e.g. the Markov package could potentially be used to represent other forms of state machine or other probabilistic state-based analyses).

Some elements are more generic and common to all three types. These include:

- *Failure* is the key element of the package and represents a generic failure of a function, system, or component. Failures can be either 'input' (i.e., propagated

from an external element and received at an input Port), 'output' (propagated from an output Port, e.g. an omission of intended output), or 'internal' (i.e., an internal failure mode of the component/function). Failures also have a class (e.g. omission, commission, value, timing, etc.) so that input and output failures can match, as well as a probability distribution (e.g. failure rate). Common cause failures can also be represented using the isCCF flag and a reference to the common cause failure being represented locally by the given Failure.

- *FailureModel* represents a type of analysis model (e.g. fault tree, FMEA). It contains all the failures referenced by that model as well as the results in the form of MinimalCutSets.

- *MinimalCutSets* represent the causes of a specific Failure or set of Failures (and are typically the outcome of an FTA).

The analysis-specific packages then include further elements to support those analyses, e.g. fault tree gates for FTA, States and Transitions for Markov etc.
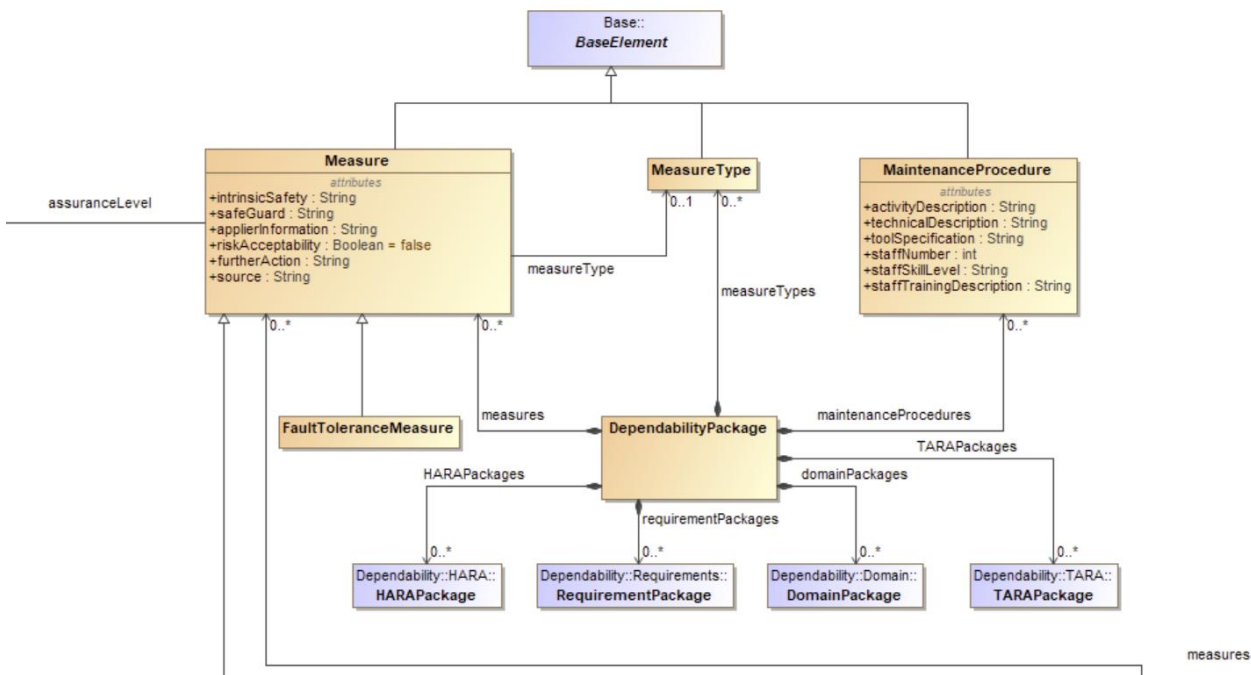
Figure 35 - ODE::Dependability package

The Dependability package is the overall package related to dependability requirements, standards, and risk analyses. The key element of the base Dependability package is the *Measure*, which represents a planned or applied measure to support dependability and/or ensure safety.

More extensive capabilities are provided by the various sub-packages. The Domain sub-package, for example, contains elements to describe relevant standards and definitions of assurance levels.
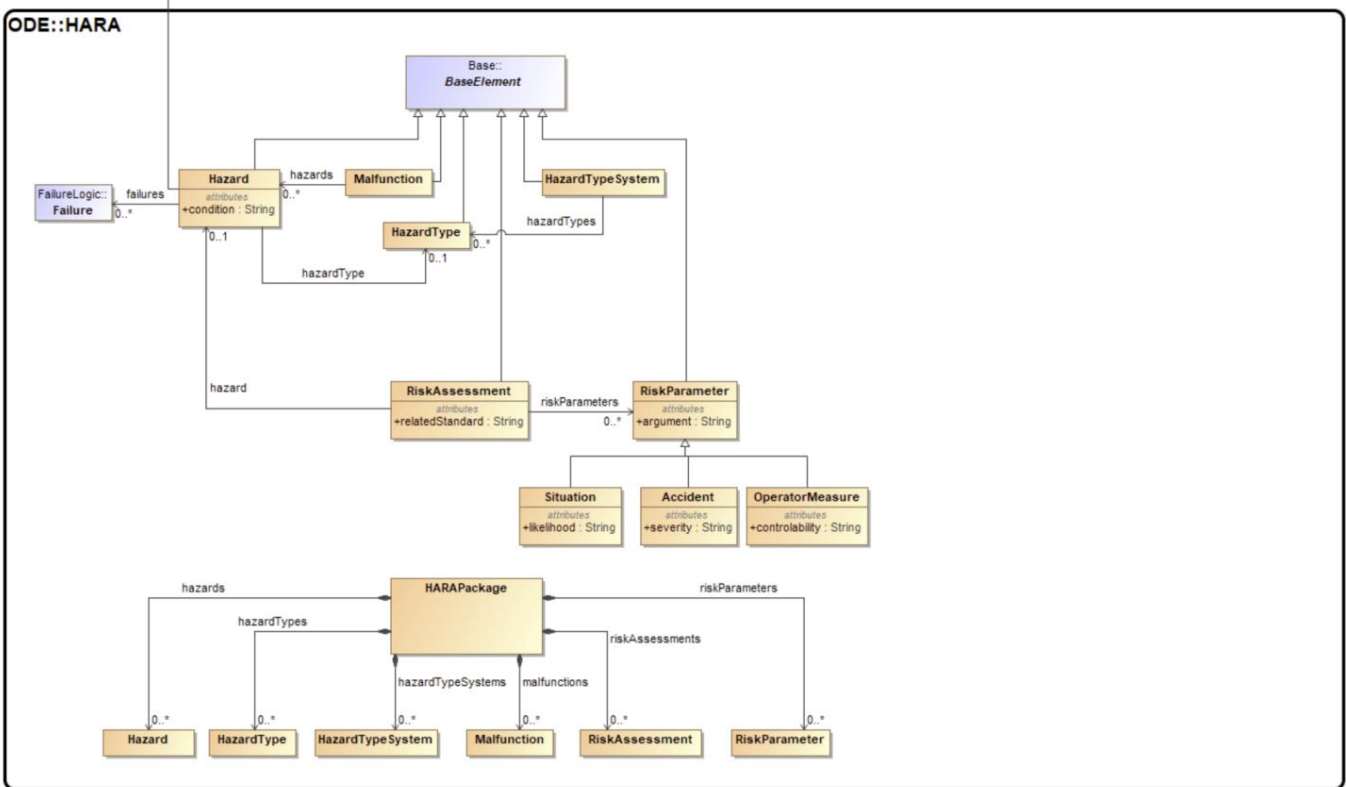
Figure 36 - ODE::HARA package

The HARA sub-package contains the necessary elements to support a hazard and risk analysis. The key elements here are the *Hazard*, which here specifically represents a situation involving loss of dependability with negative consequences, and the *Malfunction*, which is an unintended condition of a Function that causes one or more Hazards. Various risk parameters can be defined as part of the assessment and linked to the relevant hazards.
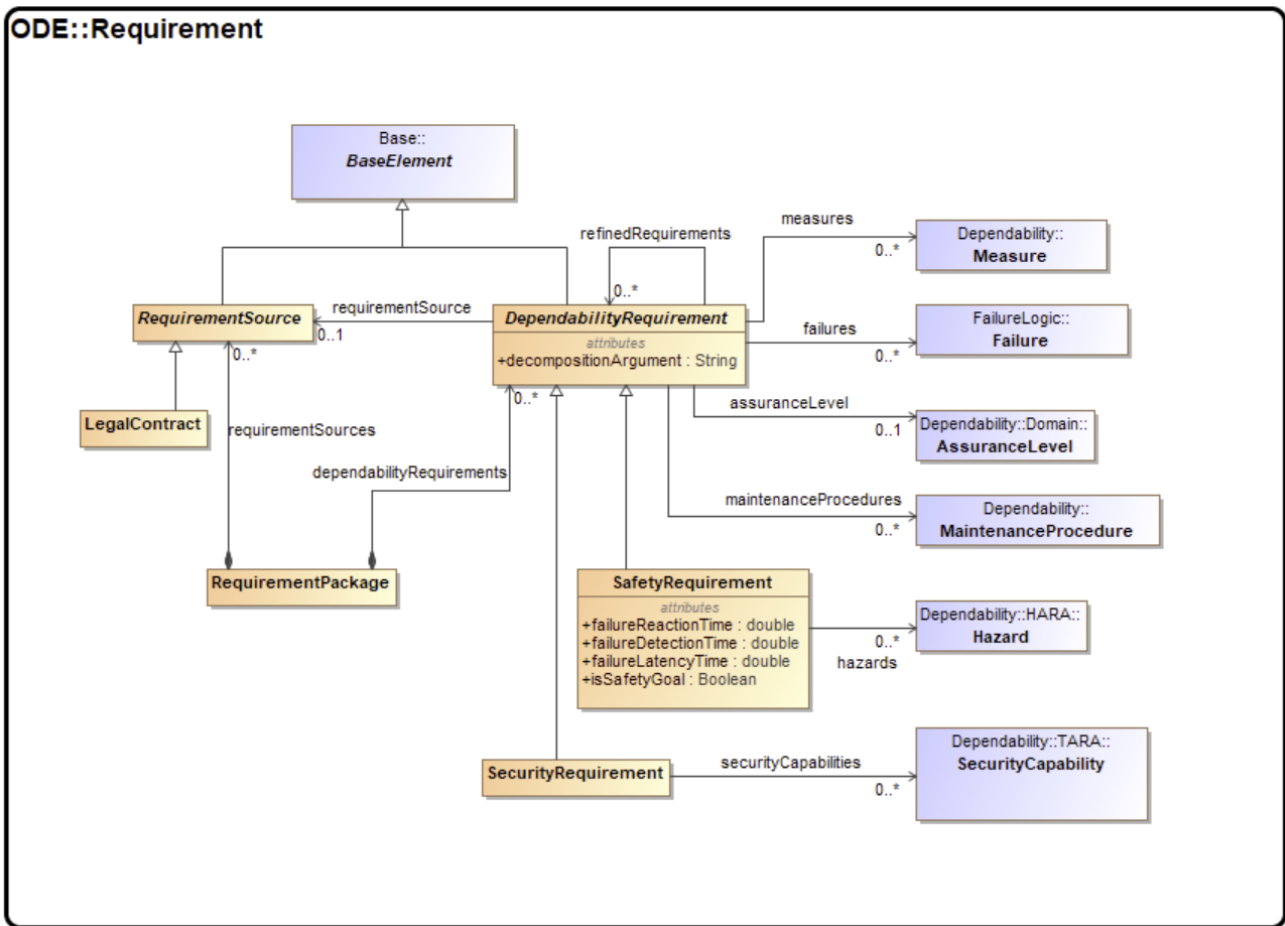
Figure 37 - ODE::Requirement package

The Requirement sub-package, as one might expect, models dependability requirements (including both safety and security requirements). A Requirement can have a source and links to the failures and hazards that it is concerned with, the measures that address them, and any refined requirements that the Requirement can be decomposed into.

Finally, the TARA sub-package is used for modelling a Threat Analysis and Risk Assessment:
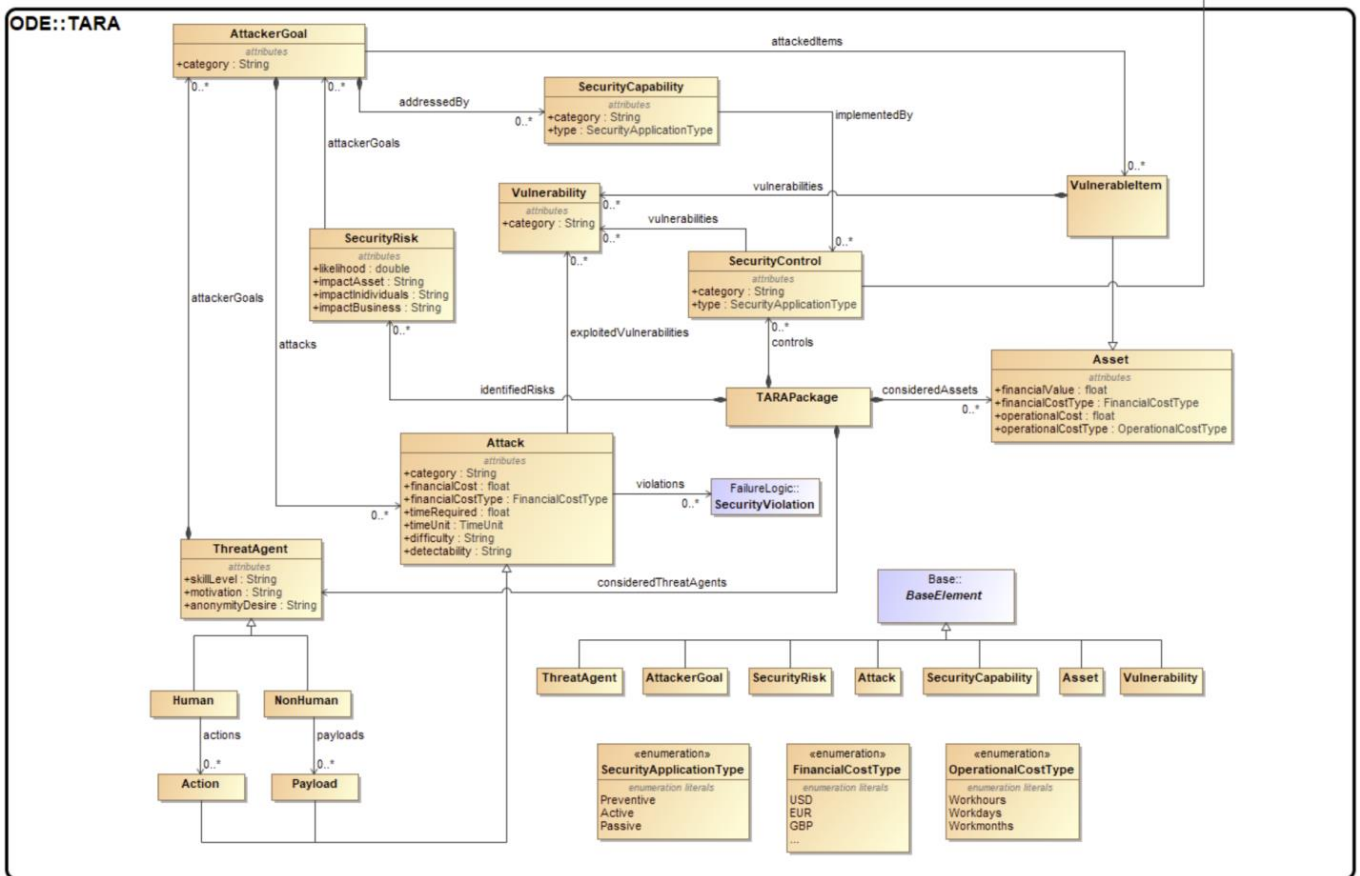
Figure 38 - ODE::TARA package

The TARA package includes key elements such as the Asset (an important concept or entity related to the operation of the system), Attacks (initiated by a ThreatAgent which threaten an Asset), and Vulnerabilities.

Moving past the Dependability sub-packages, the Validation package specifies elements to model a testing-based approach to validation. This is built around the idea of TestExecutions that satisfy (or not) TestCases, which have a variety of TestParameters and AcceptanceCriteria associated with them.
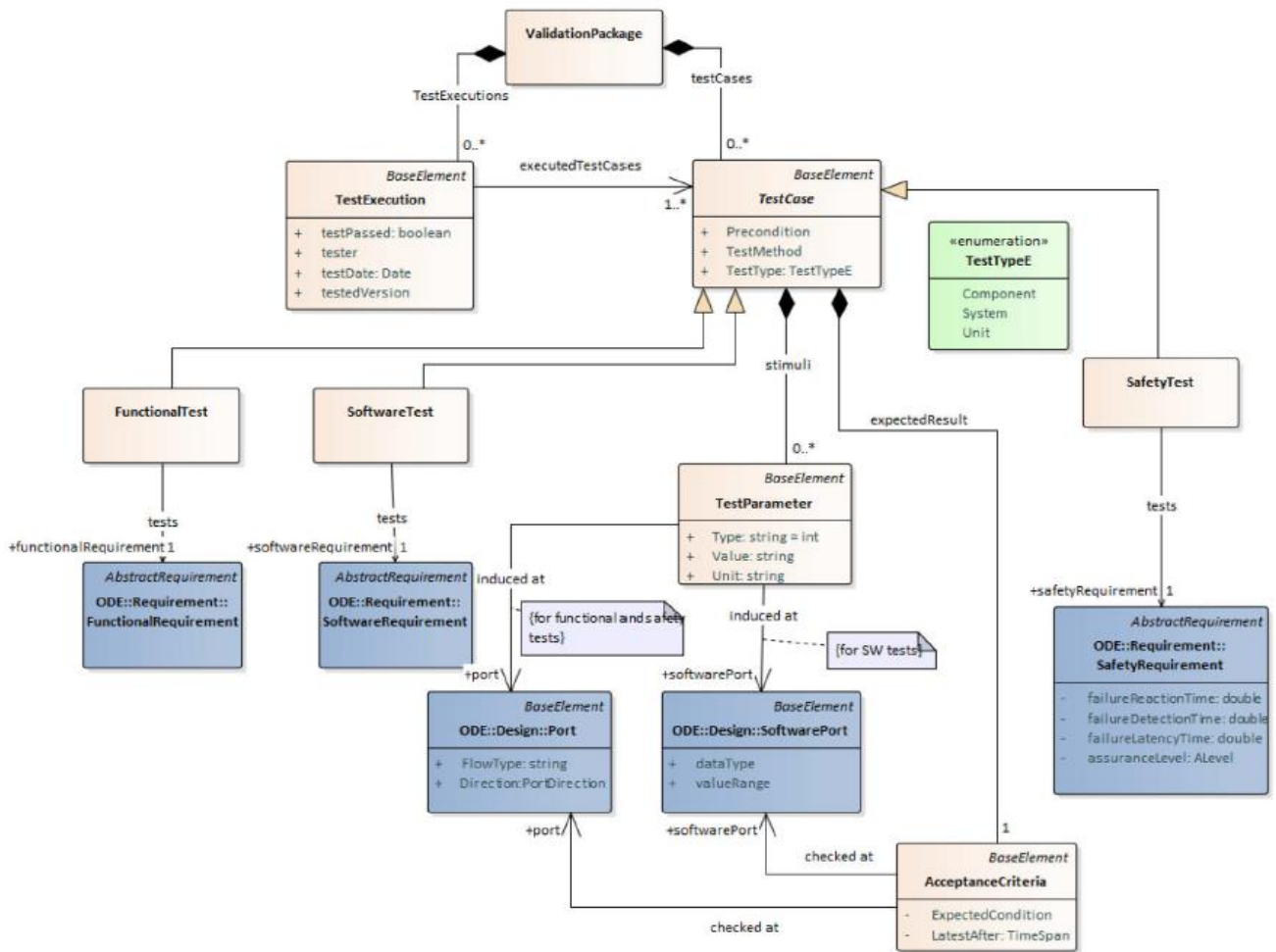
Figure 39 - ODE::Validation package

The remaining top-level packages of the product ODE metamodel are smaller and include:

- The Integration package, which integrates the various ODE packages together and supports possible future extensions.

- The Traceability package, which defines formal traces between SACM and the product metamodel packages (e.g. for the purposes of linking evidence in SACM or maintaining synchronisation).

- The Environment package, which provides elements to describe the operating context.

2.3 SAFETY ANALYSIS IN SESAME

The overall goal of SESAME is to develop an open, modular, configurable, model-based approach for systematic engineering of dependable multi-robot systems (MRS). The unique challenges of MRS — namely, complexity, intelligence, and autonomy — all pose obstacles to that goal, and a holistic solution needs to tackle them all together rather than coming up with disparate, disconnected answers to each problem.

At the core of any solution, therefore, there needs to be a common foundation of knowledge: a model or set of models that supports all the necessary activities to achieve the goal. These models need to be compatible with each other, modular (to support the open and distributed nature of an MRS), and be applicable both at design time and at runtime.

When it comes to dependability in SESAME, the key model is the EDDI or Executable Digital Dependability Identity. Like DDIs, these are intended to be model-based artefacts that contain all the required dependability information about a given system or component — such as safety and security hazards, their potential causes, effects, and possible corrective actions, as well as safety argumentation and information about the system architecture itself. They should also support any relevant dependability activities, whether that be safety analyses, allocation of requirements, or synthesis of safety argumentation. Unlike DDIs, however, EDDIs are intended to be fully executable at runtime, capable of communicating and adapting to changing circumstances to help ensure continued safe operation.

A model-based solution to dependability and safety in particular is critical to developing this capability. In particular, several of the tools and technologies discussed in this section will prove valuable in this regard.

2.3.1 Application of MBSA at design time

While much of the focus in SESAME is on runtime capabilities, design time activities are not neglected either. Although not all aspects of an MRS can be argued to be safe purely on the basis of design-time analyses, thanks to the uncertainty inherent in their dynamic, autonomous operation, there are aspects that can benefit from such a process — particularly those which are static or limited to a single platform.

Furthermore, much of the work to ensure safety at runtime relies on a solid foundation of prior dependability analysis done at design time: investigating possible hazards, establishing potential causes and effects, hypothesising mitigation measures etc. Even if some issues can only be solved at runtime, they have to be identified first before solutions can be developed.

To this end, many of the MBSA techniques described earlier could prove invaluable. In particular, tools and techniques that support compositional safety analysis — like HiP-HOPS's support for FTA and FMEA, or safeTbox's Component FTA — already allow an integration between the nominal system architecture models and the models of failure behaviour. They also provide some support for other activities that cover more of the design lifecycle, whether that be in terms of decomposing requirements via the allocation of integrity levels or in the generation of safety argumentation in the form of safety/assurance cases. Additionally, some work has already been done to enable these tools to support the ODE and thus create DDIs.

Use of such tools should enable us to:

- model a system architecture at various levels of abstraction;

- identify potential hazards;

- develop and decompose safety requirements across the system;

- annotate the model with failure behaviour and subsequently perform safety analyses (whether static, such as FTA or FMEA, and dynamic, e.g. Markov);

- verify that safety requirements are met and, if so, potentially generate safety argumentation semi-automatically.

Nevertheless, further work is necessary. Extensions to the ODE will be necessary to support the additional features needed to better represent MRS, and this in turn will necessitate modifications to existing safety synthesis and analysis algorithms. Integration or communication with other forms of models — such as the Executable Scenarios covered in WP3 — will help manage aspects that may not fall directly within the scope of a system model, such as environmental hazards and variability in operational tasks or operational context.

ODE extensions are likely to include (but may not be limited to):

- More support for dynamic models, including generic state machines and Bayesian networks.

- Some degree of support for ML safety techniques.

- Specification of runtime evidence monitoring requirements, e.g. types of data or event to be monitored, trigger conditions.

- Means to capture information about system and environmental variability.

- More interconnections between different model types to support later runtime diagnosis.

- Significant extensions to support security analysis and runtime security monitoring.

However, should such work be successful, it should be possible to provide comprehensive support for development of dependable MRS throughout the design lifecycle, from initial functional modelling and risk assessment to implementation architectures and safety analyses to verification of requirements and assurance cases.

2.3.2 Generation of runtime artefacts

Though the generation of EDDIs at design time is a worthwhile endeavour in itself, it is insufficient on its own to help ensure safe operation of MRS: it is also necessary to have runtime capability in order to tackle the other challenges of intelligence and autonomy. To that end, EDDIs capable of being executed at runtime to provide monitoring, diagnosis, and potential adaptation capabilities are needed, and these will derive to a

significant degree from the initial design-time EDDIs generated via the various MBSA approaches.

While it may not be possible to completely automate their generation, being able to synthesise frameworks or templates that can then be augmented with additional runtime-specific information will both save effort and ensure that the valuable information acquired at design time is able to be taken advantage of at runtime.

Although the runtime aspects of EDDIs are primarily the domain of WP7, they are also touched on further in Section 4 of this document.

3. THE CHALLENGE OF INTELLIGENCE

3.1 DEFINING THE PROBLEM

Artificial intelligence is not a new topic. The term itself was first coined in 1956 by acclaimed computer scientist John McCarthy [61] and the field began to develop across the 1960s and 70s with work on neural networks, natural language processing, and symbolic reasoning. The term "machine learning" (or ML) was defined in 1959 [62], though work in the field was already taking place before an umbrella term existed for it. Three main paradigms of ML exist:

- Supervised learning, in which the program attempts to learn general rules from example inputs and corresponding desired outputs;

- Unsupervised learning, in which the program is left to create its own structures about input data;

- Reinforcement learning, in which the program receives feedback to reward success or penalise failure.

Artificial neural networks, one of the most common forms of ML model, actually predate the term artificial intelligence, originating in the 1940s as a form of unsupervised learning and then evolving over the following decades with the development of back-propagation and multi-layer networks for deep learning. Artificial neural networks (NNs) are inspired by the biological structures of the human brain and consist of a group of interconnected *neurons*, typically organised in several layers as part of a directed, weighted graph. Each neuron is represented by a node in a graph and is generally connected to other nodes in the next layer by weighted links, with the weight indicating its relative importance.

A neural network is then evaluated by assigning input values to the neurons of the first (input) layer, and then using those to calculate the values of the next layer. This process repeats for each subsequent layer until the values of the final output layer are obtained. The value of a neuron can be obtained in different ways, but typically involves calculating a weighted sum of all inputs to the neuron and then applying an activation function, such as the Rectified Linear Unit function (where $ReLU(x) = max(0, x)$) [63].

Critically, during training NNs act as an adaptive system, meaning they are able to change their structure (or weightings) based on the input to the network. In general terms, this is thus how a neural network 'learns': sample input is provided and compared to expected output, and the weightings of the network are adjusted accordingly to minimise the deviation between expected and actual output. In this way, a NN can be trained to generalise a task — such as recognition — from a finite set of training data to unseen, real-world data. Doing so is often more efficient than developing an equivalent algorithm manually, especially since NNs can be tailored to specific uses based on the training data provided.
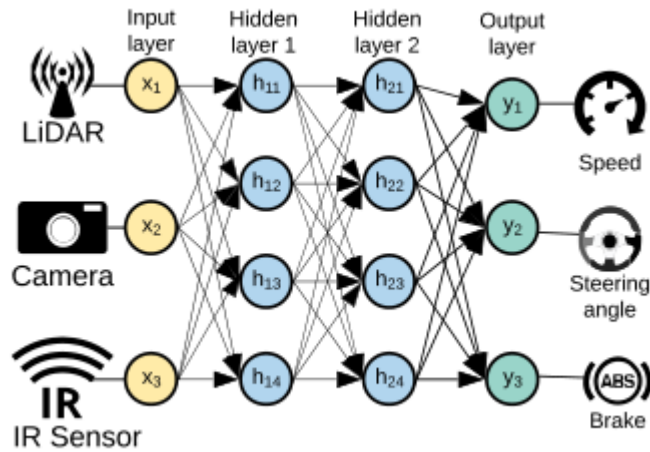
Figure 40 - An example neural network (from [64])

Neural networks can come in a variety of forms and have a range of applications. For example, one of the most common forms of NN used for image recognition and computer vision is the convolutional neural network (or CNN). More generally, deep neural networks (or DNNs) are those in which there are multiple 'hidden' layers between the input and output layers. DNNs are a typical tool used in deep learning, a form of ML that focuses on artificial neural networks specifically.

It is only in recent years, however, that the potential for widespread usage of machine learning in general and deep learning in particular has emerged. In many cases, these applications are safety-critical systems, such as environmental perception and self-navigation in robots or diagnostic support in the medical domain. There are several such examples from amongst the SESAME use cases; for example, the ability of a drone or robot to detect the presence of a nearby human is important to prevent unintended exposure to UV-C light in the Locomotec disinfection robot use case, and the ability to detect fungal infection in crops via drone observation is a key element of the Domaine Kox / Aero41 / LuxSense vineyard use case.

Autonomous systems like MRS stand to benefit enormously from the potential offered by ML — but only if the artificial intelligence can be trusted to perform safely [65]. While performance of ML algorithms continues to improve, safety assurance is perhaps *the* key challenge to be overcome before ML solutions to such problems can find widespread acceptance.

Part of the reason for this is that one of the fundamental problems facing ML proponents is that the decisions made by ML components are both opaque and inherently uncertain. While this uncertainty can never be entirely erased, methods for quantifying the degree of uncertainty would at least provide some measure of confidence in the results and allow for requirements on ML safety and performance to be more easily verified. One key aspect of this is the property of *robustness*, i.e., the quality of a DL model such that its decision is unaffected by small perturbations of its input [66]. Even highly accurate neural networks can be fooled by so-called *adversarial examples* [67] — taking a correctly classified input, applying a slight modification to it (e.g. random noise), and receiving an incorrect classification as a result.

$$x \qquad \text{sign}(\nabla_x J(\boldsymbol{\theta}, \boldsymbol{x}, y)) \qquad \begin{array}{c} \boldsymbol{x} + \\ \epsilon\,\text{sign}(\nabla_x J(\boldsymbol{\theta}, \boldsymbol{x}, y)) \end{array}$$

"panda"   "nematode"   "gibbon"

57.7% confidence   8.2% confidence   99.3 % confidence
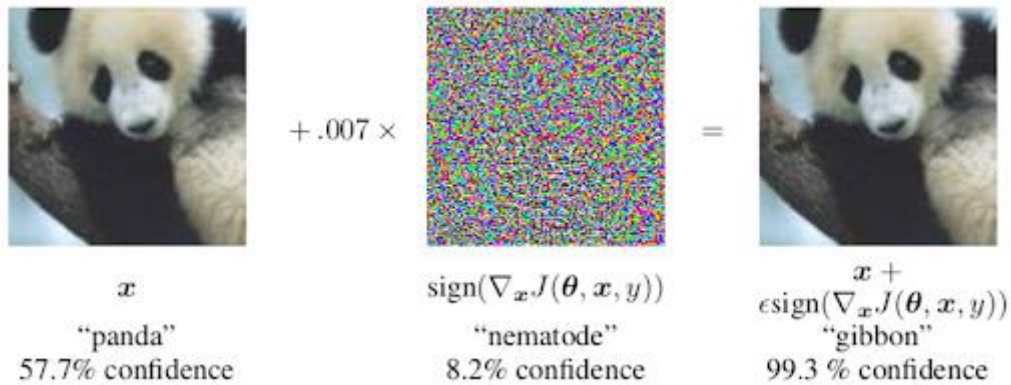
Figure 41 - An adversarial example leading to a misclassification[34]

More generally, even the best trained DL model would likely be unpredictable in the face of unexpected input it had not been trained for — data which is *out-of-distribution* (or OOD) [68]. The ability to detect such input and assign a low confidence to the ensuing decision at least enables the system to warn of potential risk in such cases.

This latter point touches on another important aspect of ML safety: that of *explainability*. As mentioned earlier, one of the issues with ML components is that their behaviour is largely opaque; they are not like manually crafted algorithms, where the underlying code can be examined, and so it can be difficult to know why they behave as they do. Finding ways to explain and justify the decision made by a NN on the basis of its input helps reduce the opacity of the ML component, thereby improving trust. This latter quality is sometimes termed 'XAI' — explainable AI.

The ability to determine the accuracy (and thus safety) of ML components, as well as explain their reasoning, is a vital step towards the effective assurance and regulation of AI in safety-critical applications. A safety requirement is of little use if there is no way to determine whether or not it has been upheld, and safety argumentation is difficult to produce if there is no explanation behind the behaviour of the system in question. An analysis of the ISO 26262 methodology, for instance, found that up to 40% of software safety methods are not directly applicable to ML models [69]. Consequently, until such methods exist, it is also hard to develop suitable standards to govern the use of AI-enabled safety-critical systems [70].

3.2   STATE OF THE ART: SAFETY OF MACHINE LEARNING

Unlike traditional system safety, which has had decades of development across a wide range of domains to produce tried and tested analysis techniques such as FMEA or FTA and well-regarded paradigms like MBSA, ML safety is still a new and evolving field. While many approaches have been proposed, they often focus on different goals or have a relatively narrow applicability (e.g. a specific type of model or a certain type of input).

The table below provides a summary of some relevant research in the field. Afterwards, key examples will be discussed in further detail. The table indicates whether an approach is model-specific (i.e., needs access to the model's underlying parameters and structure) or model-agnostic (i.e., treats the model as a black box and requires only the

---

[34] http://machinelearningintro.uwesterr.de/attacks-on-ml-models.html

inputs and outputs), the type of input (e.g. images, tabular/numeric data), the type of task (regression or classification), and whether or not it can operate at runtime.

Table 3 - Summary of various ML safety approaches

Approach	Features	Access Type	Input Type	Task Type	Run-time?
DeepCert [71]	Aims to verify the robustness of DNN image classifiers in terms of sensitivity to image-based perturbations, e.g. blur, haze, contrast etc. Instead of measuring small pixel variations, these contextually relevant perturbations are encoded and quantified specifically. Demonstrated via integration with the Marabou DNN verification toolbox.	MS	I	C	No
DeepImportance [64]	Presents a systematic methodology for DL testing with new Importance-Driven Criteria. This allows a layer-wise functional understanding of DL components — the causal relationships between neurons — and thus makes it possible to assess the semantic diversity of a test set in terms of testing important neurons (in effect, a form of test coverage). Has an open source tool.	MS	T/I	R/C	No
Marabou [63]	A verification tool that can query fully connected and convolutional DNNs to provide a reachability and robustness assessment for a given neural network. Requires internal knowledge of the DNN to work as it performs a lazy search to locate solutions to non-linear constraints on the model.	MS	T/I	R/C	No
NN-Dependability [72]	Proposes new dependability metrics to measure the effect of uncertainty elimination in the ML/DL lifecycle. Also provides a formal reasoning engine to guarantee ML/DL behaviours.	MS	T/I/TS	C	Yes
ReAsDL [70], [66]	Focuses on the impact of the operational profile on robustness. Divides the input space into small cells and evaluates the reliability of the ML/DL based on robustness and operational profile of those cells. Prototype tool available online.	MA	T	C	No
Safe AI [73],[74],[75],[76]	A collection of related approaches, e.g. DiffAI, DL2, AI^2, PRIMA etc. Their main focus is on possible perturbation to the input space (adversarial examples) and providing robust, safe, and interpretable solutions and certifications.	MS	T/I	C	No
SAFE-DNN [77]	Investigates the property inference in DNNs as part of the verification process. Combines supervised and unsupervised learning by	MS	T/I	C	No

Approach	Features	Access Type	Input Type	Task Type	Run-time?
	augmenting the feature space of the (supervised) DNN with features extracted by an (unsupervised) spiked neural network, increasing robustness of the DNN.				
Safeguard AI [68]	Calculates probability for out-of-distribution input as confidence loss and adds that probability to the existing loss function. Intended for use during training by identifying OOD samples and generating improved training data using a GAN to minimise confidence loss.	MS	I	C	No
SafeML [78]	Uses statistical distance measures to quantify the distributional shift. Then estimates the accuracy, updates the uncertainty, and evaluates reliability.	MA	T/I/G[35]	C	Yes
Uncertainty Wrappers [79], [80]	Focuses on three main ML verification domains: model performance, input quality, and scope compliance. Provides a set of useful functions to evaluate the existing uncertainties in each step.	MA	T/I	C	No

Key:
- MA/MS = Model-agnostic / Model-specific
- T/I/TS/G = Tabular numeric data / Image data / Time-series / Graph data
- R/C = Regression / Classification

Most of the approaches in the table are model-specific, meaning they require internal knowledge of the DNN to operate. While this can yield more in-depth information, it also limits the applicability of the approach and generally increases the complexity in the process. Model-agnostic approaches, while potentially less expressive, require only the inputs and the outputs (e.g., test data and resulting classifications), meaning they can be applied to a wider range of DL components, including those where access to the underlying model structure is not available.

The ability to operate at runtime is also valuable in a SESAME context, though few approaches allow this.

3.2.1 Maribou

Maribou [63] is a DNN verification and analysis tool[36] which builds on an earlier project called Relupex [81]. Both transform the problem of DNN verification into a constraint satisfaction problem and use an SMT[37]-based, lazy search technique to obtain a solution that meets the given constraints. Non-linear constraints are treated lazily as it is possible that they may prove to be irrelevant and thus do not need to be addressed.

---

[35] Use of graph data input for SafeML is still in early development
[36] https://github.com/NeuralNetworkVerification/Marabou
[37] Satisfiability Modulo Theories

Maribou further attempts to simplify the non-linear constraints by deducing new facts about them.

Maribou has native support for fully connected convolutional DNNs. Unlike its predecessor, it is not limited to only ReLU activation functions. It also has support for a divide-and-conquer approach to solving, in which an initial time limit is set; should this timeout be reached, the input query is decomposed into further sub-queries (with a new, larger time limit) and each sub-query is then processed individually (and potentially in parallel). A simplex-based linear programming core is used as the internal solver while input queries can be obtained via text format or via a TensorFlow model; similarly, properties can be provided via text format or compiled into the solver via Python code.
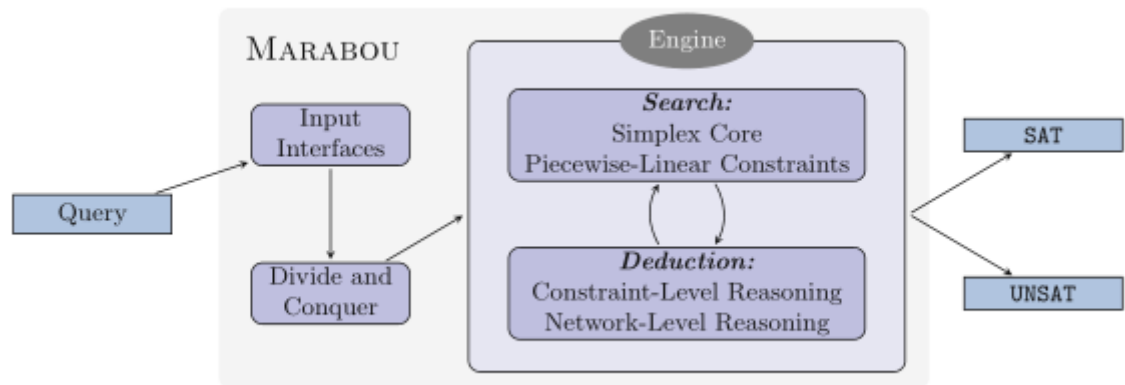
Figure 42 - Components of the Maribou tool (from [63])

The tool treats each neuron as a variable, and thus attempts to find an assignment of neurons that satisfies the constraints. Lower and upper bounds are also maintained for each variable over the process, and in each iteration, variable assignments are adjusted to attempt to correct any violated constraints. The loop is guaranteed to eventually terminate, although only for activation functions that are piecewise-linear. The deduction mentioned earlier helps to tighten and refine the upper and lower bounds via network-level reasoning.

As Maribou is a model-specific technique, it requires internal knowledge of the DNN in question.

3.2.2 ReAsDL

The ReAsDL approach[70] is an attempt to address the potential impact of operational profiles on DL robustness. Operational profiles are a concept that originates in software testing; an operational profile is meant to quantify how software will be operated, thus allowing testing efforts to be focused on the parts of the software liable to be used the most.

Robustness, as defined earlier, is the property of a DNN to resist the effects of small perturbations of input on its decisions. This can be framed as saying that all inputs in a particular region have the same predicted label. If there are inputs in that region which receive a different classification, then they are adversarial examples that reduce robustness. Note, however, that robustness is not necessarily the same as reliability or

accuracy: a DNN that always outputs the same classification for any given input is perfectly robust, but hardly reliable. Robustness only translates into reliability if the classifications produced are accurate with respect to the ground truth.

Robustness is heavily impacted by the r-separation property, i.e., the distance in the input space between input data points with different ground truth labels. This is intuitive: if there are larger differences between inputs with label X and inputs with label Y, the DNN is more likely to be robust; conversely, if there is little difference between them, then robustness is likely to be reduced.
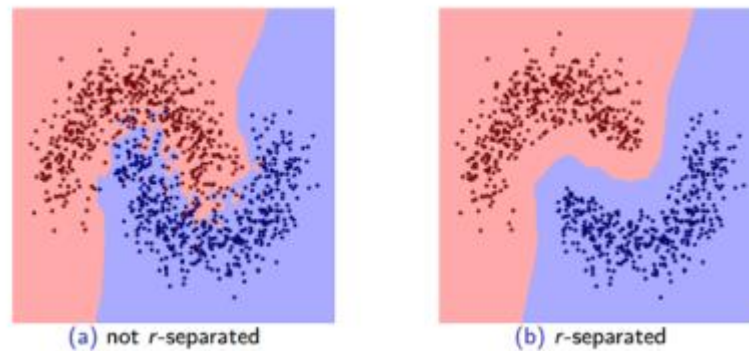
Figure 43 - R-separation (from [70])

ReAsDL takes advantage of this concept via the use of partitioning. The input space is divided up into cells according to the r-separation value. There are three types of cell:

- For cells with existing test data points (i.e., inputs from the training set) that share the same ground truth label, the ground truth of all future real-world inputs in that cell is based on that of the test data. For example, if all test inputs in a cell C have ground truth label X, then ReAsDL assumes that all future inputs in C should also have label X.

- For cells without any test data points, ReAsDL assumes that the DNN's classification is always correct (in the absence of any evidence to the contrary).

- For cells with test data points with mixed ground truths (e.g., C contains test data with labels X and Y), the estimated accuracy is set to 0 as a conservative measure.

In this way, the problem is decomposed into a series of sub-problems: given cell C with known (or assumed) ground truth label Y, what is the probability of a random input in that cell being misclassified? Then the operational profile concept is essentially used to estimate the probability that an input will be in a given cell, which is derived from the original training set as a kind of weighting factor — determining how many data points are in each cell and assuming the same holds true for live data. The more likely a cell is to contain a future input, the bigger its impact on the overall robustness.

The assessment itself involves generating further test data using Monte Carlo methods to see how many match the predicted classification for the given cell. The original training data is effectively used to seed new test cases around them in the input space,

and the expected classification of the new test data is based on the r-distance and the seed's ground truth label. Overall reliability is therefore a summation of how reliable each cell is estimated to be combined with how frequently an input is expected to belong to that cell.

ReAsDL is model-agnostic and designed for pre-trained DL models. By analysing data input and model output it can yield pessimistic estimates of the probability of misclassification per input with confidence levels.

3.2.3 SafeML

SafeML[38] is a framework for the exploration of techniques for safety monitoring of machine learning classifiers at runtime. By using statistical distance measures, the aim is to be able to evaluate — and quantify — the difference between a given operational input and the trained context [78], [82]. By doing so, SafeML can yield a form of 'confidence measure', i.e., how confident we can be that the classification is correct based on how similar it is to the original training context. If the confidence is below a given threshold, mitigating actions can be taken in response, such as reverting to some safe state and/or disregarding the classification. In this sense, SafeML can therefore be used to monitor the uncertainty in the operational context of the ML system; when the system is operating out of its intended context, SafeML can warn us.

This basic concept is illustrated below.

---

[38] https://github.com/ISorokos/SafeML

SafeML for Classification
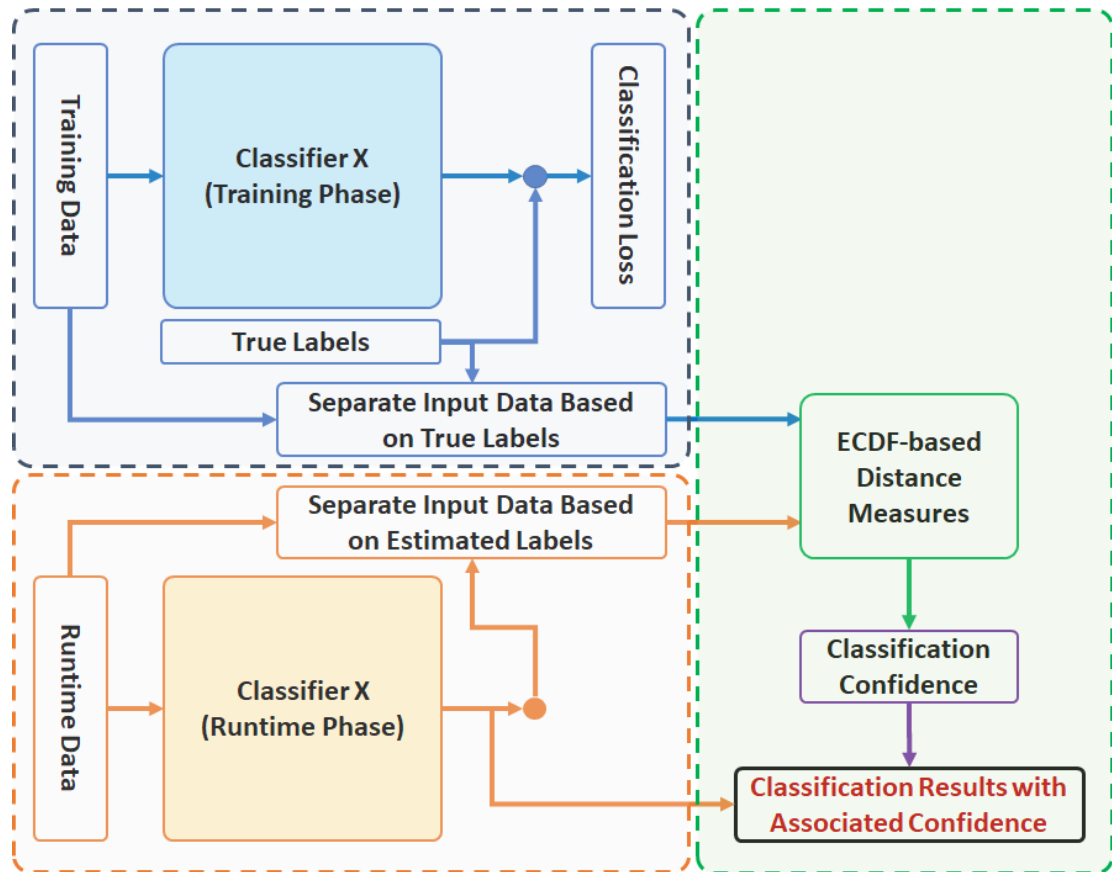
Figure 44 - SafeML concept

ML classification algorithms such as DNNs are typically employed to categorise inputs. The nature and purpose of the categories varies according to context. For example, in the medical domain, the purpose of the classification is often to detect abnormalities based on exceeding normal ranges. In other cases, such as object recognition, classification is often used to distinguish between classes of object or detect the presence of a given object (such as people or cars).

As described earlier, when the r-separation is low, classification errors become more likely because there is less conceptual distance between inputs that ought to receive different classifications. The diagram below illustrates this with a very simple example:
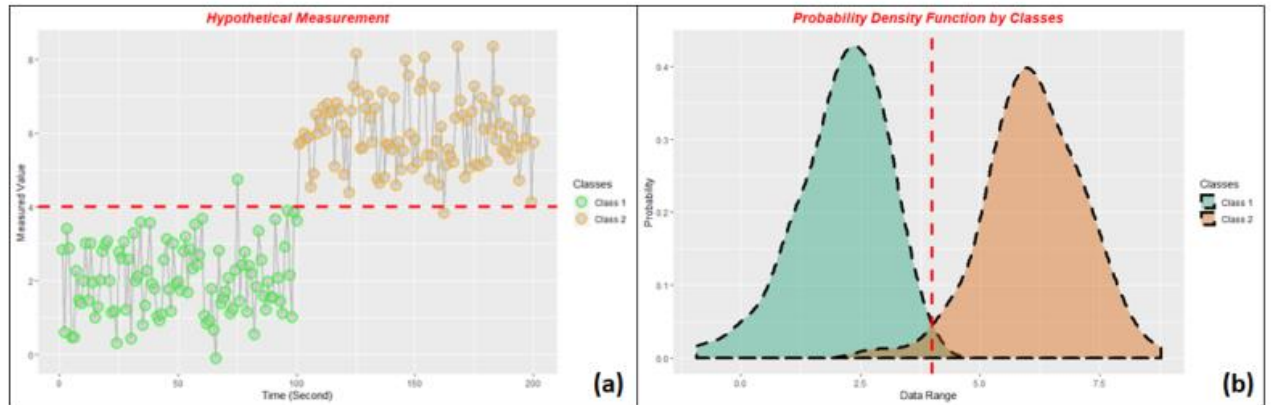
Figure 45 - Overlap between classes (from [78])

On the left, we see a range of input measurements given to a classifier *D*, where:

$$D(t) = \begin{cases} Class1, if\ 0 < n \leq 100 \\ Class2, if\ 100 < n \leq 200 \end{cases}$$

The threshold between classes is shown as the red dotted line but note that there are a few inputs that cross the threshold yet belong to the other class — reducing the r-separation. Because of this, the probability distribution functions (PDF) of the true classes overlap, as seen on the right of the diagram. It is this kind of overlap that makes misclassifications more likely, whether they be false positives (incorrectly identifying something as a given class) or false negatives (failing to identify something correctly as a given class). It can be shown therefore, as in [78], that there is a relation between the probability of error and the statistical difference between the cumulative distribution functions (CDFs) of two given classes. Because of this, Empirical CDF-based statistical measures can be used as a yardstick for the probability of error of a given input, and it is on this basis that SafeML functions.

Examples of the distance measures employed include Kolmorogov-Smirnov, Kuiper, Wasserstein, and Cramer-Von Mises, as illustrated below. Because these various measures assess different aspects of the ECDF — e.g. maximum distance, sum of the maximum distances in both directions, or the area between the two (continuous or stepwise) — they are used together to create an overall picture, thus minimising the risk that a single measure might cause an error.
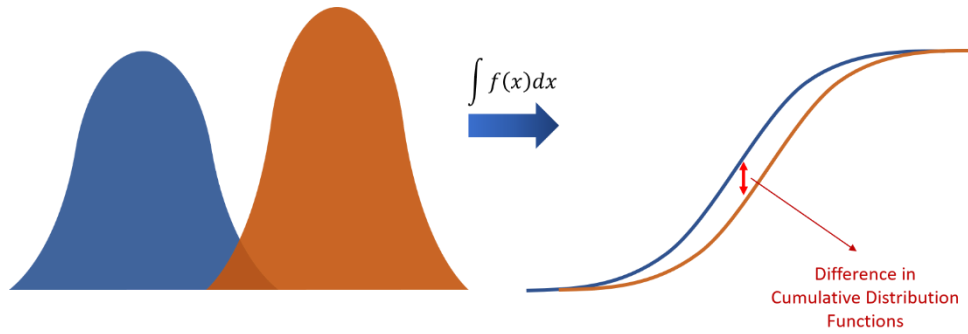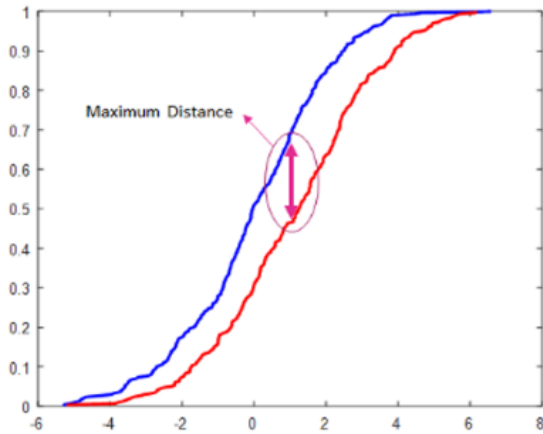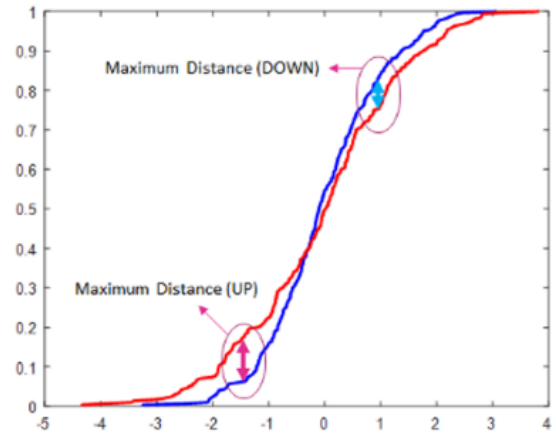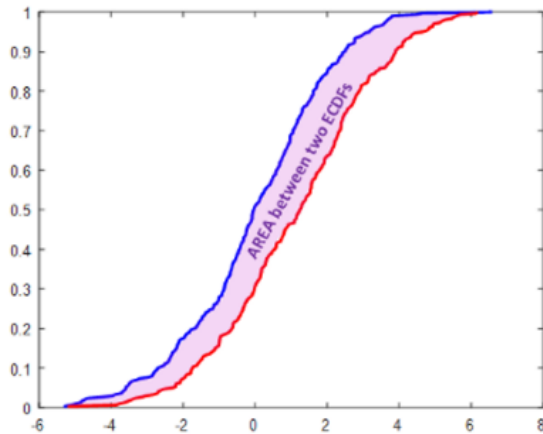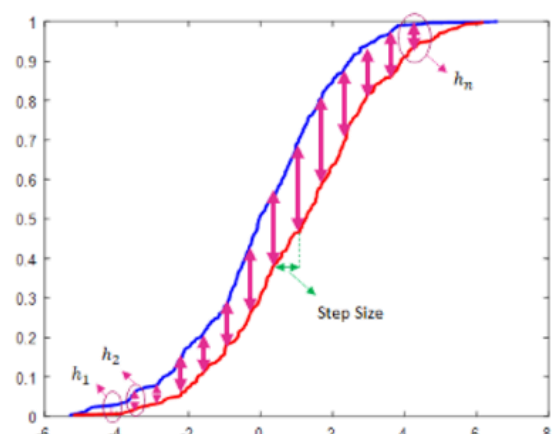
Figure 46 - Example SafeML ECDF distance measures

The following flowchart shows the SafeML process in its current state. In this flowchart, there are two main phases: 1) the training phase, which is an offline or design time procedure, and 2) the runtime phase, which is an online or real-time procedure.
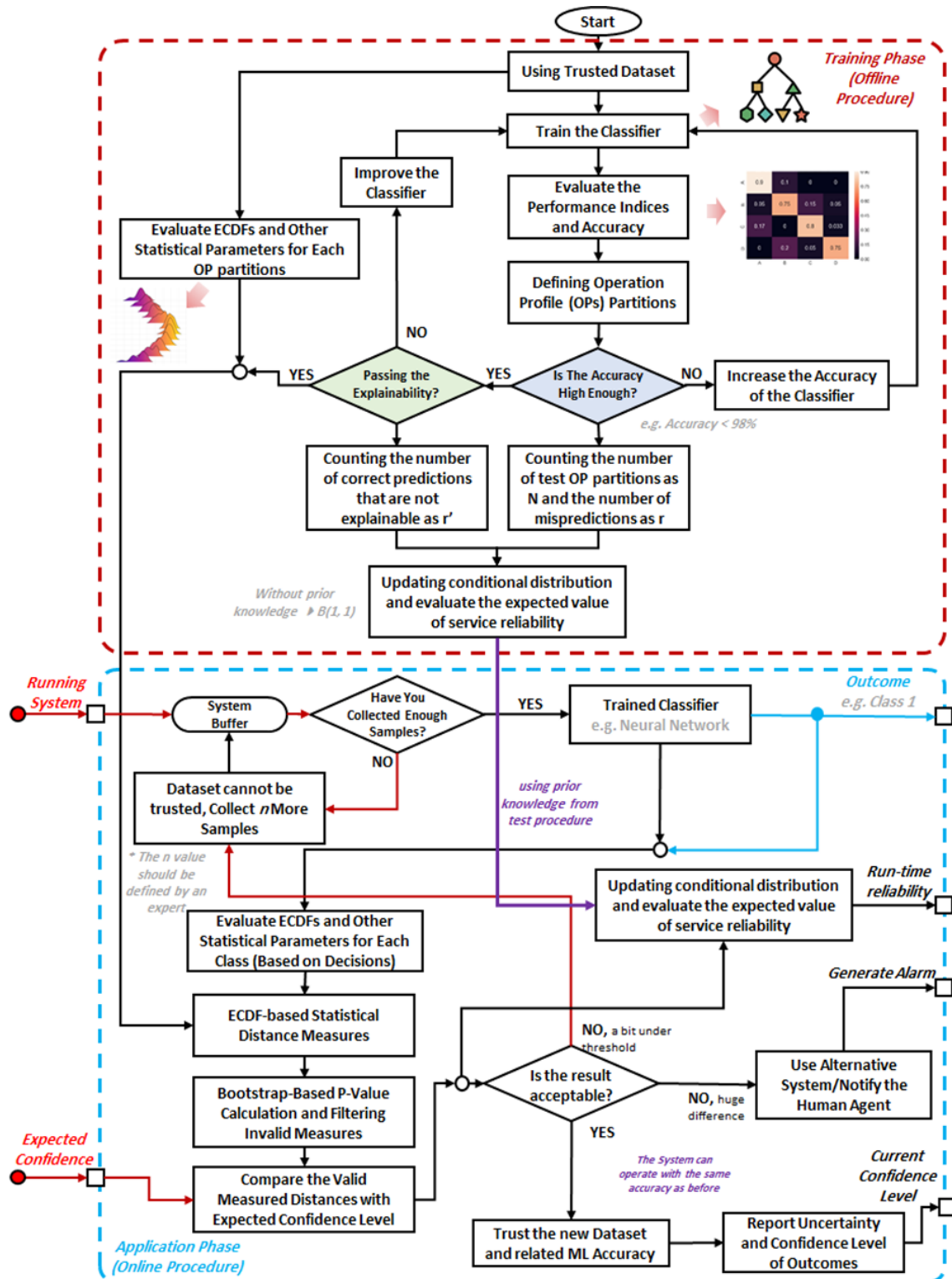
Figure 47 - The SafeML process

In the training phase, it is assumed that there is a trusted dataset that has been certified by a group of safety experts, i.e., so that the true classification of the data is known. The trusted dataset is needed to make sure that dataset labelling itself does not add any further uncertainty to the SafeML calculation.

Given the trusted dataset, the chosen ML/DL algorithm can then be trained and its performance can be evaluated through existing key performance indices such as accuracy, f1-score, area under ROC curve, etc. If the selected KPIs fail to meet the desired levels, the training phase is repeated until the trained model reaches a certain level of performance.

Having obtained a high-performance model, the next step is to check the explainability of the trained model. For checking the explainability we would apply our novel explainability approach called SMILE (see Section 3.2.5). There should be a trusted ground-truth for explainability to be able to measure the model explainability KPIs. With a model with high level of accuracy and high level of explainability, SafeML can store statistical parameters and a trusted dataset with regards to the true label of each class. In addition, both "number of misclassified samples" and "number of mis-explained samples" are stored to be used for reliability evaluation of the ML/DL classification task.

In the runtime phase, a buffer is used to store a certain number of samples to perform statistical analysis. Once enough samples have been collected, they are fed to the trained classifier to get the predicted labels. Based on the predicted labels, the statistical parameters (e.g. ECDF) of the collected data will be extracted. The extracted ECDF from runtime and design-time are then compared using various statistical distance measures, such as those described earlier. Furthermore, a bootstrap-based p-value evaluation is executed for each statistical measure to make sure their values are valid. Any invalid statistical distance measure will be filtered.

The statistical distance measures can then be compared to a desired/expected confidence level threshold. Based on the result, the reliability profile of the system is updated accordingly. There will be three main choices:

- If the result is acceptable (i.e., the confidence threshold is achieved), that means there is no distributional shift in the incoming data and the output of the trained classifier can be accepted with high confidence;

- If the result is not acceptable but very close to the threshold, the system may ask for more data (e.g. a second or third reading);

- If the result is not acceptable and falls well below the threshold, a proposed human-in-the-loop procedure can be applied. It should be noted that the decision of the human or expert in this scenario can be stored in the system as trusted data and it can be used to improve the system in the future.

SafeML has been applied to a variety of different case studies and different types of data. This includes artificial tabular/numeric data, security logs to detect DDoS attacks, and image recognition, as in the case of traffic sign recognition for autonomous vehicles [82]. Because SafeML is a model-agnostic technique, it does not require in-depth knowledge of the ML model in question, whether that be a DNN or something else. Nevertheless, it does require some adaptation to the type of data. For image data, for example, the images were converted to flattened vectors, allowing pixel-wise ECDF-based comparison.

How best to determine the acceptance distance/confidence thresholds in a systematic manner remains a subject of investigation. So far, acceptable distance thresholds have been specified only in terms of correlation to accuracy based on trusted datasets. If a method for systematically determining such thresholds can be found, it can be integrated as part of a wider ML quality assurance process, helping to improve overall performance of the approach.

3.2.4 Uncertainty Wrappers

Uncertainty Wrappers (UWs) [80],[79] address the unpredictable response of ML models by structuring this uncertainty in terms of a layered mathematical model, as seen in Figure 48.

The model fit layer refers to uncertainty originating from inherent limitations of creating ML models. Such limitations can be assessed as part of the typical testing approach applied as part of ML model development, and include evaluation by metrics such as accuracy, precision, recall, and the F1 score [83].

The input quality layer refers to imperfections in the quality of the input provided to the ML model during its operation. As an example of such input quality defects, we can consider an ML model trained to identify objects and is provided with images as visual input. If the images are provided by cameras whose lens have been damaged or have accumulated occluding debris e.g. condensation, snow, or dust, then these imperfections in the images may lead to the ML model not recognizing objects in them.

Finally, the scope compliance layer refers to uncertainty due to applying the ML model in conditions it was not trained for. For example, training an ML classifier to recognise vehicles using images from a specific region (e.g. urban environments), may lead to the classifier being unable to recognise vehicles in areas where the landscape differs significantly (e.g. snowy rural highways).
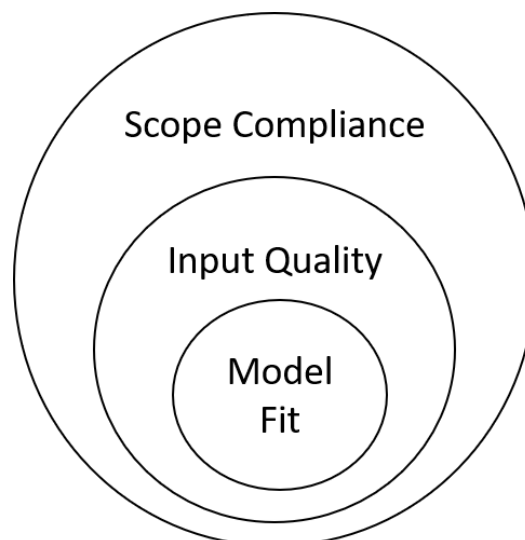
Figure 48 - Uncertainty Wrapper Layered Model

To apply UWs in practice, Figure 49 shows how the outcome of an ML model (aka Data-Driven Model) can be enhanced. In the example considered by the figure, the overall component (aka Data-Driven Component) is intended to detect pedestrians from

a vehicle, therefore uses vehicle-mounted cameras and vehicle sensors to acquire relevant input and context information.
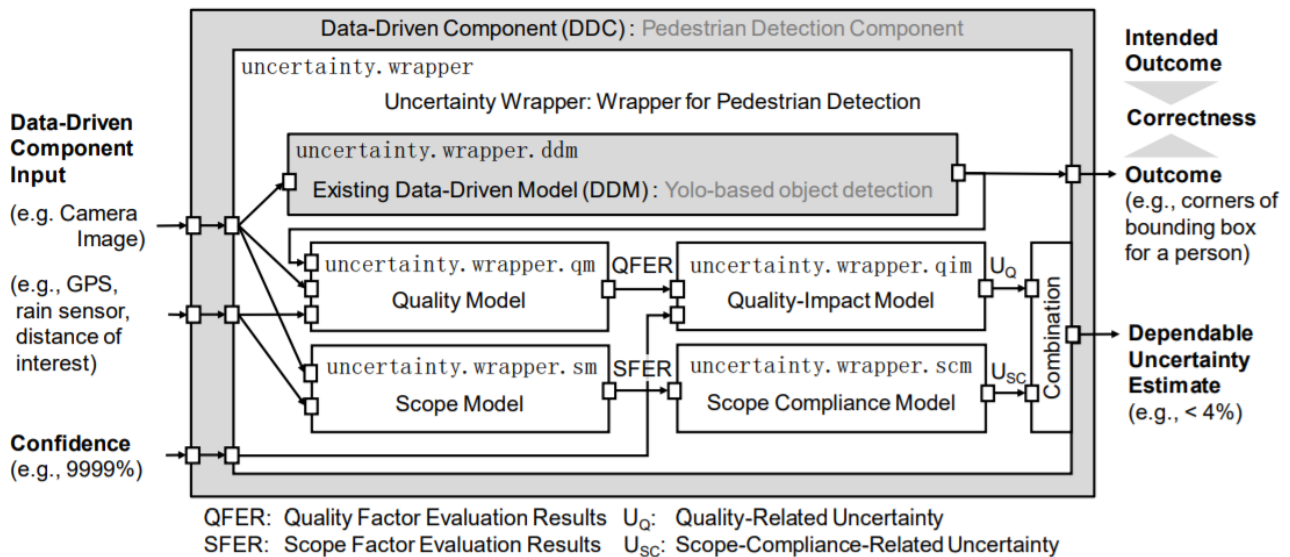
Figure 49 - Uncertainty Wrapper wraps a Data-Driven Model (from [79])

Specifically, within the component using the ML model, UW sub-models for addressing input quality and scope compliance are added. These correspond to the (Input) Quality Model and Scope (Compliance) Model seen in the figure. Both of the sub-models may consider sets of their own sub-models as factors (Quality-Impact and Scope Compliance Model) that should be considered when assessing the quality of the input or the determination of the application scope. Besides the input to the ML model and the contextual information made available to the UW sub-models, a user-defined confidence level is also specified as part of the enhancement, as seen in the bottom-left of the figure. This confidence level is used to determine, based on the UW sub-model outputs, an uncertainty estimate on the ML model's outcome, i.e., how uncertain is its response for the given input. In this sense, uncertainty is defined by the likelihood that the ML model outcomes are incorrect, with the UW-generated estimate providing a justifiable upper boundary for the given level of confidence.

Per [79], to construct a UW, a labelled dataset representative of the model's target application scope is needed, meaning it should be consistent with the conditions that will be observed and provided as input to the ML model during operation. For each dataset sample that will be provided as input to the ML model, the corresponding correct answer for that sample needs to also be provided. If this is not directly possible, then at least a definition of correctness is needed, so that the ML model outcomes can be evaluated on demand. Given such a representative dataset, appropriate data quality and scope factor models can be specified and composed into the wrapper's quality impact and scope models accordingly.

Quality factor models can be specified for a given application based on domain knowledge e.g. by understanding the impact that runtime environmental conditions may have on the input data quality e.g. rain, light, temperature etc. Quality impact models are responsible with providing the 'input quality' part of the overall uncertainty estimate, by accounting the results across all quality factors (as given by their

corresponding models). For this purpose, an information-gain-based decision tree can be trained to cluster similar uncertainties together. The tree can be trained to classify the ML model's outcome correctness (as established by the representative dataset), dependent on the quality factor model outputs.

Scope factors can range across:

- Boundary-based factors, which specify valid ranges for specific model inputs e.g. geolocation within a country's borders

- Condition-based factors, which model multivariate concepts e.g. temperature at a given geolocation

- Novelty-detecting factors, which aim to detect conditions that are outside the target application scope but went undetected by the other factors. The default option discussed by the authors in [79] involve the use of 1-class classification, where an ML model is created to determine whether a given sample belongs to the class (i.e. is within the target application scope) or not (i.e. outside the scope). As a complementary option to the existing approach, SafeML, presented previously in Section 3.2.3, could potentially be used for this purpose as well. We aim to further explore this potential synergy over the course of the SESAME project.

Scope factors can be combined by a corresponding scope compliance model based on whether each factor considers the ML model input to fall within the target application scope or not. The detailed specification of the scope compliance model requires domain and/or application knowledge to yield high precision. That being said, with some simplifying assumptions, an example of a basic scope compliance model can consider the input 'out-of-scope' when any one of its scope factor models determines so. More details are discussed in [80].

A UW tooling framework for creating and validating UWs has been developed in Python, and is compatible with the popular scikit-learn[39] open-source Python package for ML [79].

As a brief example of how UWs can be employed for uncertainty management of ML models in dependability-critical systems, the authors in [84] provide such an example for handling uncertainty of ML models used in autonomous vehicles. In their example, autonomous vehicles maintain distance from their predecessor in traffic through the use of kinematic models that also account for a minimum safe inter-vehicle distance, given the operational conditions e.g. road friction. However, if the road friction or other relevant conditions were being estimated by ML models, the uncertainty of the latter may introduce risk of underestimating the minimal safe distance. The authors proceed to demonstrate how UWs can be constructed to provide justifiable safety margins that increase as the uncertainty of the ML model increases, thereby increasing the inter-vehicle distance in such cases.

We anticipate that MRS are likely to feature similar concerns when faced with their own kinematic and other decision challenges that depend on input from ML models e.g. when determining whether to turn on the UV-C lamps given the lack of detected people

---

[39] https://scikit-learn.org/

in view of the lamps. We intend to explore the potential of UWs further in similar directions as part of the SESAME project's ongoing work.

3.2.5 Explainability of ML

As described earlier, one of the shortcomings of machine learning models is that they are generally black boxes: whether their decisions are correct or not, it is not easy to understand *why* those decisions were made. Such explanations are, however, critical in determining the trustworthiness of ML models. Explainability also helps with the development of models; if we can see and understand the cause of errors, we can attempt to correct them.

Although explainability of ML is still a nascent field, two approaches that work towards achieving it are described next.

3.2.5.1 *LIME: Local Interpretable Model-agnostic Explanations*

Local Interpretable Model-agnostic Explanations — known as LIME — is a model-agnostic algorithm that aims to explain the predictions of an ML model by approximating it locally with an interpretable model [85]. "Locally", in the context of LIME, means *local fidelity*; while an explanation separate from the model itself (as in the case of all model-agnostic approaches) by definition can never be 100% faithful, local fidelity means that it corresponds to the model behaviour in the vicinity of the instance being predicted. This means that features which are less important overall may be more important for the given explanation, or vice versa.

The other key objective of LIME is *interpretability*, i.e., providing a human-understandable explanation of the relationship between input variables and the ML model's response or decision. Note that making it easy to understand is prioritised over comprehensive coverage; rather than showing a bewildering number of input variables, for instance, it may limit it to only those that are most important or have the biggest impact, to make the relationship easier to follow. This priority sets up a tension between fidelity (a comprehensive explanation would cover all factors) and interpretability (i.e., ease of understanding).

To achieve local fidelity in its explanations, LIME randomly samples the local context of a given input needing to be explained. These samples are then weighted by the distance between them and the original instance/input. From these, an explanation that is locally (but not globally) faithful is generated. This process is illustrated in the figure below.
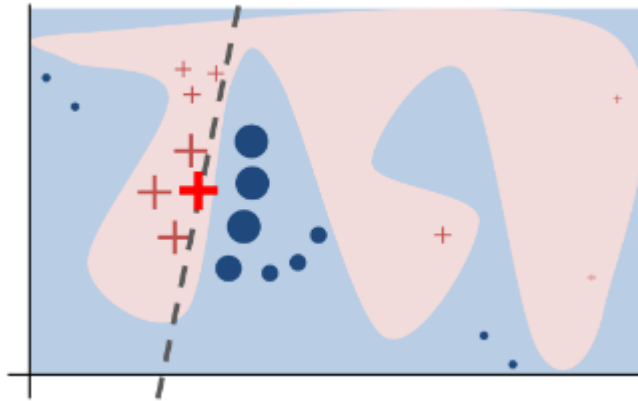
Figure 50 - LIME example (from [85])

Here, the model's decision function is represented by the two colours, but is too complex to be represented linearly (at least globally). The bolded red cross indicates the instance needing to be explained. LIME then randomly samples other instances (represented by the other dots and crosses), asks the model for their predictions, and weights them according to distance (here indicated by relative size). The dashed line is the learned explanation — one that is interpretable, since it is an easily understood linear function, and locally faithful to the instance in question.

More generally, the LIME process is as follows:

1) Generate random samples and feed them to the black box ML classifier to get the predicted labels.

2) Calculate the Euclidean distance between the given sample and each randomly generated sample.

3) Using a kernel function, the calculated distances can be mapped to weights.

4) Having a set of positions, predictions, and weights from previous steps, a weighted linear regression can be trained and its coefficients can be used as the local explanation for each feature and each class.

The following figure illustrates the overall view of the procedure explained above.
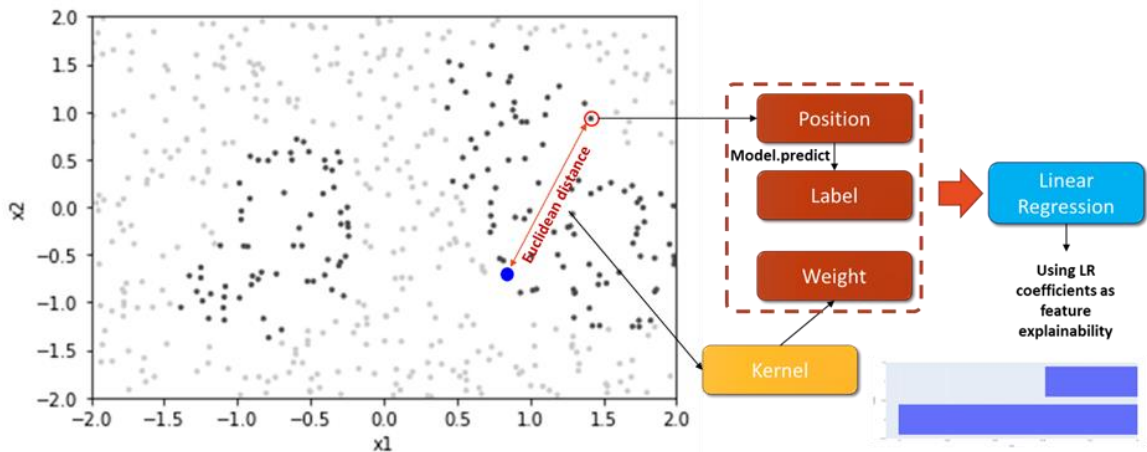
Figure 51 - An example of how LIME works

Since it is model-agnostic, LIME is not restricted to a given model. Consequently, it has been applied to a variety of models, including text-based and image-based problems.

3.2.5.2 *SMILE: Statistical Model-agnostic Interpretability with Local Explanations*

Understanding and predicting the behaviour of machine learning algorithms through explainability and interpretability is one of the vital steps towards making them dependable. LIME is one of the well-known approaches that has received significant attention in the field [85]. However, it is proven that this approach is able to be fooled by adversarial attacks [86]. To make this approach more robust to adversarial attacks and more reliable, we are currently developing a new approach called SMILE: **S**tatistical **M**odel-agnostic **I**nterpretability with **L**ocal **E**xplanations (with publications to come). The proposed approach uses the same overall procedure as LIME but makes it more robust by modifying its weight calculation and embedding empirical cumulative distribution function (ECDF)-based statistical distance measures.

SMILE supports tabular (numerical), image, text and graph-based datasets and its capabilities and performance have been measured through various examples. In addition, the robustness of SMILE against adversarial attacks is tested and compared with LIME and SHAP (**SH**apley **A**dditive ex**P**lanations, [87], [88]) and it is shown that it is more robust to such attacks. It should be noted that SMILE is capable of dealing with both classification and regression tasks and it is completely model-agnostic. Thus, the approach can be applied to any machine learning algorithm. Regarding graph-based datasets, it can perform node, edge, and feature explainability with a high level of consistency and faithfulness.

SMILE for tabular/numeric datasets
Similar to LIME, in SMILE, for a given sample, the local explanation can be generated using the following steps:

1) Generating random samples and predictions as with LIME.

2) For each randomly generated sample, the other samples in its vicinity are found and the expected value of their labels (machine learning predictions) is calculated.

3) In comparison to the LIME approach, instead of calculating the Euclidean distance, the ECDF-based statistical distance between two sets: a) given sample plus some random samples around it and b) each randomly generated sample and some random samples around it.

4) Using a proper kernel function, the calculated statistical distances can be mapped to weights.

5) Similar to the LIME approach, having a triple set including positions, the expected value of the predictions and weights from previous steps, a weighted linear regression can be trained and its coefficients can be used as the local explanation for each feature and each class.

The following figure illustrates the overall view of the procedure explained above.

Figure 52 - An example of how SMILE works

SMILE for image datasets

This idea can be expanded further for image and text-based datasets. In the distance calculation part, instead of using cosine distance between a vector representing each perturbed image or text and a vector representing the original image or text (as in LIME), we can use the ECDF-based statistical distances. The result shows more accurate results for image and text explanations.

Figure 53 - SMILE flowchart for explaining image-based classification or regression

The figure above delineates the flowchart of SMILE for the image-based dataset (either classification or regression tasks). In this figure, it is assumed that we already have a trained image classifier (e.g. a trained classifier for traffic sign recognition). For each input image, its super-pixels will be extracted to reduce the SMILE distance calculations and make it faster. Based on the number of detected super-pixels, K number bi-nominal random perturbation vectors are generated, a factor that impacts the scalability, accuracy, and consistency of the approach. Each element of a perturbation vector represents the status of a corresponding super-pixel (0: means the super-pixel should be off, and 1: means the super-pixel should be on). So, based on these perturbation vectors, K perturbed image can be generated and for each one of them, we can get the prediction from the trained machine learning algorithm (e.g. image classifier).

The ECDF-based statistical distance measures can be used to generate the distance of each perturbed image from the original sample. A proper kernel function can be used to convert statistical distances into weight values. The kernel function maps the distance values to the weight value that can be used in the weighted linear regression. In SMILE we have used a Gaussian kernel function with hyper parameters of epsilon and kernel width. The accuracy of the explanations can be tuned with these hyper-parameters. It should be noted that the effects of using other kernel functions (like Weibull) will also be investigated.

Having a triple set of perturbations, predictions and weights, a weighted linear regression model can be trained as a surrogate model. By sorting the coefficients for the trained weighted linear regression model, M super-pixels that gained bigger coefficients

Confidentiality: Public Distribution

can be selected and presented as explainability. Therefore, those M super-pixels can show which part of the image has more impact on the machine learning's decision.

SMILE for Text Datasets

Similar to the flowchart above, the text-based version is illustrated in Figure 54. Instead of finding super-pixels we have some functions for text pre-processing. Also, to compare perturbed text strings with the original one with ECDF-based statistical distance measures, we need to use word2vec embeddings. In our implementation, we have used Gensim Word Embeddings. The rest of the procedure is the same as explained for image datasets.
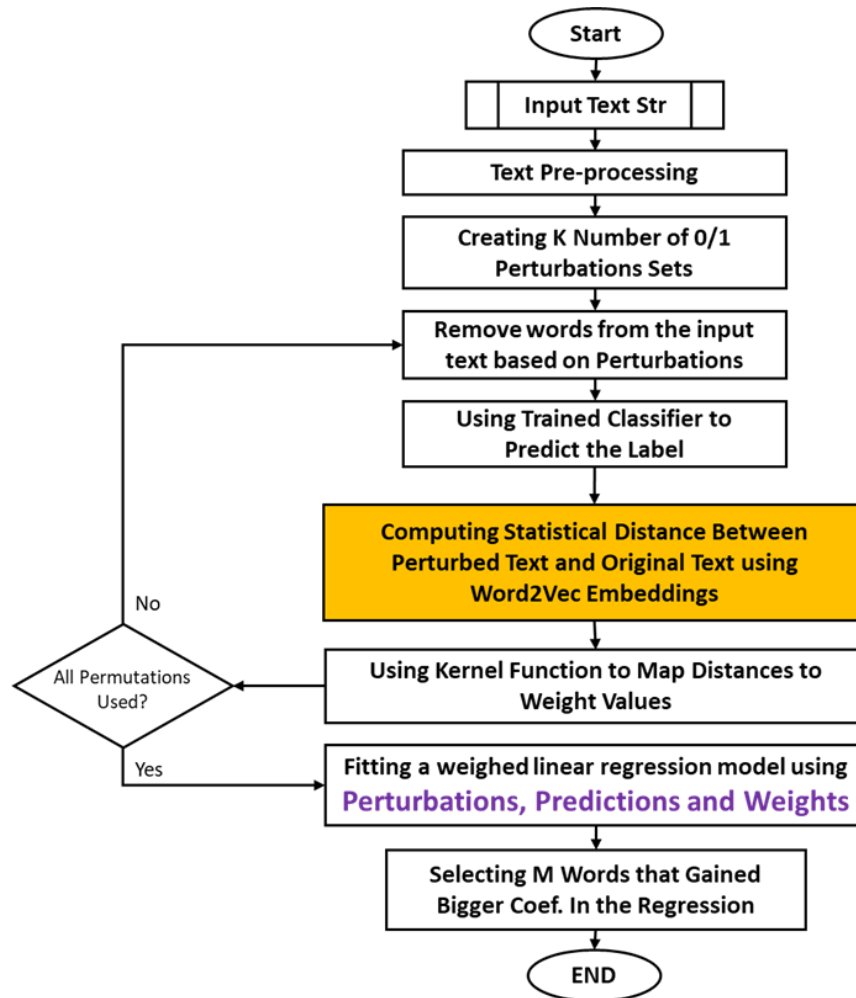
Figure 54 - SMILE flowchart for explaining text-based classification or regression

SMILE for Graph Datasets

The implementation of SMILE for graph datasets — or in other words, for explaining the decisions of graph neural networks — is a bit different. In graph datasets, we can have node, edge and feature explainability. The feature explainability part is similar to tabular datasets. However, for node and edge explainability, we need to find a way to calculate the Cumulative Distribution Functions (CDFs) of each graph to calculate statistical distance measures.

To do so, we have used the idea of Shimada, et al. [89] to create CDFs from given N x N adjacency matrices. The generated CDFs are based on the cumulative distribution of elements of the rth eigenvector. The original method used Kruglov distance to measure the statistical distance between two different graphs or sets of graphs. In SMILE, we have used the Wasserstein statistical distance measure instead.

Figure 55 illustrates the proposed overall procedure for generating the explainability for a given input graph and a given graph neural network. Each graph has two main parts including the feature matrix and adjacency matrix. In the middle of the figure, we have a perturbation-based mask generator which can generate random binary numbers with the length of features, edges or nodes. Depending on the type of explainability (node feature, edge, or node explainability), the perturbation-based masks can be used to, for example, remove a mask or remove a node from the original graph and generate a new set of perturbed graphs. Using the aforementioned method, the statistical distance between each perturbed graph and the original graph can be calculated. Moreover, the calculated statistical distance measures can be converted to weight values using a Kernel function. Also, the perturbed graphs are given to a graph neural network to get predictions. With triple sets of weights, predictions and perturbations, a weighted linear regression can be fitted as a surrogate model and its coefficients can be used for graph-based explainability.
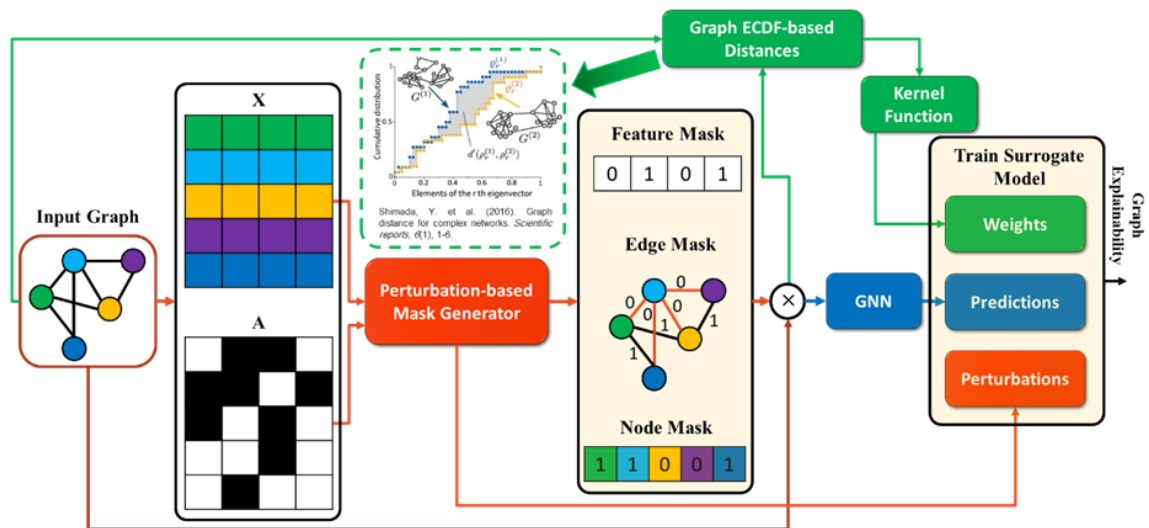
Figure 55 - SMILE block diagram for explaining graph neural networks' decisions

3.3 SAFETY OF MACHINE LEARNING IN SESAME

Machine learning already plays an important role in many robotic systems and is likely to increase in prominence. As explained earlier, this makes it vital to develop methods to assess and understand the safety of ML models where they are used as components in safety-critical systems like MRS. As such, dependability assurance of AI components in MRS is one of the areas where SESAME intends to push beyond the state of the art.

Several of the use cases in SESAME feature ML components already:

Confidentiality: Public Distribution

- The Locomotec use case, Disinfection of Hospital Environments using Robotic Teams, makes use of ML for the purposes of human detection. Given the hazardous nature of the UV-C light being projected by the robots for the purposes of disinfecting surfaces, it is important to be able to detect nearby humans in the environment in order to switch off the lamps and prevent inadvertent irradiation of unfortunate passersby.

- The Domaine Kox / Aero41 / LuxSense use case, Autonomous Pest Management in Viticulture, likewise makes use of object detection (for cars and people) to avoid having drones inadvertently spray them with fungicide. ML models like convolutional NNs, Support Vector Machines, and Random Forest classifiers are also used to evaluate the vines and detect problems such as fungal infections or pest infestations.

- The Cyprus Civil Defence / KIOS use case, Power Station Inspection using Autonomous Multi-Robot Systems, may also make use of ML models to help detect anomalies during routine inspection of the power station and to identify people or vehicles in hazardous environments during emergency situations.

Given the widespread use of ML models, frequently in critical scenarios (e.g. the person detection in the Locomotec use case), there are ample opportunities for application of ML assurance methods. Of the approaches discussed in this section, SafeML, Uncertainty Wrappers, and SMILE all offer the potential for significant advancements.

SafeML and Uncertainty Wrappers are both designed to provide confidence levels in the classifications or decisions of an ML model, though their precise approaches and other features vary. Along with DeepImportance from the University of York, they may be employed during testing phases during development, both to evaluate the training of the ML models themselves and also to assess performance during testing with preliminary real-world data. On the strength of these findings, the ML component may be redesigned or re-trained until a suitable level of confidence in the accuracy of the model is obtained. Further work in this area is expected as part of WP6.

Moreover, both techniques can be applied at runtime as a form of event monitor, providing warning of situations where confidence in ML decision making has dropped below an acceptable threshold. In this role, they can be employed as one sub-component of an EDDI (see Section 5). In particular, this helps address the problem of distributional shift and can warn of situations where the system is operating in a context or environment it was not designed for (or, perhaps more accurately, trained for).

Work has already begun to see how SafeML and our other ML assurance methods can best be applied to the Locomotec and power station inspection use cases, but so far only on an ad-hoc basis. Work is ongoing and the intention is to see how these techniques can be extended to runtime as part of the EDDI framework.

Explainability is also an important element of dependable ML and equally applicable to multi-robot systems, such as those in the use cases mentioned above. Explainability is particularly useful at design-time, since it can help evaluate and inform the results of training and testing of ML models. Knowing that accuracy is low during testing is one thing; knowing *why* it is low is another. With that understanding in place, corrective

measures can be taken to adjust the model or the wider system to compensate for any issues before they can become major problems at runtime.

Given its advantages over major approaches like LIME, SMILE seems to be an excellent candidate for application in SESAME. It could be of particular benefit when tuning person or object detection algorithms.

While explainability might be less directly applicable to autonomous systems at runtime, where the ability to explain every decision to a human is unlikely to be necessary, there will be cases where explainability remains important, especially in the case of any errors detected. Where e.g. SafeML or an Uncertainty Wrapper detects that confidence has fallen below the required threshold, explainability techniques like SMILE will prove valuable in explaining not just the behaviour of the ML component but also forming part of the behavioural log of the wider system.

4. THE CHALLENGE OF AUTONOMY AND OPENNESS

4.1 DEFINING THE PROBLEM

The preponderance of safety standards, techniques, and tools — many of which are described in Section 2 — might lead the naïve to believe that it is possible to guarantee that a system is safe. Alas, this is not the case. Even if every effort is made, every technique applied, and every standard followed to the letter, no system can ever be made 100% safe.

Part of this is simply due to human fallibility; even the most experienced expert, the most capable analyst, cannot envisage every possible problem — and even if they do, it may not be feasible (or more likely, not cost effective) to adjust the system design to prevent them. But the other major reason is that there is only so much that can be done at design time to ensure safety. No battle plan, as they say, survives contact with the enemy, and the same is true of safety: a system in operation will very likely encounter things at runtime that were either entirely unforeseen at design time or that may have been anticipated but the system was not designed to deal with.

Most obviously, random hardware failures can occur at any time in even the best designed, most well-maintained systems. To some extent, these can be planned for at design time, e.g. by selecting robust components, ensuring redundancy (such as k-out-of-n or primary/standby configurations), adding monitors, and specifying a rigorous maintenance schedule. But they cannot be prevented completely.

Furthermore, while a system can be meticulously designed, no designer or engineer has complete control over the environment in which that system may eventually operate. Environmental conditions are often inherently unpredictable and, all too often, completely uncontrollable. A drone may experience a wide variety of rapidly changing weather conditions, for example, and even the best weather forecast may turn out to be wrong.

These problems are only amplified when it comes to autonomous, multi-agent, cooperative systems like MRS. Ensuring the safety of a single system in a specific operating environment is difficult enough; attempting to ensure the safety of a robot expecting to function as part of a wider — as yet unknown — open system, operating in an unpredictable environment with other robots whose own safe behaviour cannot be guaranteed, is considerably harder. Needless to say, design-time efforts can only go so far when uncertainty, openness, and autonomy are such inherent characteristics of the systems in question.

It is for reasons such as these that techniques exist to help assure runtime safety. From fault detection and diagnosis to safety mechanisms and safe states to runtime safety assurance, such approaches help to resolve the uncertainty present at design time and enable systems to monitor and react to dynamic changes in their runtime dependability profiles.

Nevertheless, runtime safety assurance cannot be achieved in isolation. It is frequently more difficult than design time measures, requiring much greater overhead and infrastructure, and any runtime safety measures must themselves be subject to safety assurance procedures. As such, any runtime measures need to be built upon a solid

foundation of design time dependability assurance activities, and only those aspects which are necessary should be moved to runtime.

One of the first requirements of any runtime dependability solution is the acquisition of evidence. In general, this is a platform-dependent problem, reliant as it is on the precise nature and capabilities of the system in question. Evidence must be gathered from onboard sensors (or, in some cases, environmental sensors or those on other platforms in the vicinity) and while certain patterns and commonalities exist, for the most part implementation is specific to the platform.

Even so, it is possible to crudely describe the features of a generic event monitor. Typically it will need some form of buffer or memory (e.g. a ring buffer), recording past values in order to determine trends and filter out spurious or transient readings. It will need access to one or more sensors to do so and may need to transform the raw sensor data in some fashion. If it is designed to monitor for specific events, then the triggering conditions for those events also need to be defined. This may be a simple threshold value (e.g. exceeding a given maximum value), some form of Boolean or temporal expression, a mathematical equation (e.g. differentiation to obtain the rate of change), or something else.

Evidence and events alone are insufficient, however. For the system to respond appropriately, semantics are needed: what does the event mean for the system? This leads to the area of fault diagnosis. In the majority of cases, there is no one-to-one relationship between a sensor and a fault; simply knowing that there is no power to a given component is not enough to explain *why* there is no power. Yet without that additional knowledge, it may be impossible to decide on the correct course of action. Fault diagnosis uses a variety of methods to help determine causes and consequences of the events that have been detected, typically in the form of some kind of causal logic or model.

With a diagnosis, it is then possible to respond — whether to simply report the presence of the problem to a human operator or to automatically shift the system into a safe or degraded operating state.

In open multi-agent systems, however, this may not be enough. In such systems, tasks are achieved cooperatively, with different actors all dependent on each other. This requires a level of trust between agents: guarantees that they can achieve the goals that others depend on — and do so safely. As such, in an MRS or other multi-agent system, it may be necessary to warn other agents that such guarantees no longer exist or need to be renegotiated, e.g. to allow other agents to reorganise and attempt to compensate for the faulty agent.

By bringing together multiple levels of functionality (detection, diagnosis, response, and assurance) in a single, comprehensive framework, one that takes full advantage of rigorous design time dependability assurance processes, it may be possible to eliminate at least some of the uncertainty and achieve a degree of runtime dependability assurance that has never been possible before.

4.2 STATE OF THE ART: SAFETY OF MULTI-AGENT SYSTEMS AT RUNTIME

This section discusses some of the techniques used to address different problems in runtime dependability assurance, from runtime fault diagnosis to evidence monitoring to distributed safety assurance.

4.2.1 Runtime Fault Diagnosis

Design-time safety analysis is valuable in helping to identify potential failures and their possible causes. Causal models such as fault trees make these logical relationships clearer, and hazard and risk analyses help to assess the consequences, which is important in revising the design to help eliminate any problems identified.

It is impossible to correct all possible faults at design time, however, and it would be prohibitively expensive to even try. Rather than trying to remove all possible problems (those that can be foreseen, at least), it may be necessary to add mechanisms to mitigate potential faults, or even to leave them entirely unaddressed if their consequences are not too severe. Common design patterns like primary/standby, parallel vs serial, or k-out-of-n voter configurations can be used to enable a system to adapt to a failure and continue operating; in other cases, the system may fall back to a degraded state of operation or a non-functional safe state instead. In other cases, the system itself might not be capable of any form of adaptation but it may warn a human operator of a problem instead.

In all of these cases, however, the system must first detect and diagnose the faults that occur at runtime.

Fault *detection* is the first step. Monitoring systems observe the system parameters and detect when those parameters exceed normal thresholds and become abnormal. This requires additional system complexity, internal sensors, monitors, and appropriate processing units in addition to those components necessary for the system to carry out its primary function(s).

However, detecting abnormal parameters is not always enough for a system to know how to respond; in many cases, fault *diagnosis* is required to relate those abnormalities — or *symptoms* — to a probable cause. A rise in engine temperature may not be of concern in isolation, but when coupled with a drop in coolant pressure, diagnosis may hint at a failure of the cooling system, which will likely warrant some remedial action (such as switching the engine off or at least reducing its output). Replacing or repairing a component is also made much harder if one does not know which component is at fault. The difficulty in fault diagnosis is that failures seldom have single, mutually exclusive symptoms, and thus it is typically not possible to determine a single responsible fault from a given symptom.

Causal models like fault trees (and to a lesser degree FMEAs) can help with this because they relate combinations of failures to one or more effects. Intermediate nodes of a fault tree may also represent symptoms; "drop in coolant pressure", for example, could be a node with multiple children, each representing different possible causes. The difficulty would then be in choosing which of the possible causes is the one responsible — and that assumes the model itself was correct and complete in the first place.

It is for these reasons that many different fault diagnosis approaches have been developed over the years. Typically, these approaches function by predicting how the

system *should* operate in a given state and comparing this prediction to how the system is *actually* operating at present. In all cases, however, in-depth knowledge is required about the system, how it operates, and the potential causes and effects of failures it may experience.

Broadly speaking, there are three general categories of runtime fault diagnosis techniques:

- *Rule-based diagnosis* operates on the basis of a collection of logical "if-then" style rules of causality: "if condition X exists, then Y". These rules may be deductive and used to determine causes ("if parameter X exceeds value Y, then failure Z has occurred"); they may be inductive and used to predict consequences ("if event X occurs, then Y will happen next"); or they may be some combination thereof.

- In *Model-based diagnosis,* instead of the knowledge of the system failure behaviour being captured in a series of logical rules, it is represented instead by a model (often, but not always, a derivative of a design-time model). The model may represent normal functioning of the system, e.g. in a kind of simulation, and then diagnosis occurs when the simulated behaviour of the model deviates from what is expected; alternatively, the model may represent the failure behaviour of the system, modelling causes and effects of faults only. Combinations of the two are also possible, though tending to favour a focus on one or the other.

- Finally, *data-driven diagnosis* focuses on long-term observation of historical trends and deviation from them. ML-based approaches to fault diagnosis may also be considered to fall into this category, at least in some cases [90].

This section will provide a brief overview of these categories and some of the approaches within them.

4.2.1.1 *Rule-based diagnosis*

Perhaps the oldest form of diagnosis approach, rule-based diagnosis operates on the basis of simple cause and effect: if we know condition X exists and we know that fault F is the only cause of condition X, then we can also say that fault F has occurred. Obviously, this becomes more difficult when multiple potential causes exist or when multiple conditions or events have occurred.

Rule-based diagnosis has its roots in the medical domain, e.g. the MYCIN expert system [91]. Expert systems are designed to emulate the decision making of human experts (hence the name). They typically contain a knowledge base containing declarative facts and a series of production rules, usually if-then style, to enable reasoning over the knowledge. In the case of fault diagnosis, these rules are typically causal rules that describe the relationships between system components or states, their failures, and their effects.

Both "forward" and "backward" chaining of rules is possible. In the former case, system parameters are monitored for certain conditions that trigger the activation of particular rules. Once activated, a rule's effects are considered to be new facts about the system, which may in turn trigger new rules at a later point. If a rule's effect is considered to be

a system failure, then the chain of activated rules captures the sequence of events that caused it.

In backward chaining, hypotheses are tested instead. A hypothesis is typically a malfunction or system failure. Testing the hypothesis is achieved by working backwards to see if the hypothesis is the outcome of any of the rules; if one is found, then it is provisionally set as true and its preconditions or triggers are treated as new hypotheses to be tested in turn. This continues until either the hypothesis is proved false (e.g. because no preceding rule can be found, or because its triggers are unsupported by recorded system observations) or the hypothesis is verified (because all preceding rules match the recorded system observations).

It is also possible for both to be used together, e.g. as in the REACTOR expert system designed for use with nuclear reactors [92].

One limitation with this simple if-then style of rule is the inability to perform any kind of serious temporal reasoning, as in cases where long-term trends or particular sequences of events are important. This can be addressed with more complex rules that feature temporal logic or similar time-based operators.

Another problem, perhaps more fundamental, is that of uncertainty. The relationship between the measurement of a particular system parameter and a fault or failure is often not absolute; there can be many potential causes, some more likely than others, and there may not necessarily be a causal relationship at all if the reading is anomalous somehow (or if the knowledge base is incorrect or incomplete). In some such cases, "certainty factors" are applied as a measure of uncertainty or even probability for different diagnoses (as with the MYCIN system).

Despite these disadvantages, there are advantages to rule-based systems too. The separation of knowledge (i.e., facts) and reasoning (rules) makes the approach relatively generic, helping to explain why such systems have been applied across a wide range of domains. The ability to use both inductive and deductive reasoning is also of benefit, allowing diagnoses to be confirmed using multiple methods. Their flexibility also means that rule-based systems can relatively easily be combined with other approaches, e.g. the QUINCE system combines a neural network for symptom detection and a rule-based system to diagnose the causes of those symptoms [93].

However, rule-based systems tend to struggle when applied to larger, more complex systems. Inconsistencies emerge that can be impossible to resolve, at least without some degree of uncertainty, and it can be much harder to guarantee completeness of the knowledge base and rule set.

4.2.1.2 *Model-based diagnosis*

Model-based diagnosis approaches still rely on expert knowledge, like rule-based approaches, but instead represent this knowledge in a separate model of the system. This provides a deeper understanding of the system in question, capturing more of its essence than a simple set of facts and rules. For example, the model may be architectural, thus encapsulating the relationships of the various components to each other; or it may be behavioural, allowing some degree of simulation or at least better representation of the various system states and actions. Diagnosis is less about

attempting to match specific rule patterns against a set of data recorded in a database, but rather about following causal relationships as they propagate through the model of the system.

As mentioned, models may primarily represent either the normal functional behaviour of the system or the abnormal failure behaviour, though combinations of both are also possible. Abnormal behaviour models are typically fault propagation models while the normal models are generally simulation-based models. A kind of hybrid model category is that of causal process graphs, which describe interactions between system processes over the course of a disturbance (e.g. a failure, though not necessarily) rather than fault propagation directly. Various examples of these types are briefly described in the table below:

Table 4 - Summary of model-based fault diagnosis approaches

Name	Type	Description
Cause Consequence Diagrams [94]	Fault propagation	One of the earliest forms, developed for use with nuclear reactors. They represent causal relationships using a network of cause and consequence trees; the former describe the causes of hazards in terms of combinations of failures, while the latter begin with a hazardous event and describe how other conditions or mitigating actions can lead to safe or unsafe system states.
Fault Trees	Fault propagation	Fault trees can be used directly for fault diagnosis, as long as there also exists some form of monitoring/fault detection system. Different patterns of sensor readings are associated with different trees (or nodes within a tree), and when such patterns are detected, the associated nodes are assumed to be 'true'. One can then work up or down the tree as required to determine consequences and causes.
Diagnostic Decision Trees [95] [96]	Fault propagation	DDTs are binary trees; each node represents a true/false question to narrow down a diagnosis. Can be constructed from fault trees or trained on historical system telemetry. The latter also enables them to adapt to faults that were unknown or unpredicted at design time.
Digraphs [97][98]	Causal process	An example of causal process graphs, these model the effects that changes in process parameters can have on other process parameters. For example, a dependent parameter may have a positive relationship with a preceding parameter (i.e., if the first parameter increases, so does the dependent one) or a negative relationship. Diagnosis is achieved by tracing the propagation of value deviation through the graph.
Logic flowgraph [99][100]	Causal process	An extension of digraphs to improve expressiveness. Able to represent continuous and binary state variables as well as logic gates and various conditions. Varieties of flowgraphs exist, e.g. dynamic flowgraphs.
Goal Tree Success Tree [101]	Normal behaviour	GTSTs are logical trees similar to fault trees containing AND and OR gates that decompose high-level safety

Name	Type	Description
		objectives into sub-goals and the conditions necessary to achieve those goals. Their flexibility means they have often been combined with other approaches such as inference engines or probabilistic techniques.
General Diagnostics Engine [102]	Normal behaviour	An assumption-based truth maintenance system; uses a predictive engine that propagates both values and underlying assumptions. When discrepancies are detected, a set of suspects is generated. Further conflicts reduce the set of suspects — only those suspects that remain can explain all detected deviations from expected behaviour.
QSIM [103]	Normal behaviour	A qualitative simulation tool which simulates the system as a set of qualitative constraints. QSIM predicts qualitative values (e.g. increasing, stable, decreasing) and ranges for each monitored parameter. When deviations between predictions and actual readings are detected, hypotheses are generated and a new QSIM model created to test them. Hypotheses are discarded if simulation does not match the observed behaviour.
Rodelica & RODON [104][105]	Normal behaviour	Rodelica is a declarative, equation-based, object-oriented language derived from Modelica and intended for fault diagnosis. Allows better numerical representation than QSIM with intervals, ranges, and constraints etc. Once modelled, the Rodelica system model can then be processed by the RODON reasoning engine for diagnosis, first detecting deviations between predicted and observed behaviour, then generating and eliminating hypotheses.

4.2.1.3 *Data-driven diagnosis*

In data-driven approaches, a predictive model of the system behaviour is generated from empirical data collected from the system operation (whether during testing or after deployment). Generally speaking, this predictive model encapsulates the relationships between the input and output parameters of the system.

Examples of data-driven approaches include statistical process monitoring approaches, qualitative trend monitoring, and neural networks.

- Statistical process monitors are either univariate, in which a single fitting function is employed to relate one dependent variable (e.g. an output) to multiple independent variables (e.g. inputs), or multivariate, in which statistical methods and historical data are combined to create more accurate predictive models for both dependent and independent variables. An example of the latter is CART, or classification & regression trees [106], which uses binary partitioning to enable analysis of large datasets.

- Qualitative trend analysis generally consists of two steps: identification of trends on the basis of system measurements, then interpretation of those trends. Trends are qualitative, e.g. increasing, stable, decreasing etc. Deviations from the expected trends are then detected and diagnosed.

- Neural networks, as already explained in Section 4, are ML models that are trained on an established dataset. For the purposes of fault diagnosis, they are typically trained to differentiate between normal behaviour and abnormalities.

As with other forms of ML, neural networks are an increasingly popular choice for fault detection (and sometimes fault diagnosis) [90]. Statistical approaches are found to be fast and effective for fault detection, but less so for diagnosis, while ML and other pattern recognition techniques tend to be much better at diagnosis but slower at detection.

4.2.2 Dynamic Risk Assessment

Runtime fault diagnosis and the respective approaches mentioned in the previous section focus on detecting causes for observed safety-related errors or failures within the multi-agent system. However, whether a particular error or failure is safety-critical and poses an actual risk depends on the current operational situation the system finds itself in during runtime. For instance, if a planned trajectory of a drone differs from specification due to a fault in the system, a collision with other dynamic or static objects may only occur if those objects are present in the current operational situation. Thus, hazardous events and their associated risk are always conditioned on the operational situation.

Current safety standards address this issue by designing the system in a way that safety goals and their integrity will lead to safe behaviour in all operational situations based on worst-case assumptions. Put differently, the system always expects the worst-case situation to happen. This approach indeed leads to safe behaviour and cost-effective safety assurance, as the situation space needs to be analysed only for the identification of worst-case situations. In reality, however, worst-case situations rarely occur and in the majority of operational situations, the risk is low. This results in situations where diagnosed faults lead to the execution of minimum risk manoeuvres or to a transition to the safe state, although the actual risk would not demand it. Consequentially, if we monitor the presence of risky/non-risky operational situations, we can increase MAS performance by a) being able to tolerate certain faults and failures if their associated risk is low in the current situation and b) actively reducing particular risk parameters with tactical decisions, where severity, controllability or the operational situation itself is changed.

Dynamic risk assessment (DRA) techniques thus treat the MAS or a constituent system as a black box and provide means to analyse the consequences of MAS behaviour deviations on the risk in the current operational situation. Note that DRA can be both performed for constituent systems and on the MAS as a whole. The difference lies in the definition of hazards resulting from deviating behaviour. Such hazards can be analysed for MAS collaborative behaviour or single system behaviour.

For autonomous systems, in particular in the automotive domain, DRA techniques have been applied to address the trade-off between performance and safety risks. The existing approaches can be classified broadly in three categories.

1. DRA is incorporated into motion and trajectory prediction frameworks by specifying behavioural or kinematic constraints that are *input* to the trajectory planner. Such constraints come either in the shape of an augmented map, which

treats risk as occupied spaces the planner has to avoid, or they come in as boundary conditions the trajectory planner needs to respect, e.g. speed less than a particular threshold. The result of these approaches is usually a planned trajectory that is claimed to be safe. Representatives of this class are [107] and [108].

2. DRA is performed during the online verification of an already planned yet potentially unsafe trajectory. Thus, this class of approaches uses the *output* of the trajectory planner and checks afterwards whether it may lead to unsafe behaviour in the current operational situation. If the safety criteria of the verifier are violated, a transition to a safe state or degraded operation mode is triggered. Representatives of this class are [109] and [110].

3. DRA is not performed in direct relation with a planned trajectory, but instead monitors influence variables that enable distinction between the presence/absence of a hazardous event and influence its criticality through risk parameters such as exposure, external controllability and accident severity. Such DRA approaches are closely related to design-time hazard and risk assessment models, as the monitors capture the observable variability in these models. The output of these approaches is usually a list of safety goals that are relevant in the *current* operational situation along with the dynamic risk rating of these safety goals based on a dynamic assessment of the risk parameters. Representatives of this class are [111][112][113] [114] and [115].

Although different architectures for incorporating DRA into autonomous systems exist, all of them need to decide which particular risk-influencing situation features are to be observed in the present, project the evolution of these features into the future by using a set of assumptions, and rate the risk of the projected future situation (Figure 56). For each of these DRA sub tasks, respective design time engineering activities are required.
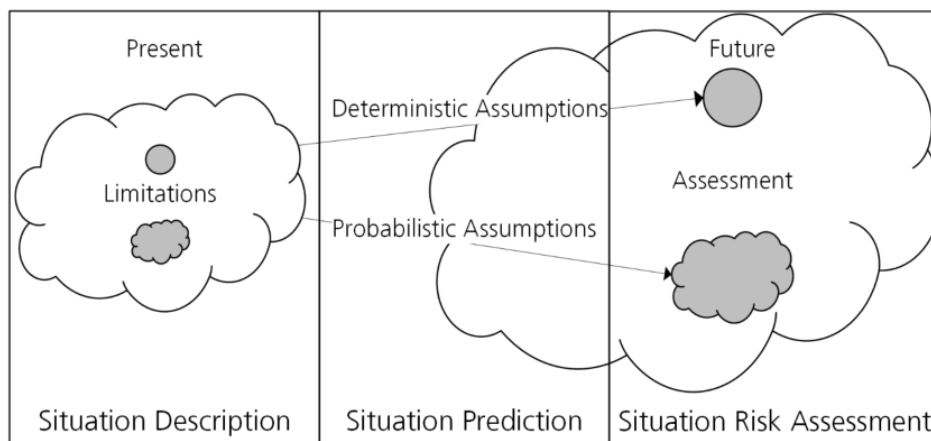
Figure 56 - Dynamic Risk Assessment Conceptual Overview [115]

Situation Description demands a risk-driven situation space decomposition. This task is already performed during hazard analysis and risk assessment (HARA), where those operational situations are sought for indicating the highest risk. In contrast, DRA needs to identify risk-variable operational situations and thus extend current design-time HARA activities with a more fine-grained situation space analysis. This analysis requires an understanding of the intended operational domain and may consider

situation features of dynamic and static objects, environmental conditions as well as interactions between actors.

Based on a selection of situation features being risk-relevant in the intended operational domain, *Situation Prediction* predicts the future state of the selected situation features based on different assumptions and prediction models. For instance, for autonomous vehicles, such behaviour prediction models may assume traffic rule adherence or constraints on the kinematic state such as constant speed or acceleration. The assumptions can be either deterministic or probabilistic.

Having a predicted state of situation features, finally their relation to risk needs to be established during *Situation Risk Assessment*. For this purpose, risk metrics are used. Such risk metrics are typically highly domain- and even application-specific. Although on a high level, risk metrics always combine the likelihood of an unwanted event with its impact severity, the concrete relation between the predicted situation features and those risk parameters needs to be explicitly modelled. Examples for DRA metrics are Time-To-Critical-Collision-Probability (TTTCP) [116] or Deviation-From-Expectation [117].

Further information on situation prediction techniques and concrete risk metrics can be found in relevant literature reviews (see [115], [118], [119], and [120]).

Up to this point during design-time, situation features have been selected based on a risk-driven analysis of the operational domain (possibly through a HARA analysis with a more fine-grained situation analysis), prediction models for those situation features have been selected based on a set of deterministic or probabilistic assumptions and domain-specific risk metrics have been selected to quantify the situation risk based on the predicted situation feature state.

In order to enable machines to perform DRA, modelling formalisms are required to technically capture the relationship between situation features and risks. For this purpose, different classes of models can be used, which are model-based (e.g. structural equation models), rule-based (Boolean models), generative (Gaussian Mixture Models, dynamic Bayesian networks, Hidden Markov Models), discriminative (Decision Tree, Random Forest, Support Vector Machines) or deep learning-based (Multi-Layer Perceptron, Convolution Neural Network – CNN, Recurrent Neural Network – RNN, Long-Short-Term-Memory Network – LSTM). Some of them are developed purely based on expert knowledge, some of them can be adapted to new operational domains with machine learning techniques. Depending on the concrete modelling formalism selected, support for deterministic or probabilistic assumptions may be given as well as the consideration of uncertainties during feature perception is possible.

At Fraunhofer IESE, DRA for autonomous systems was considered in a recent dissertation [115]. The work contributed a conceptual taxonomy of DRA and provided a concrete application of DRA using concrete instances of controllability and severity metrics to rate collision risks for automated driving. In order to be applicable in many different operational situations, one explicit requirement in [115] is that the risk metric is supposed to be situation-agnostic, i.e. it is only allowed to use the kinematic state of dynamic actors. Situation-specific features such as road structure, lighting and weather conditions, traffic rules, actor interactions were consequentially not considered. To account for the fact that risk is often influenced by exactly those situation-specific

features, the work in [115] was extended to the *Situation-Aware Dynamic Risk Assessment* (SINADRA) framework [121]. The approach uses Bayesian networks as a modelling formalism and explicitly considers the mentioned situation-specific features as risk influences. A proof-of-concept and tool implementation of the approach is presented in [122]. The design-time method for building situation prediction models with machine learning techniques is presented in [123].

In summary, DRA enables a system or MAS to automatically assess the risk of a (malfunctioning) behaviour in the *current* operational situation. Based on this dynamic risk estimate, the unavailability of safety capabilities can be potentially tolerated in low-risk situations or tactical decisions can be enacted to actively lower the risk to an acceptable level. For this purpose, situation features need to be selected, their future state predicted, and a risk metric be calculated on the future state to get a qualitative or quantitative risk rating. Apart from a fully formal inference mechanism at runtime, quite some effort needs to be put into the safety engineering of the DRA mechanism for a particular use case in a particular domain. To come up with a meaningful trustworthiness argument for a DRA monitor, the selection of considered situation features needs to be grounded in a systematic HARA, the validity of the assumptions behind the situation prediction needs to be demonstrated and the adequacy of selected risk metrics for the concretely considered risks needs to be argued.

4.2.3 Dynamic Safety Concepts: Conditional Safety Certificates

One challenge in ensuring the safety of cooperative automated systems is to deal with uncertainties and unknowns with respect to the cooperation partners. In other words, one might not know what kind of guarantees come along with a certain information or service of a 3rd party system. Still, it is clearly beneficial to utilize such information and services for safety-critical applications, because there is such a huge potential in terms of new applications, improved performance and also improved safety. As an example for the latter, consider systems warning other systems regarding obstacles, systems orchestrating at a crossroad, and so on. Unfortunately, the lack of knowledge regarding external services and their safety properties typically leads to worst case assumptions, which in turn severely constrain performance, or even lead to the decision not to use external services or information at all. A straightforward solution to this problem can be to enable constituent systems of a MAS to explicitly negotiate their safety-related properties at runtime. This implies that we establish runtime safety models describing these properties for a (constituent) system and standardize a protocol for their negotiation.

Conditional Safety Certificates (ConSerts) [124][125] is an approach to do exactly that. ConSerts operate on the level of safety requirements. They are specified at development time based on a sound and comprehensive safety argumentation (e.g. an assurance case). They conditionally certify that the associated system will provide specific safety guarantees. Conditions are related to the fulfillment of specific demands regarding the environment and the fulfillment of the conditions is checked during runtime. In the same way as "static" certificates, ConSerts shall be issued by safety experts, independent organizations, or authorized bodies (depending on the respective application domain) after a stringent manual check of the safety argument. To this end, it is mandatory to prove all claims regarding the fulfillment of provided safety guarantees by means of suitable evidence and to provide adequate documentation of the overall argument – including the external demands and their implications.

There are some significant differences between ConSerts and static certificates that are owed to the nature of open cooperative systems: A ConSert is not static but conditional; it therefore comprises a number of variants that are conditional with respect to the (dynamic) fulfillment of demands; and it must be available in an executable (and composable) form at runtime.

Conditions within a ConSert manifest in relations between potentially guaranteed safety requirements ("guarantees") and the corresponding demanded safety requirements ("demands"). Demands always represent safety requirements relating to the environment of a component, which cannot be verified at development time because the required information is not available yet. These demands might directly relate to required functionalities from other components.

On the other hand, evidence can be required beyond that, since safety is not a purely modular property and it cannot be assumed that a composition of safe components is automatically safe. To this end, ConSerts support the concept of so-called Runtime Evidences (RtE) as an additional operand of the conditions. RtEs are a very flexible concept. In principle, any runtime analysis providing a Boolean result can be used. RtEs might relate to properties of the composition or to any context information, e.g. a physical phenomenon such as the temperature of the environment that is safety relevant. Other RtE require dynamic negotiation between components.

In any case, ConSerts must be available at runtime in a machine-readable representation and the systems need to possess mechanisms for composing and analyzing runtime models. Based thereon, a valid safety certificate for the over-all system of systems can be established. ConSerts are a relatively lightweight runtime safety approach and they are not far from traditional safety engineering. The main difference being that unknown context is structured into a series of foreseen variants, which are then specified in a runtime model to be resolved at runtime.
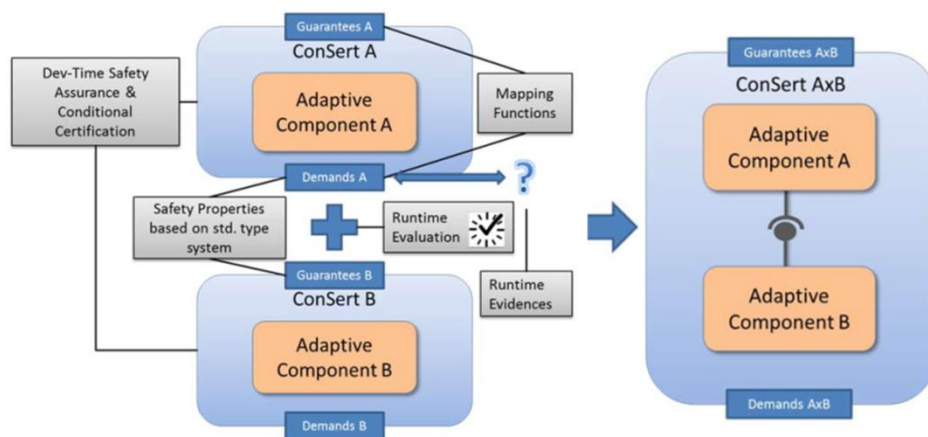
Figure 57 - ConSert Composition Conceptual Overview

ConSerts capture modular, conditional, and pre-assured safety concepts that enable dynamic reconfiguration of the underlying system (of systems) based on observed changes of the operational context (Figure 57). This relates to runtime fault diagnosis and dynamic risk assessment in the following way: Dynamic risk assessment dynamically determines MAS or constituent system safety goals along with a dynamic

rating of the safety goal integrity derived from the risk estimate. This represents the set of top-level safety goals that need to be fulfilled in the *current* operational situation.

Having these dynamic safety goals, the system has to have a safety concept in place that is able to address those dynamic safety goals. Since different safety concepts are conceivable to address different safety goals and variable integrity demands, the need for specifying variable safety concepts arises. Such a safety concept is operationalized by so-called safety capabilities that the system uses during operation. Examples of such capabilities are fault diagnosis mechanisms to establish runtime evidences, fault tolerance mechanisms to achieve safety guarantees or mechanisms for the transition into a safe state. As such, ConSerts are formal representations of dynamic safety concepts expressing the dynamic parts of a safety concept necessary to distinguish between the availability of different safety capabilities and in consequence different safety guarantees that a system can give at runtime. Runtime diagnosis techniques as introduced in Section 4.2.1 are related to ConSerts in that they provide the basis for observing runtime evidences that represent the leaf nodes in a ConSert tree.

Figure 58 highlights the relationship between a design-time safety concept and its transformation into a Boolean model representing the dynamic parts of the safety concept.
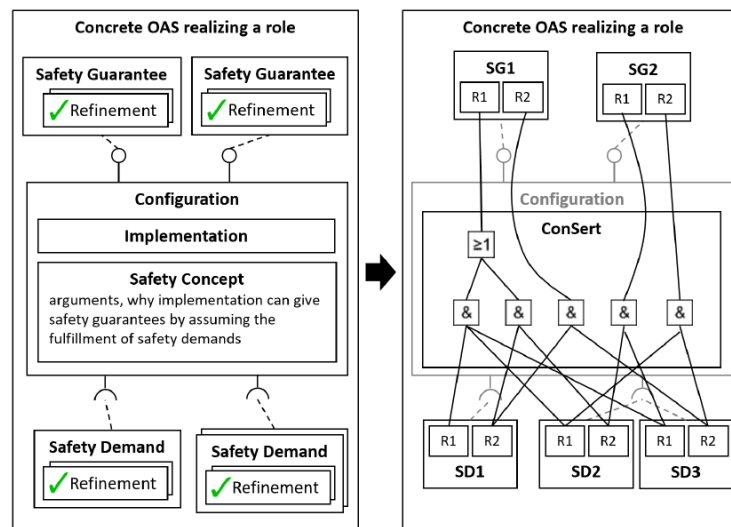
Figure 58 - Relation between safety concept and ConSert for an open adaptive system

4.2.3.1 ConSert Operationalization

In order to operationalize ConSerts for an open adaptive system (OAS), an essential precondition is that the used architectural modeling approach supports the abstraction from concrete system interfaces in terms of a formal description of both functional and nonfunctional properties the systems provide to their environment. Service-oriented architectures support the required concepts in a powerful and efficient way by establishing provided and required services for systems that represent well-defined interfaces for provided and required black-box functionalities. Although the application of ConSerts is in general not limited to service-oriented architectures, the ConSert approach assumes systems to be architecturally decomposed with services due to the advantages of SOA.

Figure 59 depicts the ConSerts metamodel describing the relations between the elements used for the architectural composition of OAS by means of services (green area), the elements used for formalizing the safety properties of these services (red area) and the elements for the composition of safety contracts for OAS in the shape of ConSerts (blue area). A more detailed description of the meta-model can be found in [126].
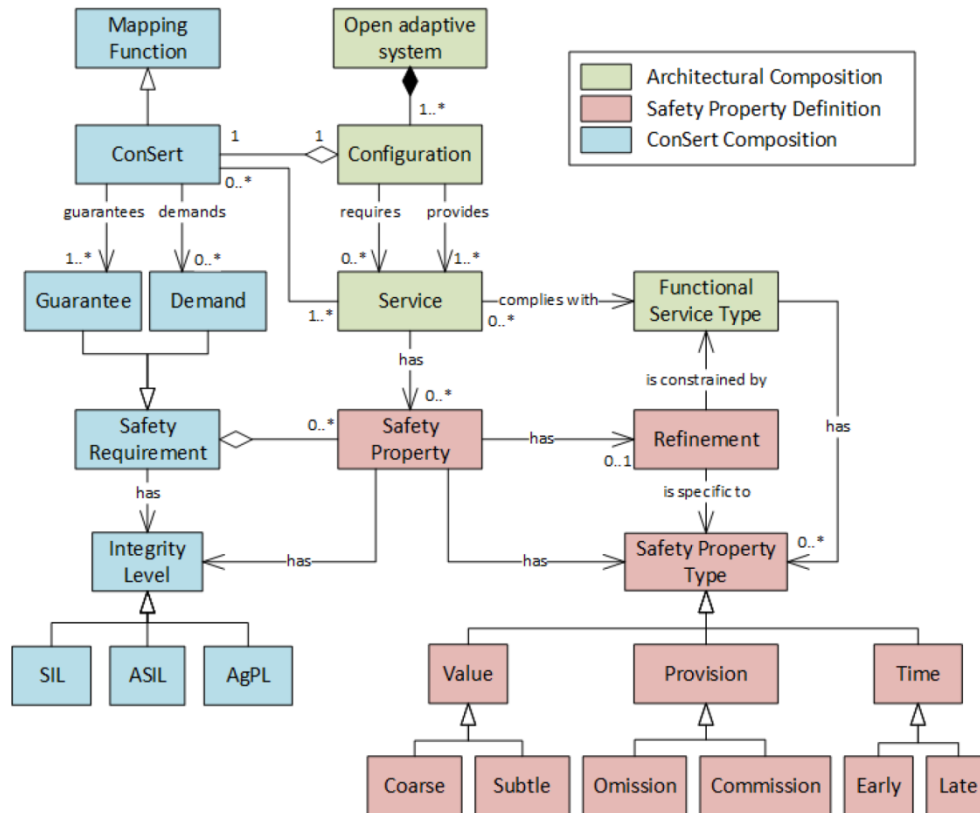
Figure 59 - ConSerts Metamodel

For the application of ConSerts, it is sufficient to describe the functionality of an *open adaptive system* in terms of a set of *configurations*, which provide certain functional services to the environment and require other functional services to deliver their provided functionality. Configurations express predefined functional variants of OAS that result from their ability to adapt themselves to different runtime contexts mainly due to dynamic service availability of other collaborating OAS. A provided or required *service* as such first and foremost describes a functional purpose. This could be for example the provision of a speed value or an interface for receiving remote control commands from other systems. *Functional Service Types* serve in this respect to capture the detailed and formalized description of the services' functional aspects, which include properties like data types (e.g. numeric or compositional), units, valid value ranges, value resolution or provision frequency (continuous or event-based).

In this way, each concretely instantiated *service* complies with exactly one functional service type. Note that the explicit separation of functional service types from their usage in concrete services allows to build domain-specific repositories of reusable service types. In addition to so-called *basic* functional service types, which represent the building blocks for the communication among OAS, there exists also a special service

type referred to as *application service*. Rather than representing an interface between OAS, the application service provides the overall collaboration's functionality to human beings and thus generates the desired business value of the collaboration.

The ConSert approach enriches functional service types with safety-related information. In this way, a concrete service does not only comply with a functional service type anymore, but has associated *safety properties* that themselves comply with a *safety property type*. Safety property types describe possible failure modes of functional services or more precisely, they describe deviations from a specified functional behavior. Figure 59 shows a commonly used classification for safety property types that distinguishes between value, provision and timing failures. In the same way as functional service types, safety property types could be organized in a domain-specific type system and assigned to specific safety properties. So far, a concrete service consists of its functional behavior and safety properties that define possibly occurring failure modes for the functional behavior.

With respect to the speed provision service example, one possible safety property type is "Value too high". However, this information as such is not sufficient for describing a service failure in its entirety, because whether an excessively high speed value is critical for a collaboration may differ from scenario to scenario. Thus, when instantiating a safety property from a safety property type, it needs to be refined with respect to a specific collaboration scenario. This refinement has to specify quantitatively when a certain deviation from the functional specification is considered to be a service failure. In addition, it can contain information on the possible consequences of the failure within the specific scenario. The knowledge on potential consequences of service failures is a precondition for assessing the risk of behavioral deviations expressed as refined safety properties.

In summary, refined safety properties offer the possibility to specify guarantees or demands that assure with a specific level of confidence that certain safety requirements will be satisfied for a specific service. Put differently, the safety requirements assure with a certain confidence that behavior deviations of services do not exceed the specified boundaries. From a safety point of view, the collaboration partner providing the application service also has the special responsibility of providing the safety guarantees associated to the application service. These guarantees are direct translations of the collaboration's safety goals.

Having defined concrete services and their associated safety properties, the final step is the composition of *ConSerts* for the existing configurations of the OAS through mapping functions. These mapping functions relate guaranteed safety properties of provided services to demanded safety properties of required services or runtime evidences. By using Boolean functions for the mappings, dependencies between guarantees and demands can be expressed by common OR and AND relations. For each provided guarantee of each provided service, there shall be a separate ConSert Tree (CST), represented by a Boolean function $f : B^k \rightarrow B$. Inputs of the mapping function are k Boolean variables, each representing a demanded set of safety properties belonging to a required service. Such a Boolean variable is true if the demanded safety properties are actually met at runtime. Thus, if all Boolean variables that are logically related to a specific guarantee render true, the safety properties of that guarantee hold

for the provided service. In a nutshell, ConSerts consist of multiple CSTs, which model the conditions for the guarantee variants of each provided service.

4.2.3.2 *ConSert Engineering*

The ConSert case studies have shown that it is useful to split the engineering activities enabling the composition of ConSerts in domain-level and system-level activities [127].

Domain-level engineering

The observation that collaboration scenarios within specific domains often use similar basic functional service types which also have similar safety property types, led to the idea to create a domain-specific repository for these elements, the so-called Safety Domain Model (SDM), which is shown in Figure 60. Although concrete OAS are not completely known at design time, basic functional service types together with their safety property types can be assumed to be required for several collaboration scenarios within a domain. To that end, the SDM captures this knowledge so that it can be easily reused in the system-level engineering phase, when the ConSerts for concrete OAS have to be created.

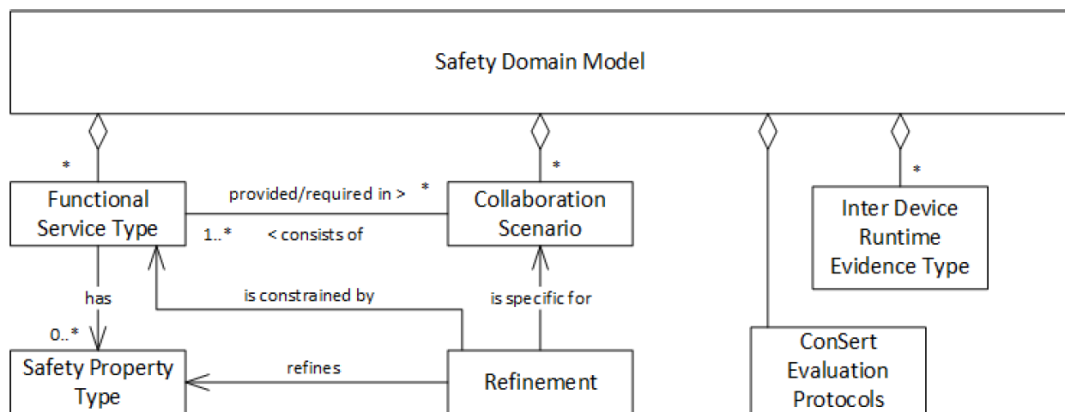**Figure 60 - Safety Domain Model**

Collaboration scenarios can be thought of as typical interaction patterns that occur within a specific domain. Such interaction patterns include participant roles, structural information about the role interfaces in the shape of functional service types as well as interaction schemes. A classification of functional service types is given in [128] and depicted in Figure 61.
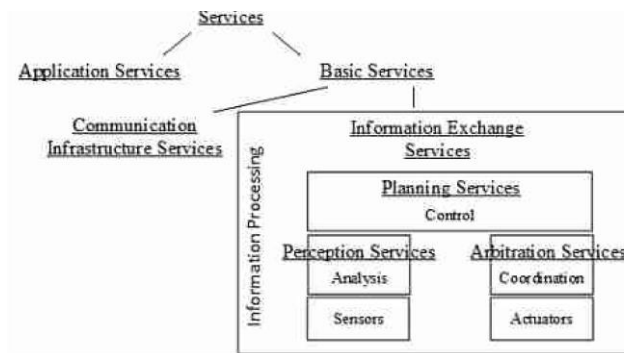
Figure 61 - Classification of functional service types of open adaptive systems

Assuming that a comprehensive set of basic service type specifications exists for an application domain, safety property types can be derived by the application of suitable safety analysis techniques. [127] recommended deductive *hazard operability studies* (HAZOP) for the safety analysis of service types, since the causal model of HAZOP matches the idea that failures occurring in concrete OAS will eventually manifest themselves at the service-level and will therefore have consequences on a given collaboration. In addition to the identification of functional service types and safety property types, *inter-device runtime evidences* have to be standardized on the domain-level as well, since their evaluation involves the interaction between multiple participant devices/roles. Thus, the interaction protocols for this evaluation need to be specified on the domain-level. Furthermore, domain-specific communication protocol stacks in particular have to be considered for the technical realization of ConSert composition and evaluation and thus ConSert evaluation protocols have to be standardized on the domain-level as well.

Despite the advantages provided through an existent SDM, there exist some challenges for its creation, too:

1. The creation of the SDM includes a high amount of preparation work, during which no direct benefit can be generated.

2. Domain engineering relies on the background knowledge of experts rather than on real systems and thus the standardized elements might not fit concrete required usage scenarios.

3. Standardization for an entire domain needs common agreements within a majority of companies of that domain. Such agreements can be negatively influenced by communication challenges and competition.

4. A suitable level of abstraction needs to be found for the SDM to leave enough space for solution flexibility in concrete OAS realizations.

However, having the identified challenges of domain-level engineering in mind, a diligently maintained SDM can be highly beneficial for the industry in the long run. The fact that technology research and innovation as well as experience are key enablers for new collaboration scenarios, also means that the SDM will be subject to a continuous evolution.

System-level engineering

Having the safety domain model definition as a basis, it can be used for the creation of ConSerts for concrete OAS that should later on participate in collaboration scenarios. Figure 62 shows the relation between domain- and system-level activities through the SDM.
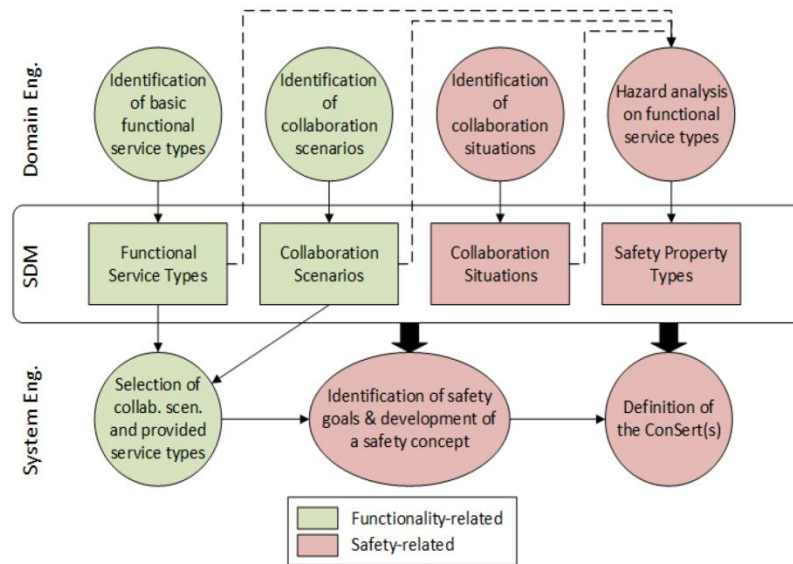
Figure 62 - Engineering activities and the Safety Domain Model (SDM)

When a new OAS should be developed or extended to support a specific collaboration, the first step is to explore the safety domain model for available information on the desired collaboration scenario and its variants. The selection of variants that should be realized in the OAS under development directly guides the determination of the functional services that have to be provided by the OAS. Selecting several variants implies the realization of multiple configurations where each configuration describes exactly one supported collaboration variant.

Based on configurations describing provided and required services, safety goals and associated safety guarantees have to be determined, which can be created by means of traditional safety engineering techniques like *hazard and risk assessment* (HARA) techniques. Obviously, the selection of multiple collaboration variants leads to higher development costs due to the provision of higher safety guarantees. However, the business benefits that can be leveraged from a better collaboration performance in case of higher provided guarantees, can make this investment attractive. Thus, the main engineering goal is to provide sufficient guarantees for the provided services of the collaboration variants which have been selected according to business decisions.

Once the safety goals and their associated safety guarantees have been derived, a safety concept has to be developed that provides an argumentation ensuring that the safety goals are satisfied. The creation of a safety concept starts with a safety analysis of the OAS under development which explores the causes that can lead to a violation of the safety goals. The safety property types defined in the SDM can act as a starting point for the system-level safety analysis. The safety goal violation causes can either be located internally in the OAS under development or can be caused externally by required

services or runtime evidences. In any case, the safety concept should address the identified causes with appropriate countermeasures or additional safety demands. The formalization of the relation between safety guarantees and safety demands (=the ConSert(s)) of the OAS under development can be carried out based on the contents of the safety concept.

The final step is the actual certification of the OAS under development: After an extensive examination of the documentation of the safety goals, the safety concept and the derived ConSerts, an authorized certification body will issue the required safety certificate. The required documentation could be for instance organized in a safety case, while the actual structure of the safety case might be dictated by domain-specific safety standards.

4.2.3.3 *Recent extensions and applications of ConSerts*

A case study of ConSerts for truck platooning has been performed in [129]. In addition to the application of ConSerts for two-vehicle platoons, a simulation-based approach has been explored to quantify the specification of safety properties that lead to a safe behavior on the MAS level.

In [130], the systematic derivation of ConSerts from safety concepts has been explored in more detail for platooning scenarios. In addition, the relation between Digital Dependability Identities (DDI) as developed in the H2020 DEIS project and ConSerts has been explicitly described. Figure 63, Figure 64 and Figure 65 show the results of this case study.

Since ConSerts rely on Boolean logic to express variants and binary variables to express runtime evidences, uncertainties in the presence of runtime evidences cannot be expressed and propagated towards the safety guarantees. To address this issue, [131] examined the relationship between ConSerts and Bayesian networks as a probabilistic inference method to address uncertainties of runtime evidences and their propagation to safety guarantee uncertainties.
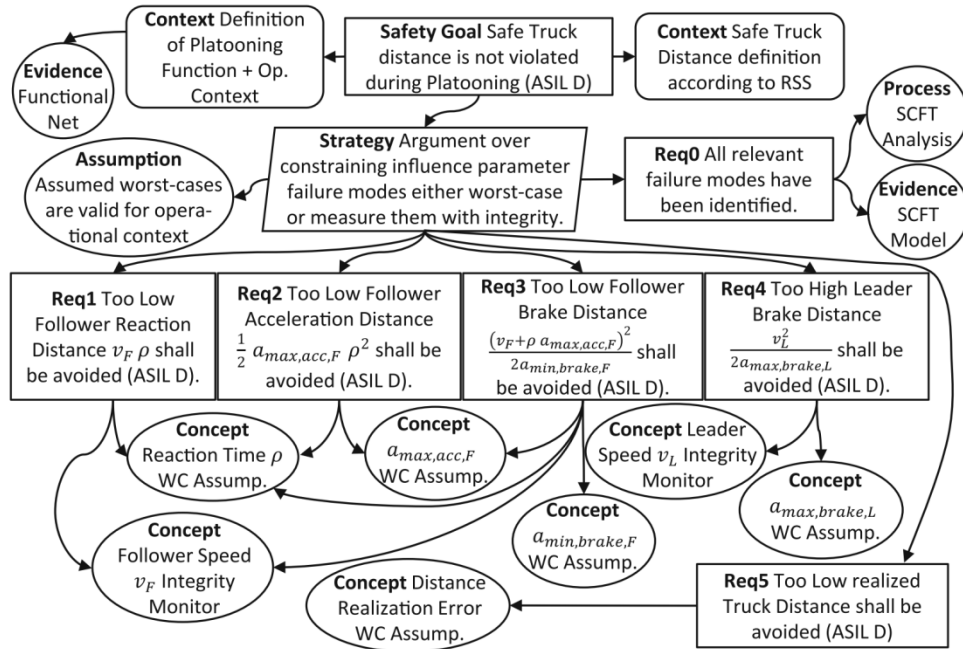
Figure 63 - Platooning safety concept
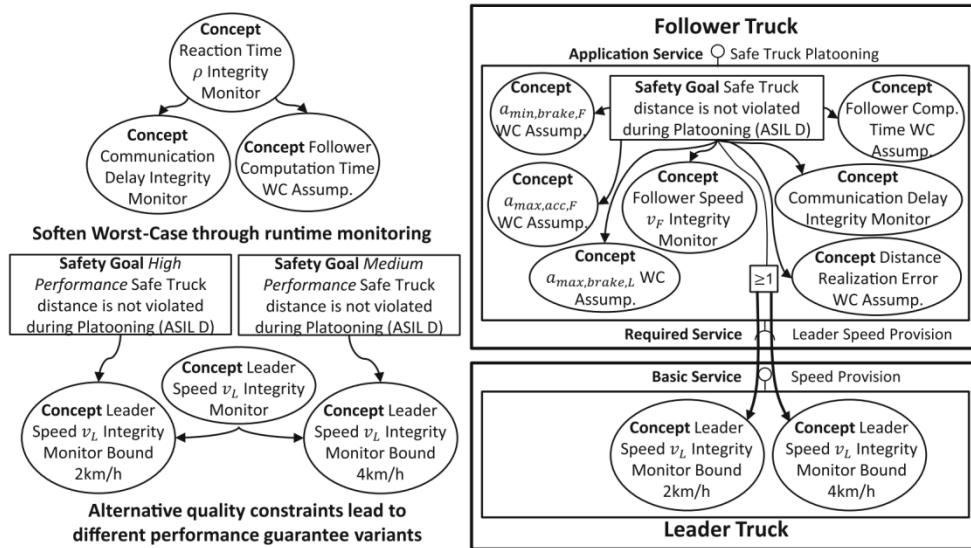
Figure 64 - Left: Platoon variant analysis, Right: Modular Platoon Safety Concept
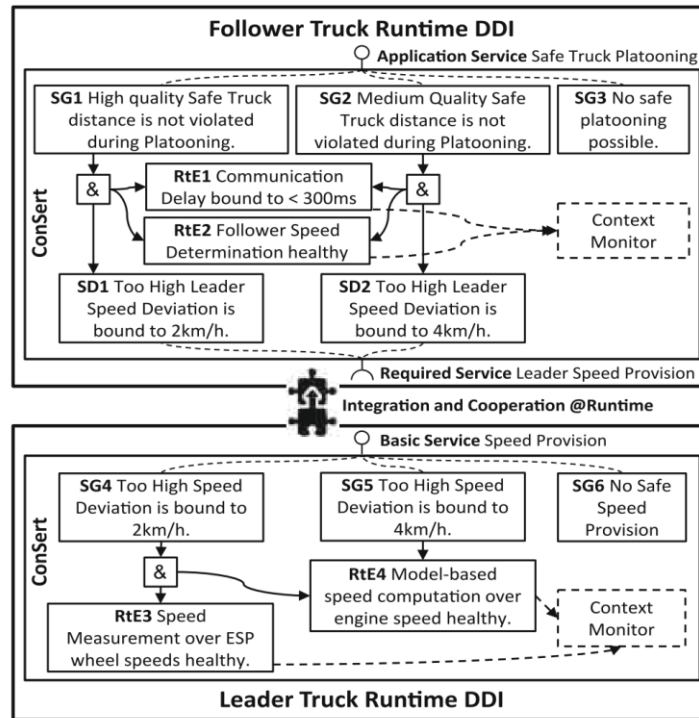
Figure 65 - Platooning runtime DDIs and ConSerts for leader and follower trucks

4.2.4 Model repair

When a model is being used as the basis of a runtime fault detection, fault diagnosis, or fault management system, the consequences of that model being incorrect or incomplete are much more significant than if the model was a purely design-time safety artefact. If the system is an MRS or other form of multi-agent system, then the issue is made worse because errors may impact not just the current system but also the others operating alongside it.

Errors in the runtime safety model may manifest in different ways. For example, limited or imperfect expert knowledge of the system design may mean that causes of failure exist that were not anticipated, thus not being present in the model. Alternatively, at the opposite end of the scale, potential hazards may exist that were unforeseen. Or the error may not be a causal issue at all, but one related to the monitoring of system parameters or the actions the system is expected to take in response to diagnosed faults (one can imagine the outcome of an automatic braking system that causes a vehicle to *accelerate* in response to a detected obstacle ahead rather than braking).

It is possible, however, for problems with the model to be identified at runtime without leading to a disaster first. By using data-driven approaches like ML, it is possible to perform some manner of corrective action when evidence of an unexpected failure scenario is encountered at runtime. If the observed behaviour of the system does not match that predicted by the runtime safety model, the model could be corrected or extended accordingly so that if a similar situation is encountered in future, it can be identified appropriately. This broad concept is known as *model repair*.

One example of model repair is the concept of process mining, in which analysts extract insights from a log of system data and create a performance model from it. These

models can then be updated through further data mining [132]. Process mining can be made iterative and hierarchical, such that during each iteration, the model is checked to detect any changes in steady-state behaviour.

A novel approach based on fault trees and machine learning is presented in [133]. The approach presupposes that safety analysts have already constructed fault trees at design time — models which may contain errors. Real-time operational data (e.g. during testing or immediately after deployment) is used to train a One Class Support Vector Machine to recognise the normal or abnormal behaviour of the system. Then, later during operation, fresh data provided by system monitors is compared to this normal behaviour to detect any anomalies. If such anomalies are detected, then the fault trees created earlier are consulted in an attempt to diagnose the abnormal behaviour. If the fault trees can provide no explanation, then one of several possible recommendations are made based on the perceived severity.
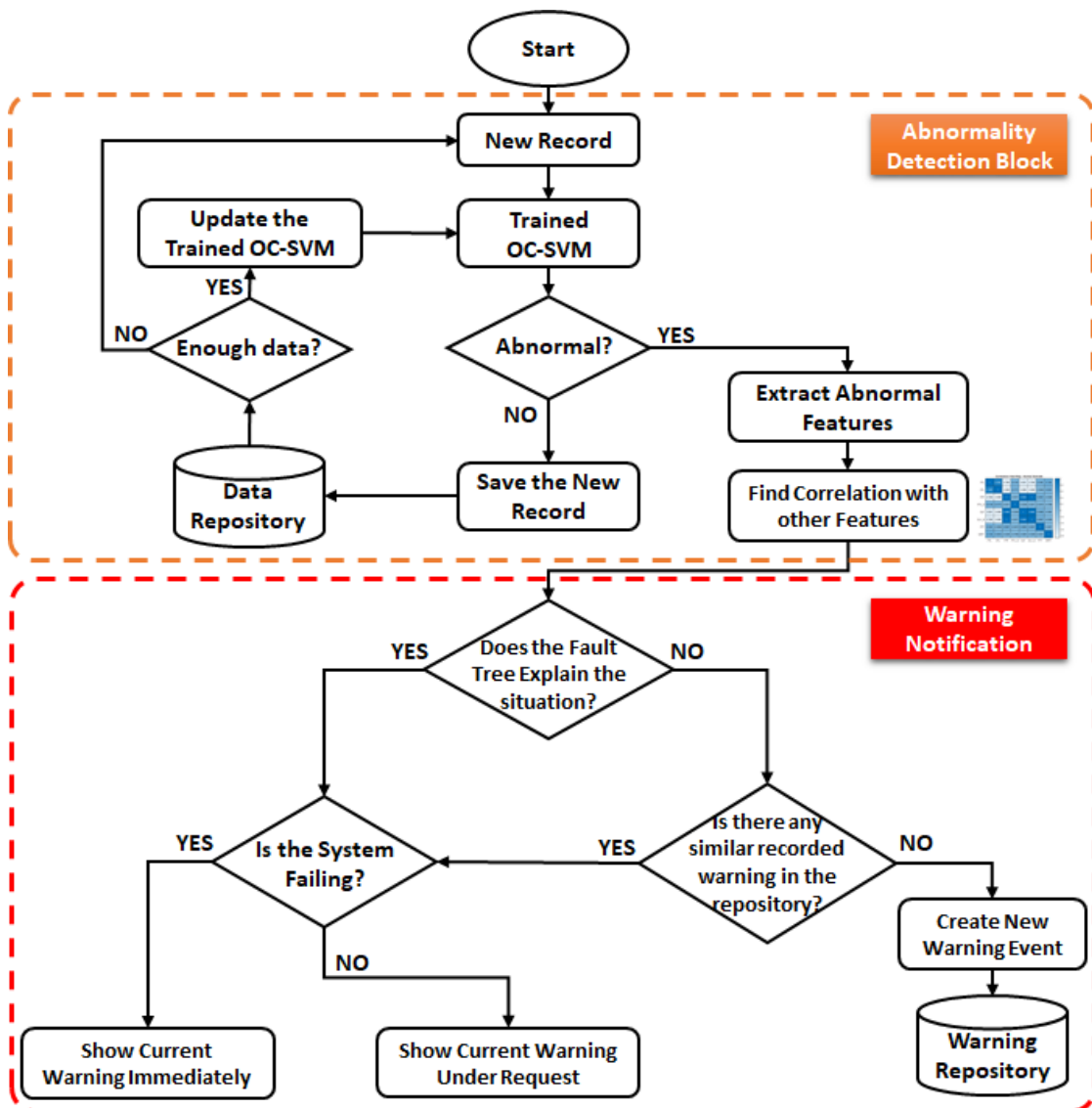
This process is illustrated in the figure below:

Figure 66 - Abnormality detection and response (from [133])

Confidentiality: Public Distribution

The fault detection is not so different from some of the techniques described in Section 4.2.1. The novelty of this approach stems more from the second part of the framework, the decision-making process.

When an anomaly has been detected, there are two possible outcomes:

- The diagnosis model (a fault tree in this case) can adequately explain the abnormal scenario, i.e., the observed behaviour matches a node (or nodes) in the fault tree. For example, if the behaviour is described by the two children of a specific AND gate, then it would appear that the fault tree correctly adequately describes the anomalous behaviour and a suitable diagnosis and prediction can be made accordingly.

- The other case is that the fault tree cannot explain the abnormal scenario. This could be because there are missing events — e.g. anomalous sensor readings have been encountered that are not represented by any event in the fault tree — or it could be because the causal relationships in the fault tree are incorrect. For example, an intermediate node might be identified as true on the basis of the current readings, but the fault tree says that situation should only occur if two events have occurred when in fact only one has occurred. In this case, the logic of the fault tree would appear to be incorrect: perhaps an OR relationship rather than an AND relationship. In such cases, a recommendation for potential repair of the fault tree is made. Depending on the problem identified, this may be a recommendation to add a new event to include any additional readings detected, or it may be to change the logic of the tree by e.g. changing the type of a gate. In any case, the situation and any warnings generated are stored in a repository so that if it arises once more, the safety monitoring system will recognise it.

The approach has been applied to an aircraft fuel distribution system. A fault tree containing a deliberate mistake was created. Different sensors monitored key aspects of the system, e.g. flow rates through valves, temperatures, and fuel levels in the fuel tanks. Readings from each sensor during normal operation are used to train the support vector machine. Then different abnormal scenarios are considered, each with anomalous readings. In one scenario, the abnormal behaviour is not explainable by the fault tree; fuel flow from the central tank (feeding both left and right engines) is interrupted. This corresponds to one or more of the events at the bottom left of the fault tree in Figure 67, each of which represents different problems with valves or pumps connected to the central tank. However, the AND gate highlighted red remains false, even though we know the top event has occurred (in this case, meaning that the flow of fuel to the left engine has been interrupted). This is because the second basic event immediately below it, representing a leak in the forward fuel tank, is also false.
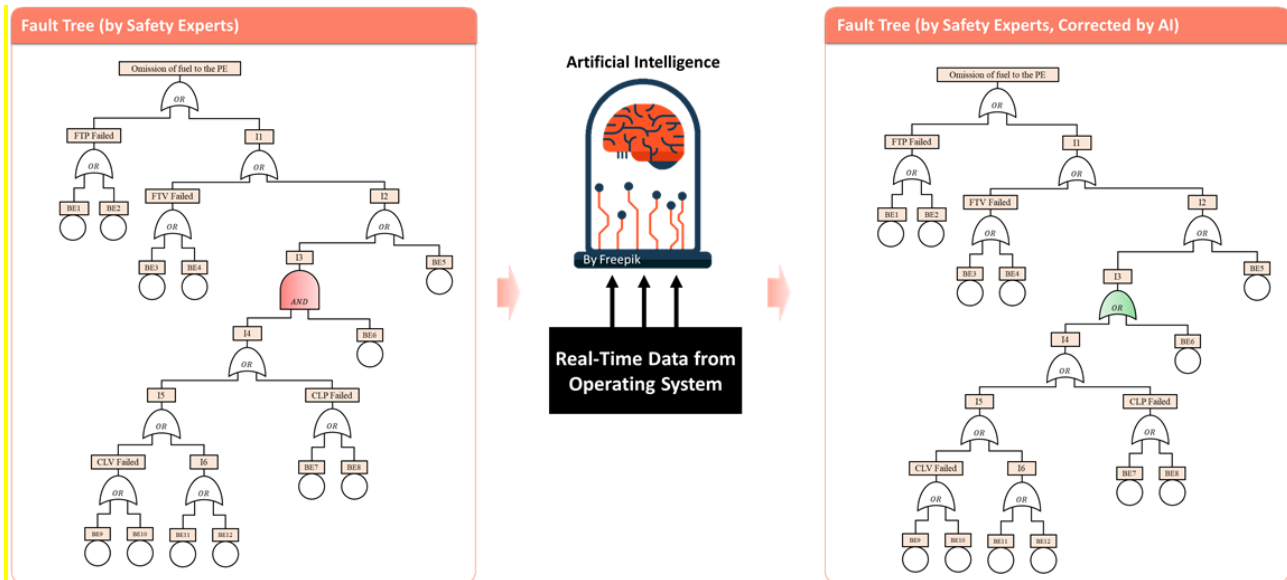
Figure 67 - Recommended repair of an erroneous fault tree

This points to a problem with the logical relationship between these events. The result would be the creation of a warning event and a suggestion to check the logic from the highlighted gate downwards, in this case to change the gate from an AND to an OR (thus correcting the deliberate error inserted into it).

Although this is still a very experimental approach, it could be of significant use as an element of a runtime safety monitoring system because it can work around potential shortcomings in the diagnostic models used. In the case of an MRS, it could also be used to issue notifications to other robots/agents operating as part of the wider system and, if they make use of the same models, even share corrections amongst them.

4.3 ASSURING SAFETY AT RUNTIME: THE EDDI CONCEPT

The Executable Digital Dependability Identity — or EDDI — is our solution not just to the challenge of autonomy, but also to the challenges of intelligence and complexity as well. The EDDI is the evolution of the DDI concept (see Section 2.2.5), extended to include a range of new features necessary to operate at runtime and address the sorts of issues faced by MRS and autonomous systems of systems in general.

Like DDIs, EDDIs are composable model-based artefacts that contain dependability information about a system. They can even be considered supersets of DDIs: an EDDI can contain everything a DDI does and can serve as a design-time dependability artefact in just the same way. Unlike DDIs, however, they are not purely static design-time artefacts; they are also intended to be executed at runtime onboard or alongside their target system to perform dynamic dependability management. This requires new semantics and protocols for distributed, cooperative operation to achieve collaborative certification and dynamic reasoning. It also means a renewed focus on security and safety working in tandem, to establish the safety implications of security threats and ensure a coordinated response. An EDDI is therefore both an online monitor, observing and managing its target system's safety and security, and an agent as part of a distributed system, communicating with other agents to help manage dependability of the wider multi-agent system.

An EDDI's features therefore include:

- Event monitoring to monitor dependability-related inputs from the system (such as readings from sensors);

- Runtime diagnostics to determine probable causes and possible consequences of detected failure events;

- Dynamic risk prediction, to update design-time risk estimates with new information based on the current system state;

- Mitigating actions and recovery planning, such as recommending the system enter a safe failure state or a degraded mode to continue operation.

- Intercommunication with other connected EDDIs to both assure them of the system dependability status and respond to errors reported by other EDDIs.

4.3.1 EDDI Architecture

EDDIs are not one-size-fits-all entities and can contain varying components according to need. In general, however, the architecture of an EDDI can be seen in the diagram below:
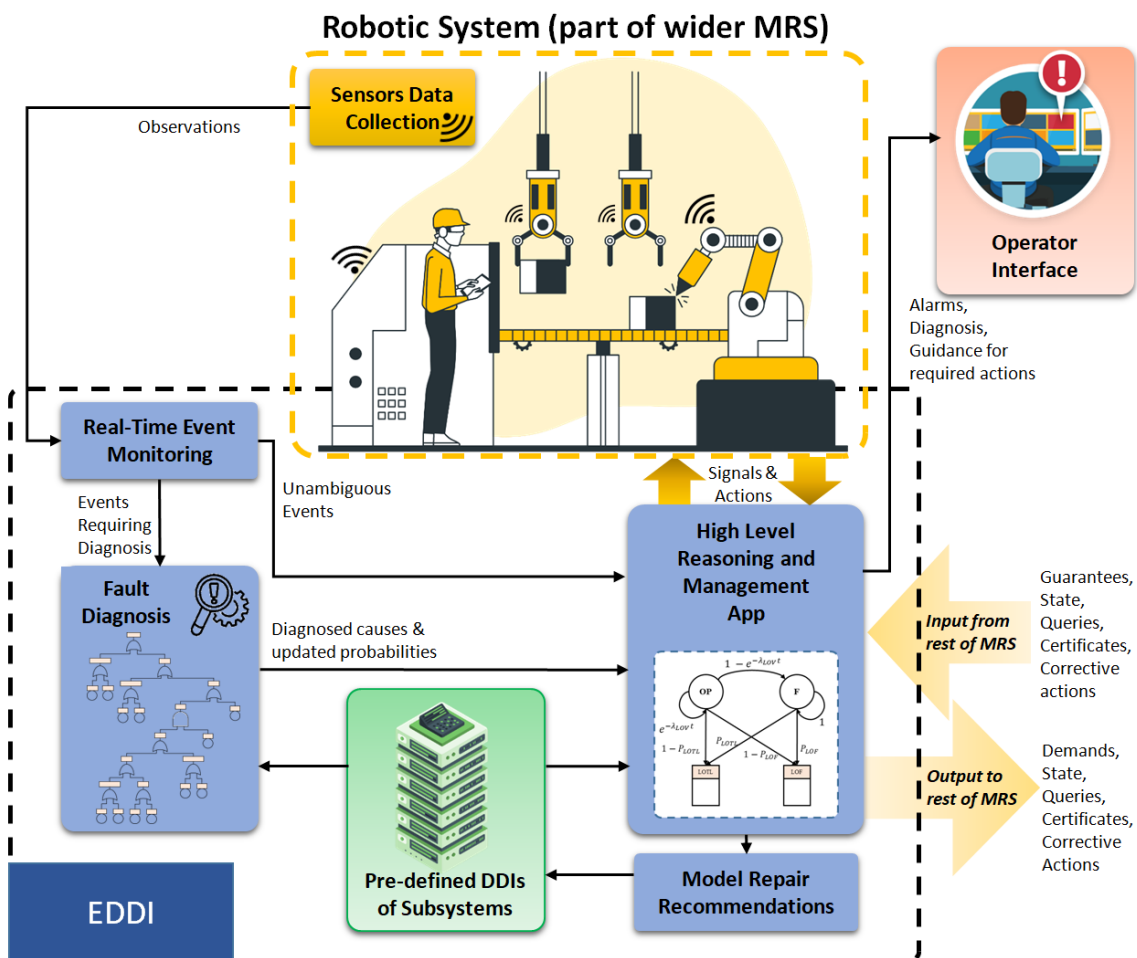

Figure 68 - The basic Executable Digital Dependability Identity architecture

4.3.1.1 *System Interface*

The EDDI requires an interface to its target system (e.g. a robot). This is shown in the diagram in two places: the observations (top left) coming from the system's sensors, and the signals and actions communicated directly to the system controller (centre-right). Together, these form the inputs received from the system and outputs the EDDI sends to it.

Through the system inputs, the EDDI receives information about the state of the system and its parameters, e.g. the readings from any onboard sensors that have dependability implications. Sensors also provide the EDDI with information about its operating environment and any prevailing conditions that may impact safety or security (e.g. bad weather). Such information will necessarily be platform-dependent, and most likely in a format dictated by the platform, though some pre-processing of the information may exist rather than reading e.g. raw sensor data, depending on the nature of the platform's controller.

In return, the EDDI sends to the system information about the dynamic level of risk as well as possible corrective measures. It is important to note that the EDDI is not meant to be a controller in itself, capable of seizing control of the system: it is an advisory monitor only. However, it can recommend e.g. that the system enter a safe failure state (such as an immediate landing for a drone, or a safe shut down for a robot arm) or that the present level of risk is such that the current task be aborted (resulting in a return to base, for instance).

Depending on the nature of the system, the EDDI may also have an interface to the human operator, if applicable. If the system is not fully autonomous, then the corrective actions may additionally (or instead) be communicated to the user, along with current system dependability status, any alerts about potential problems, and diagnostic information on any failures detected.

4.3.1.2 *MRS Interface*

If operating as part of a multi-agent system (such as an MRS), the EDDI may also require an interface to the rest of the agents to enact collaborative operation. This interface is denoted by the "Input from rest of MRS" and "Output to rest of MRS" on the right of the diagram. It is through these channels that the EDDIs work together to manage dependability for the wider system and help ensure safe operation of other agents.

As explained earlier, many of the tasks undertaken by an MRS are cooperative, requiring multiple robots working together in tandem to achieve. This can establish dependencies between robots, i.e., in order to achieve their own task, they must rely on another agent completing a different or related task. This dependency then becomes a dynamic certification problem: can the other agent guarantee that it is capable of achieving its task safely? And if not, how should the dependent agent respond?

Consider the example of a platoon of autonomous vehicles driving in convoy; the safe distance between vehicles depends on both the environmental conditions (wet weather would mean greater braking distances) as well as the speed and braking capability of the other vehicles in the convoy. If the lead vehicle experiences difficulties — whether due to environmental influences or onboard failures — it may report that it is no longer

capable of meeting its original guarantees of safe operation and drop to a degraded state with lower guarantees, e.g. it cannot ensure a short braking distance or a constant speed within a given threshold, but it can promise a longer braking distance and a looser speed threshold. In response, the following vehicle may choose to increase its following distance and/or reduce speed accordingly, which may in turn prompt the next vehicle in line to do the same and so forth.

It is through this MRS interface, therefore, that an EDDI communicates information about its current state, the dependability guarantees it offers about its own behaviour, and receives information about the guarantees offered by other agents it may depend on.

4.3.1.3 Real-Time Event Monitors

The event monitor components of the EDDI perform low-level detection of events. They are responsible for the evaluation of real-time sensory data and determining whether or not particular events of note have occurred (e.g. a fault), and if so, reporting this to the rest of the EDDI.

The exact form of an event monitor is heavily platform-specific, depending as it does on the nature of the data being monitored and the platform itself. However, there are frequent commonalities that can be generalised and so the event monitors can, to some degree, be considered instantiations of specific patterns. For example, sensor data is likely to be buffered into a time series store (e.g. via a circular buffer with shifting time windows), to better identify trends and long-term conditions and help filter out transient phenomena or spurious readings. Expressions to confirm the occurrence of events can be complex operations that involve querying both current and historical data points, and a system of three-value logic — incorporating an 'unknown' value in addition to 'true' and 'false' — may help in processing situations that involve a degree of uncertainty or otherwise incomplete information.

It is also important to note that EDDIs do not purely monitor for hardware faults. The system component being monitored may be an AI component, for instance, using SafeML or Uncertainty Wrappers to provide information about its current status and dependability of its performance. As described in Section 3, AI components such as DNNs introduce a more probabilistic type of uncertainty as they always have a chance to misclassify an input. Thus a camera connected to a person detection algorithm, for example, has a chance of not recognising a person in the vicinity of the robot, leading to a hazardous scenario. While we cannot detect this situation with 100% confidence, we may know that the ML component has a low confidence (e.g. due to environmental conditions like darkness or bad weather) and adjust behaviour accordingly.

Alternatively, an event monitor may be monitoring for security threats of some description, such as a network intrusion of some kind or a security authentication failure. As will be described later, both security and AI problems as well as more traditional hardware-related faults are subordinate to the higher-level reasoning of the EDDI so that it can respond to any dependability-related event, whether that be a hardware fault, security threat, or AI performance degradation.

4.3.1.4 Fault Diagnosis

Even if an event is detected, it may not necessarily represent a specific failure. More likely, it represents the symptoms of a failure. In order to determine the appropriate

response, it would be helpful to know the probable cause of the event and any potential consequences for the system dependability. This is where the diagnostic engine comes into play. Using causal models such as fault trees derived from design-time analyses (and stored in DDI-based repository, green in the diagram), the EDDI's diagnostic engine attempts to determine the root causes of the event(s) — such as a component failure — and can be used to dynamically update the risk if such a cause is detected. For example, a primary-standby system with a main component and a backup may still be able to operate if one or the other fails, but its reliability will be reduced and its risk will increase accordingly. Similarly, a hexacopter drone may still be able to fly with one or two engines disabled, but its performance and safety margin will be reduced.

In addition to Boolean models like fault trees, where detected events may indicate that certain nodes are true and allow both deductive (to determine causes) and inductive (to determine consequences) analysis, probabilistic models like Bayesian networks may be used to provide probabilistic estimates of likely causes where detectability coverage is limited. For example, a sensor that covers multiple components may report a problem without providing enough evidence to determine which component was at fault. But based on predicted failure rates and other information, the model may at least be able to indicate the probability that the problem is caused by each possible component.

4.3.1.5 *High-level Reasoning & Management Application*

The event monitors may detect events and the diagnostic engine may be able to determine causes and consequences of those events, but in order to make any sort of decision about possible responses — and to coordinate with the rest of the MRS, as well as any human operators — a higher-level 'brain' is required. Thus at the core of the EDDI is a model-based, high-level reasoning application capable of dynamically responding to input and acting accordingly. Such apps are therefore executable and deliver certification functionality (e.g. issuing and querying guarantees) as well as dependability management functions (e.g. alerting users and recommending mitigating actions in response to failures).

Candidates for the high-level app include ConSerts realizing dynamic safety capability assessment, Bayesian nets realizing dynamic risk assessment, and variations on State Machines (e.g. Markov models or state-sensitive fault trees). These will have knowledge of possible nominal, degraded, and failed modes, as well as the causal transitions between them, and any necessary actions to take for each mode. Development of appropriate models also requires working knowledge of the normal operation of the system, since the current system state may affect both possible events and appropriate reactions (e.g. a drone that has landed may have different responses to an engine failure than an airborne drone).

It may also be possible to use more than one model in conjunction to form the overall app, to take better advantage of the capabilities of each type of model. For example, ConSerts could be used to determine whether safety goals are being met across the overall MRS by issuing and querying safety guarantees. They are capable of verifying whether the demanded safety self-certification is capable of being satisfied, and if not, react accordingly. However, they are primarily Boolean models, so Bayesian networks may also be used to enrich ConSerts with probabilistic reasoning mechanisms. Dynamic safety goals to be satisfied dynamically via ConSert safety guarantees can come from a dynamic probabilistic risk estimation. Finally, some form of state machine could also be

used to track the current state and tasks of the system and to manage the transition into safe states and other mitigating actions in response to specific triggers.

4.3.1.6 *Model validation and repair recommendations*

EDDIs are model-based artefacts. If the EDDI model itself contains errors, then its own behaviour may be unreliable. To address this, EDDIs may contain an experimental model validation & repair component that monitors the EDDI at runtime (or during a simulation at design time) for correctness and completeness. If it detects a divergence between the detected system state and that predicted by the model, then it can report the discrepancy to the operator. It may even be possible to perform some degree of self-correction, e.g. by augmenting the model with new causal relationships or states, perhaps via deep learning techniques (see Section 4.2.4).

4.3.2 **EDDI Creation and Deployment**

No two EDDIs are exactly alike. They can grow and change and evolve across the system lifecycle. Preliminary EDDIs could be constructed even as soon as the early stages of functional design, developing alongside the system design with additional detail and new failure behaviour.

These design time EDDIs could be considered to be "Extended DDIs" rather than "Executable DDIs". Generated by dependability analysis tools such as HiP-HOPS or safeTbox, they would be ODE-based repositories of knowledge about the system architecture, failure behaviour (e.g. fault trees), and safety assurance argumentation (via SACM). Different systems and different tools may produce different forms of model; a dynamic system may make use of Markov models and/or Bayesian networks, for instance, while a more static system might only require fault trees.

These EDDIs still have many roles to play even at design time. They can be used to inform testing or simulation (e.g. with fault injection), and the ability to encapsulate and hide proprietary information while still exposing an interface to the safety argumentation means they can be used as part of a distributed supply chain, with suppliers receiving or providing EDDIs as evidence that safety requirements for individual components are being met.

However, as already discussed, design time measures can only do so much. In making the move from design time to runtime, EDDIs transition to an executable form. During this process, manual intervention is required, new information is added to the EDDI, and unnecessary information (e.g. some of the safety argumentation) is removed. For example:

- In order to function, an EDDI requires data from runtime evidence monitoring. This requires platform-specific event monitors to read sensors and communicate with the EDDI. While some generic information about the event monitor can be prepared (e.g., the nature of the event to be detected), creating the monitor itself is a separate activity.

- Diagnostic models like fault trees can, for the most part, be generated from the design-time causal failure models. However, the higher-level reasoning models are likely to be more complex and require more intervention. Though they could

be based on e.g. a state machine or Bayesian network, effort is needed to connect them to the relevant diagnostic models and event monitors.

- Similarly, any distributed safety assurance — the issuing, checking, updating, and receiving of safety guarantees — is also likely to require intervention. It may be based on existing design time argumentation already present in the EDDI, but extra work will be needed to tailor this to the runtime environment the system is likely to face.

Once suitably prepared, there are two possible approaches for deploying an EDDI. The first is to generate code from it. A ConSert, for example, could be synthesised into code that runs natively on the target platform (e.g. as a ROS node). Again, most likely some degree of manual tweaking would be required.

The other approach is more difficult but potentially more generic and would use a virtual machine-style approach: a target-specific native program is created that takes an arbitrary EDDI model and executes it. This imposes a higher overhead, since creating the executing program is more difficult, but the advantage is that it can then execute any EDDI without needing to modify the program. Even here, however, there will be work needed to e.g. connect the executor to the event monitors so that the EDDI can function.

More work in this area is being undertaken as part of WP7.

5. THE SESAME SAFETY METHODOLOGY

The previous sections have defined three key challenges to safety assurance of Multi-Robot Systems: Complexity, Intelligence, and Autonomy & Openness. The state of the art in each area was reviewed and key techniques were identified that can be used to address those challenges — most importantly, the EDDI concept. For most effective use, however, these techniques need to be combined into a cohesive whole so that the EDDI concept can be successfully applied throughout the lifecycle of the system, from design to runtime.

Much effort has gone into the definition of safety methodologies and lifecycles across safety standards. While there are some differences, e.g. to address domain-specific concerns, in general the processes established by major standards like ISO 26262 and ARP4754-A follow the same general outline:

- An initial hazard analysis is undertaken to identify hazards and assess risk. For example, in ISO 26262 a Hazard Analysis & Risk Assessment takes place as part of the concept phase, while in ARP4754-A, a Functional Hazard Assessment is carried out to identify the circumstances and classify severity of failure conditions.

- The risk of each hazard is generally categorised according to a safety integrity level (e.g. ASIL, DAL etc.) which provides a qualitative measure of risk.

- Safety requirements/safety goals are generated to prevent or mitigate each hazard, inheriting their integrity level.

- Analysis is undertaken to understand how different parts of the system architecture may be responsible for different hazards (e.g. via an FTA or FMEA). In ARP4754-A this is known as PSSA or Preliminary System Safety Assessment.

- On the basis of this analysis, derived safety requirements are allocated on a top-down basis to lower-level system components according to their contribution to the hazards. This is typically an iterative process that accompanies the development of the system through conceptual, functional, and technical design phases.

- During the testing and integration phases, bottom-up verification & validation activities take place to ensure that the requirements are being met. Testing makes sure that the system design has been correctly implemented, and further safety analyses (e.g. System Safety Analysis in ARP4754-A, Functional Safety Assessment in ISO 26262) provides evidence to verify whether the system complies with the safety requirements.

- Although not explicitly part of most standards, the requirements and evidence gathered throughout these phases forms part of the safety case/assurance case of the system, which can be important for safety certification etc.

In SESAME, we have chosen to take advantage of this established practice by adopting a similar process, illustrated in Figure 69 below:
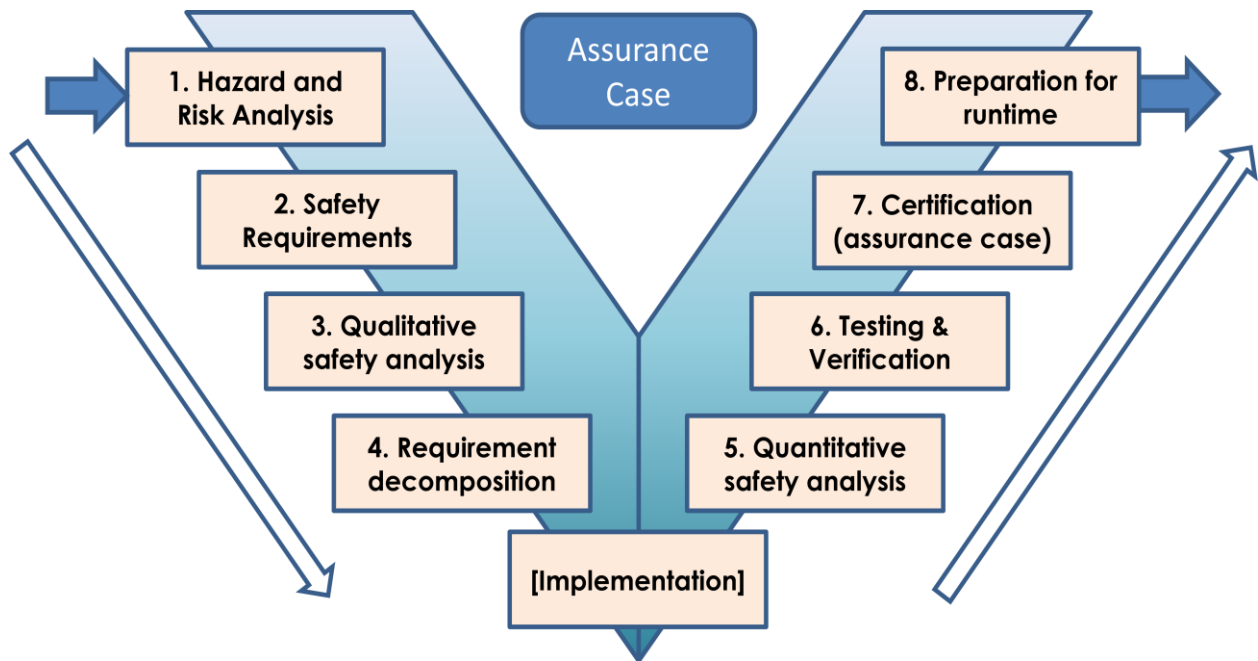
Figure 69 - V model for the design-time safety lifecycle

However, there are a few differences as a result of the need to include the transition from the design-time process to runtime safety monitoring and management. Firstly, additional information is collected to better capture the variability of risk. Whereas in a traditional analysis only the worst case is assumed, here more varied data is acquired to enable the eventual runtime EDDI to better perform dynamic risk assessment on the basis of current operational conditions, not merely worst-case predictions. This particularly affects the initial HARA, where a wider assessment of environmental variability is considered.

Consideration is also given to the potential dependencies between different agents of the multi-robot system; even if the specifics of the final operating environment are not known, the guarantees offered and demands issued by the system under development can be established.

Finally, an additional step is added to the end of the process which is responsible for preparing executable runtime safety models (i.e., the EDDIs). While much of the information gathered throughout the rest of the process is still relevant here, additional technical work is needed to adapt the model to the target platform — e.g. by developing appropriate event monitors and linking them to hardware sensors — and also to create the high-level reasoning application with the necessary logic to perform dynamic risk assessment and respond to any failures detected. As part of this, the distributed safety certification functionality (e.g. in the form of a ConSert) is also developed.

This also means that extra information may need to be captured in earlier steps. For example, it may not be enough to simply record that a given failure of a given component causes a particular hazard, but also which sensors can detect evidence of that component failure.

5.1   HAZARD ANALYSIS AND RISK ASSESSMENT

The first step in the process is the identification of hazards and assessment of risk. Section 2 has already identified techniques like HAZOP or FME(C)A that can be applied for this purpose, but the unique difficulties posed by MRS may require a more extensive approach. In particular, some consideration must also be given to the environment and likely operating context of the system, rather than limiting the analysis solely to the system itself.

Domain-specific standards should be followed where appropriate. For example, ISO 26262 mandates that a hazard requires both an item malfunction *and* a particular scenario, since the risk of a given malfunction depends on the circumstances in which it occurs (a brake failure while driving on a wet road at night is more critical than while on a dry road in daytime, for example).

The scenario is not exclusive to ISO 26262. As part of the environmental hazard analysis, relevant information about the expected operating environment should be extracted to identify safety-critical situations or scenarios. Some of the information about the environmental variability (e.g. different operating conditions etc.) could be derived from any relevant Executable Scenario specifications or as part of a separate environmental variability analysis.

Once suitably identified, risk assessment can take place for each hazard. Again, different standards may impose different procedures, but in general risk is assessed on the basis of:

- *Likelihood*: how likely is the hazard to occur? May be termed probability or exposure etc. in different standards.

- *Severity*: how severe are the consequences of the hazard?

- In some cases, a third attribute is considered, most commonly either *detectability* (how easy is it to detect or diagnose the malfunction before it can cause the hazard) or controllability (how easy is it to mitigate or prevent the hazardous event).

The DDI already contains elements to support hazard and risk analysis as part of its HARA package. The EDDI is expected to extend this further to encompass any additional information required, e.g. about hazardous environmental scenarios or environmental variability in general. In addition, the EDDI will contain sufficient expressive power to express situation features indicating variable risks and thus prepare the synthesis of dynamic risk assessment models for MAS.

5.2   SAFETY REQUIREMENTS

Once hazards have been identified and their risks assessed, this information can be used to inform the definition and allocation of high-level safety requirements. Typically this entails assigning a particular safety integrity level to each requirement (or its equivalent in whichever relevant standard is being applied). In many cases, a safety requirement may also require the definition of a corresponding safe state or safety mechanism to mitigate the hazard.

These top-level safety requirements can then be allocated to the system design architecture. Components responsible for causing particular hazards are allocated the requirements those hazards inspired, receiving the corresponding integrity level in the process.

The EDDI will contain this information as part of its Requirements and Dependability packages.

5.3 QUALITATIVE SAFETY ANALYSIS

As the design evolves, it is important to keep track of the causal relationships between hazards and malfunctions of elements of the system (as well as any environmental factors). This is the purpose of hazard causal analysis, a form of safety analysis. Different techniques can be employed for this purpose, e.g. a qualitative FTA, and the resulting models record the failure propagation behaviour of the system. However, compositional FTA techniques like HiP-HOPS or CFTs as implemented by safeTbox are particularly useful because they allow future decomposition to subcomponents as the design architecture becomes more detailed.

However, static Boolean analyses of system architectures are not the only possibilities here. Behavioural models could be produced to support dynamic analyses such as Markov analysis, Bayesian networks, or Dynamic FTA. These may provide more insight into the dynamic failure behaviour of the system, complementing the propagational models (such as static fault trees or FMEAs).

This casual failure information will all be stored as part of the EDDI's Failure Logic package (and relevant subpackages). DDIs already support FMEA, FTA, and Markov, but for EDDIs this is likely to be extended to support at least generic state machines and Bayesian networks. Meanwhile, the information about the system architecture is stored as part of the Design package, ensuring that the EDDI maintains a complete picture of how failures propagate through the system to cause hazards.

5.4 REQUIREMENTS DECOMPOSITION

As the design evolves and the architecture becomes more detailed, moving from abstract functional architectures to more detailed views over technical implementation, the safety requirements may be decomposed to the new subcomponents. This can be performed with the aid of tools with optimisation-based decomposition functionality, such as HiP-HOPS (see Section 2.2.3).

Decomposition takes place on the basis of the causal models established earlier to ensure that those components that contribute to a potential hazard fall under the auspices of the relevant safety requirements (and integrity level allocations). These logical models also establish cases where multiple components are jointly responsible for meeting the requirement, in which case the integrity levels may be divided amongst them, according to the restrictions set out in whichever standard applies.

The EDDI tracks this information through various traceability elements as well as updated design architecture models and causal failure analyses as needed.

5.5 QUANTITATIVE SAFETY ANALYSIS

Once more concrete information about the system implementation is available — particularly quantitative failure data — further safety analysis becomes possible. At this stage of the design process, the analysis is less about establishing links between causes and effects and more about verifying whether the decomposed safety requirements are being met.

Many of the same analysis tools used earlier (HiP-HOPS, safeTbox etc.) can also be applied here, in most cases using the same models that have since been annotated with the additional probabilistic failure data required.

It is worth pointing out that this process can occur at many stages, as long as the necessary data is available; it need not wait until after the system implementation. Indeed, it can help guide component selection by indicating whether or not a given component is reliable enough to meet the safety requirements; in some cases, this could even lead to an architectural optimisation process.

As always, the results of these analyses are stored in the ever-evolving EDDI, e.g. as part of the relevant FailureLogic analysis packages or the Design package elements.

5.6 TESTING & VERIFICATION

Further verification of the safety requirements requires testing. While this area is of particular focus to SESAME's WP6, it is also relevant with regard to any ML components. The various ML assurance techniques outlined in Section 4 (e.g. SafeML, Uncertainty Wrappers, SMILE) may be employed here to help establish the robustness and reliability of any ML components that are undergoing training and testing.

While ML models are unlikely to be stored as part of the EDDI directly, some information on the nature of the model and the outcomes of the safety techniques applied is expected to be stored to support further application of those techniques at runtime. Such components should already be part of the architectural and behavioural system models, particularly where they may be causes of hazards.

5.7 CERTIFICATION & ASSURANCE CASES

Although certification is often regarded as a process that takes place near the end of the system design lifecycle, in reality the assurance case will have been slowly building up throughout development (as indicated by the box within the V in the earlier diagram).

SACM is the backbone of the EDDI, and much of the evidence should already have been collected during previous analyses. At this stage, this may be augmented with any further evidence, while remaining argumentation is generated to complete the assurance case and link the requirements, the architecture, and the evidence together.

5.8 PREPARATION FOR RUNTIME

For the EDDI to be suitable for deployment and execution at runtime, it must undergo a transition from a purely design-time artefact to a runtime one. As described earlier, this involves several aspects, from the definition of event monitors to collect runtime evidence, the implementation of the high-level reasoning application (itself likely to be based on one or more of the design-time failure behavioural models), and any work

necessary to ensure the EDDI can operate as part of the wider MRS, e.g. in terms of offering or receiving safety guarantees.

This latter functionality is likely to be provided by ConSerts, as described in Section 4. Safety guarantees and safety demands are defined along with a safety concept that argues why the relevant system (or its components) can fulfil its guarantees as long as its demands are also met. Once defined, code generators can convert ConSert models into code that can be executed directly on the target platform.

More work on this particular aspect will take place as part of WP7.

5.9   ON SECURITY

Security and safety are both important aspects of dependability. Only with full consideration of both can an accurate assessment of overall dependability be achieved. Security threat modelling may reveal vulnerabilities and possible attacks that may have vital implications for safety, whether at design time or at runtime. A security threat may be identified as a potential cause of a hazard, leading to revisions of the design to improve robustness and/or mitigate the risk. A security vulnerability may also be the subject of a runtime monitor in an EDDI, which could react to a detected attack by revising its safety guarantees or moving into a safe state.

One way this could be achieved is by directly linking security models to safety models within the EDDI. For example, the top event of an attack tree could be linked as a leaf node of a fault tree, indicating a direct causal relationship. This would allow the runtime diagnostic engine to consider it as a potential cause of a given failure, and evidence from any runtime security monitors could serve as a warning of potential hazards just like anomalous readings from sensors.

In this sense, as well as being an important process in itself, security analysis can be also considered a subordinate input to the overall safety methodology, whether that be as a potential cause of a hazard, a risk factor, or as an event of note that needs monitoring in order to avoid compromising safety. Just as the behaviour AI component can have implications for safety, so can a security-critical component, and this informs the safety reasoning model used in the EDDI. The EDDI itself also raises security concerns, given its need to communicate securely with other EDDIs and ensure that safety certificates are not falsified or tampered with.

It is important, therefore, to ensure security analysis is integrated as part of the wider dependability process and that it should be one of the objectives that drives the design process just as safety should be. Security analysis would take place alongside safety analysis throughout the lifecycle: security issues should be considered as potential causes of hazards and their risks assessed appropriately; causes and consequences should be analysed via models like attack trees, broadly analogous to fault trees; safety requirements should include security aspects where threats are a potential concern; and argumentation on security issues could form part of assurance cases for certification purposes.

Finally, the runtime EDDI may also include security models and monitors for detecting security threats during operation, which feed into the same high-level reasoning

application and help to ensure runtime safety by performing dynamic risk assessment of security issues alongside safety issues.

For more information on security, please consult deliverable D5.1: Security Analysis Concept & Methodology. Further work on the topic is expected as part of WP5 (primarily focusing on design time security analysis) and WP7 (for runtime safety & security as part of the EDDI concept).

6.   CONCLUSIONS

Three major challenges to safety of MRS have been identified: complexity, intelligence, and autonomy & openness. For each challenge, we have reviewed a range of state-of-the-art techniques that may help address those challenges and described how the technologies we develop can be combined in SESAME to form an overall safety concept for MRS, the EDDI.

The challenge of complexity is largely addressed by means of compositional model-based safety analysis techniques that can break down the complexity into more manageable parts. This applies both to scale — modelling systems hierarchically and embedding local failure logic at the component-level — and to tasks, where different safety-related tasks (including not just analysis but also requirements allocation and assurance case generation) can be handled by the same set of models. All of this can be combined with the existing DDI concept to create models — EDDIs — that store all of the necessary information to support a gamut of design-time safety processes.

Against the challenge of intelligence we propose a trio of techniques: SafeML and Uncertainty Wrappers for estimating the confidence of a given classification, which can be used as a form of reliability measure, and SMILE for explainability purposes. By enabling us to measure and explain the reliability of ML decision making, we can integrate ML behaviour as part of a wider system safety model, e.g. as one input into a fault tree or Bayesian network. In addition to providing valuable feedback during training, testing, and verification, this allows the EDDI to perform runtime safety monitoring of ML components.

The EDDI itself is therefore our primary solution to the twin challenges of autonomy and openness. Using the ConSert approach as a foundation, EDDIs can be made to operate cooperatively as part of a distributed system, issuing and receiving guarantees on the basis of their internal executable safety models to collectively achieve tasks in a safe and secure manner. In addition, dynamic risk assessment approaches such as SINADRA will provide the basis to dynamically determine the safety goals to be fulfilled by the MAS in the current operational situation.

Ongoing work to apply these technologies to the use cases is in progress (see the Appendix for more) and the next major step for WPs 4, 5 and 7 is to develop and release a new, expanded version of the ODE that includes the additional features necessary to better support the full range of EDDI functionality. In addition, further tool development work is underway to improve existing tools and add or extend ODE and EDDI support.

An updated version of this deliverable (D4.5) that takes into account the new developments is expected by M30. Toolwork is represented by D4.4 (M18) and D4.6 (M30). The new ODE specification (D4.2) and Combined Safety-Security Framework (D4.3) are also planned for M18. Closely related are D7.2 (tools for generating runtime EDDIs) and the preliminary and final versions of the Runtime Safety & Security EDDI Concept (D7.1 and D7.3, M18 and M30 respectively). These related deliverables are all summarised in the table overleaf.

Table 5 - Related SESAME deliverables

Deliverable	Description	Due Date
D4.2 ODE specification	Extension of the existing ODE to incorporate new features to support EDDI functionality.	M18
D4.3 Combined Safety & Security EDDI framework	The combined safety and security EDDI concept with links to other SESAME artefacts (e.g. Executable Scenarios).	M18
D4.4 Initial design-time EDDI Safety Tools	Design-time safety analysis tools with support for EDDIs (HiP-HOPS & safeTbox).	M18
D4.5 Updated EDDI Concept & Methodology (updated version of D4.1)	Updated version of this deliverable to account for new developments.	M30
D4.6 Updated design-time EDDI Safety Tools	Updated version of the safety analysis tools to account for new developments.	M30
D7.1 Initial Runtime Safety & Security EDDI Concept	Definition of the runtime EDDI concept and associated algorithms for safety & security assurance.	M18
D7.2 Tools for generating runtime EDDIs	Tools for generating runtime EDDIs from design-time EDDIs.	M18
D7.3 Updated Runtime Safety & Security EDDI Concept	Updated version of the runtime EDDI concept (D7.1) to incorporate new developments.	M30

7. REFERENCES

[1] A. Joshi, M. Heimdahl, S. Miller and M. Whalen, "Model-based Safety Analysis," 2006. [Online]. Available: https://shemesh.larc.nasa.gov/fm/papers/Joshi-CR-2006-213953-Model-Based-SA.pdf. [Accessed 1 11 2021].

[2] H. Government, "Health and Safety at Work etc Act 1974," UK Government, London, UK, 1974.

[3] British Standards Institution, "BS EN 61882:2016 - Hazard and operability studies (HAZOP studies) Application Guide," British Standards Institution, 2016.

[4] US Department of Defense, "MIL-P-1629: Procedure for Performing a Failure Mode, Effects and Criticality Analysis," US Department of Defense, Washington DC, USA, 1949.

[5] US Department of Defense, "MIL-STD-1629A: Procedure for Performing a Failure Mode, Effects and Criticality Analysis," US Department of Defense, Washington DC, USA, 1980.

[6] W. E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick and J. Railsback, "Fault Tree Handbook with Aerospace Applications," NASA Office of Safety and Mission Assurance, USA, 2002.

[7] J. B. Dugan, B. Venjataraman and R. Gulati, "DIFtree: a software package for the analysis of dynamic fault tree models," in *Annual Reliability and Maintainability Symposium*, Philadelphia, USA, 1997.

[8] M. D. Walker, L. Bottaci and Y. I. Papadopoulos, "Compositional Temporal Fault Tree Analysis," *Computer Safety, Reliability, and Security. SAFECOMP 2007. Lecture Notes in Computer Science,* vol. 4680, pp. 106-119, 2007.

[9] G. K. Palshikar, "Temporal Fault Trees," *Information and Software Technology,* vol. 44, no. 3, pp. 137-150, 2002.

[10] S. Kabir and Y. Papadopoulos, "Applications of Bayesian networks and Petri nets in safety, reliability, and risk assessments: a review," *Safety Science,* vol. 115, pp. 154-175, 2019.

[11] P. H. Feiler and A. Rugina, "Dependability Modeling with the Architecture Analysis & Design Language (AADL)," 2007. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8277. [Accessed 111 11 2021].

[12] S. Sharvia, Y. I. Papadopoulos, D. Chen, M. D. Walker, Y. Wenjing and H. Lönn, "Enhancing the EAST-ADL error model with HiP-HOPS semantics," *Athens Journal of Technology & Engineering (ATINER),* vol. 1, no. 2, pp. 119-136, 2014.

[13] M. Biehl, D. Chen and M. Törngren, "Integrating safety analysis into the model-based development toolchain of automotive embedded systems," *ACM SIGPLAN Notices,* vol. 45, no. 4, pp. 125-132, 2010.

[14] S. Sharvia, S. Kabir, M. Walker and Y. I. Papadopoulos, "Model-based dependability analysis: State-of-the-art, challenges, and future outlook," in *Software Quality Assurance*, Morgan Kaufmann, ISBN: 978-0-12-802301-3, 2016, pp. 251-278.

[15] M. Wallace, "Modular architectural representation and analysis of fault propagation and transformation," *Electronic Notes in Theoretical Computer Science,* vol. 141, no. 3, pp. 53-71, 2005.

[16] B. Kaiser, P. Liggesmeyer and O. Mäckel, "A New Component Concept for Fault Trees.," in *Safety Critical Systems and Software 2003, Eighth Australian Workshop on Safety-Related Programmable Systems, (SCS2003)*, Canberra, Australia, 2003.

[17] B. Kaiser, D. Schneider, R. Adler, D. Domis, F. Mohrle, A. Berres, M. Zeller, K. Hofig and M. Rothfelder, "Advances in component fault trees," in *Safety and Reliability - Safe Societies in a Changing World*, London, UK, CRC Press, Taylor Francis, 2018, p. 9.

[18] M. Zeller and F. Montrone, "Combination of Component Fault Trees and Markov Chains to Analyze Complex, Software-controlled Systems," in *3rd International Conference on System Reliability and Safety (ICSRS)*, Barcelona, Spain, 2018.

[19] L. Grunske, B. Kaiser and Y. I. Papadopoulos, "Model-driven safety evaluation with State-Event-based Component Failure Annotations," in *Proceedings of the 8th International Symposium on Component-based Software Engineering (CBSE'05)*, 2005.

[20] B. Kaiser, C. Gramlich and M. Forster, "State/event fault trees—A safety analysis model for software-controlled systems," *Reliability Engineering and System Safety,* vol. 92, pp. 1521-1537,

2007.

[21] G. Ciardo and C. Lindemann, "Analysis of deterministic and stochastic Petri nets," in *Proceedings of the 5th International Workshop on Petri Nets and Performance Models*, Toulouse, France, 1993.

[22] R. German and J. Mitzlaff, "Transient analysis of deterministic and stochastic Petri nets with TimeNET.," in *Proceedings of the 8th international conference on computer performance evaluation, modelling techniques and tools and MMB (Lecture Notes in Computer Science, vol. 977)*, Heidelberg, Germany, 1995.

[23] Y. I. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *Proceedings of the 18th International Conference in Computer Safety, Reliability, and Security; collected in LNCS by Springer, vol 1698 (p139-152)*, Toulouse, France, 1999.

[24] Y. I. Papadopoulos, M. D. Walker, D. J. Parker, E. Rude, R. Hamann, A. Uhlig, U. Gratz and R. Lien, "Engineering Failure Analysis & Design Optimisation with HiP-HOPS," *Journal of Engineering Failure Analysis,* vol. 18, no. 2, pp. 590-608, 2011.

[25] Y. I. Papadopoulos, "A Synthesis of Logic and Biology in the Design of Dependable Systems," *IFAC-PapersOnLine,* vol. 48, no. 7, pp. 1-8, 2015.

[26] Z. Mian, L. Bottaci, Y. I. Papadopoulos and N. Mahmud, "Model Transformation for analyzing Dependability of AADL model by using HiP-HOPS," *Journal of Systems and Software,* vol. 151, no. 11, pp. 258-282, 2019.

[27] D. J. Parker, M. D. Walker, L. S. Azevedo, Y. I. Papadopoulos and R. E. Araujo, "Automatic Decomposition and Allocation of Safety Integrity Levels Using a Penalty-Based Genetic Algorithm," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, Amsterdam, Netherlands, 2013.

[28] I. Sorokos, Y. I. Papadopoulos, L. S. Azevedo, D. Parker and M. D. Walker, "Automating Allocation of Development Assurance Levels: an extension to HiP- HOPS," in *5th International Conference on Dependable Control of Discrete Systems*, Cancun, Mexico, 2015.

[29] S. Sharvia and Y. I. Papadopoulos, "Integrating model checking with HiP-HOPS in model-based safety analysis," *Reliability Engineering & System Safety,* vol. 135, pp. 64-80, 2015.

[30] S. Kabir, M. D. Walker and Y. I. Papadopoulos, "Dynamic system safety analysis in HiP-HOPS with Petri Nets and Bayesian Networks," *Safety Science,* vol. 105, pp. 55-70, 2018.

[31] D. Whiting, I. Sorokos, Y. I. Papadopoulos, G. Regan and E. O'Carroll, "Automated Model-Based Attack Tree Analysis Using HiP-HOPS," in *6th International Symposium on Model-based safety and assessment (IMBSA 2019)*, Thessaloniki, Greece, 2019.

[32] P. O. Antonino, D. S. V. Moncada, D. Schneider, M. Trapp and J. Reich, "I-SafE: An integrated Safety Engineering Tool," *IFAC-PapersOnLine,* vol. 48, no. 7, pp. 23-28, 2015.

[33] A. Arnold, A. Griffault, G. Point and A. Rauzy, "The AltaRica formalism for describing concurrent systems.," *Fundamental Informatica,* vol. 40, no. 2-3, pp. 109-124, 2000.

[34] M. Bozzano and A. Villafiorita, "The FSAP/NuSMV-SA Safety Analysis Platform," *International Journal on Software Tools for Technology Transfers (STTT),* vol. 9, no. 1, pp. 5-24, 2006.

[35] B. Bittner, M. Bozzano and A. Cimatti, "Timed Failure Propagation Analysis for Spacecraft Engineering: The ESA Solar Orbiter Case Study," in *International Symposium on Model-based Safety Assessment*, Trento, Italy, 2017.

[36] M. Bozzano, A. Cimatti, J. P. Katoen, P. Katsaros, K. Mokos, V. Nguyen, T. Noll, B. Postma and M. Roveri, "Spacecraft Early Design Validation using Formal Methods," *Reliability Engineering and System Safety,* vol. 132, pp. 20-35, 2014.

[37] E. Alana, H. Naranjo, Y. Yushtein, M. Bozzano, A. Cimatti, M. Gario, R. de Ferluc and G. Garcia, "Automated generation of FDIR for the compass integrated toolset (AUTOGEF)," in *Data Systems in Aerospace (DASIA 2012)*, Dubrovnik, Croatia, 2012.

[38] X. Ge, R. F. Paige and J. A. McDermid, "Probabilistic Failure Propagation and Transformation Analysis," in *28th International Conference on Computer Safety, Reliability, and Security*, Hamburg, Germany, 2009.

[39] X. Ge, R. F. Paige and J. A. McDermid, "Analysing System Failure Behaviours with PRISM," in *4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion*, Los Alamitos, Singapore, 2010.

[40] M. Güdemann and F. Ortmeier, "Towards model-driven safety analysis," in *3rd International*

Workshop on Dependable Control of Discrete Systems, Saabrucken, Germany, 2011.

[41] M. Lipaczewski, S. Struck and F. Ortmeier, "SAML goes eclipse — Combining model-based safety analysis and high-level editor support," in *Second International Workshop on Developing Tools as Plug-Ins (TOPI)*, Zurich, Switzerland, 2012.

[42] F. Ortmeier, W. Reif and G. Schellhorn, "Deductive Cause-Consequence Analysis (DCCA)," in *Proceedings of the 16th IFAC World Congress, Jun 2006*, 2005.

[43] M. Lipaczewski, F. Ortmeier, T. Prosvirnova, A. Rauzy and S. Struck, "Comparison of modeling formalisms for Safety Analyses: SAML and AltaRica," *Reliability Engineering and System Safety,* vol. 140, pp. 191-199, 2015.

[44] L. S. Azevedo, D. J. Parker, M. D. Walker, Y. I. Papadopoulos and R. E. Araújo, "Assisted Assignment of Automotive Safety Requirements," *IEEE Software,* vol. 31, no. 1, pp. 62-68, 2014.

[45] T. Kelly, ""Are 'Safety Cases' working?" Published in Vol 17 Issue 2 of the Safety Critical Systems Club Newsletter," 2008. [Online]. Available: https://www-users.cs.york.ac.uk/~tpk/2008scsarticlekelly.pdf. [Accessed 12 11 2021].

[46] U. M. o. Defence, "JSP 430: Ship Safety Management System Handbook," HM Government, London, UK, 1996.

[47] A. C. W. Group, "Goal Structuring Notation Community Standard (Version 2)," January 2018. [Online]. Available: https://scsc.uk/scsc-141B. [Accessed 12 11 2021].

[48] R. Wei, T. Kelly, X. Dai, S. Zhao and R. Hawkins, "Model Based System Assurance Using the Structured Assurance Case Metamodel," *Journal of Systems and Software,* vol. 154, pp. 211-233, 2019.

[49] S. Joba, "Are You Modelling: Visualizing Safety Cases - Tim Kelly on GSN (Goal Structuring Notation).," 23 2 2015. [Online]. Available: http://areyoumodeling.com/2015/02/23/gsn/. [Accessed 11 11 2021].

[50] R. Wei, T. Kelly, X. Dai, S. Zhao and R. Hawkins, "Model based system assurance using the structured assurance case metamodel," *Journal of Systems and Software,* vol. 154, pp. 211-233, 2019.

[51] I. Sljivo, B. Gallina, J. Carlson, H. Hansson and S. Puri, "A Method to Generate Reusable Safety Case Fragments from Compositional Safety Analysis," in *International Conference on Software Reuse 2015. In: Schaefer I., Stamelos I. (eds) Software Reuse for Dynamic Systems in the Cloud and Beyond. ICSR 2015. Lecture Notes in Computer Science, vol 8919. Springer, Cham*, Miami, USA, 2015.

[52] S. Mazzini, J. Favaro, S. Puri and L. Baracchi, "CHESS: an open source methodology and toolset for development of critical systems," in *3rd International Workshop on Open Source Software for Model Driven Engineering (OSS4MDE 2016)*, Saint Malo, France, 2016.

[53] B. Gallina, K. Lundqvist and K. Forsberg, "THRUST: a method for speeding up the creation of process-related deliverables," in *IEEE Digital Avionics Systems Conference (DASC'14)*, Colorado Springs, USA, 2014.

[54] N. Basir, E. Denney and B. Fischer, "Deriving safety cases for hierarchical structure in model-based development," in *International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Vienna, Austria, 2010.

[55] E. Denney and G. Pai, "Automating the Assembly of Aviation Safety Cases," *IEEE Transactions on Reliability,* vol. 63, no. 4, pp. 830-849, 2014.

[56] A. L. Oliveira, R. T. Braga, P. C. Masiero, Y. I. Papadopoulos, I. Habli and T. Kelly, "Supporting the automated generation of modular product line safety cases," in *Theory and Engineering of Complex Systems and Dependability: Proceedings of the Tenth International Conference on Dependability and Complex Systems*, Brunow, Poland, 2015.

[57] A. Retouniotis, Y. Papadopoulos, I. Sorokos, D. Parker, N. Matragkas and S. Sharvia, "Model-Connected Safety Cases," in *IMBSA'17: International Symposium on Model-Based Safety and Assessment. In: Bozzano M., Papadopoulos Y. (eds) Model-Based Safety and Assessment. IMBSA 2017. Lecture Notes in Computer Science, vol 10437. Springer, Cham.*, Trento, Italy, 2017.

[58] I. Sorokos, Generation of model-based safety arguments from automatically allocated safety integrity levels, Hull, UK: University of Hull, 2017.

[59] D. Schneider, M. Trapp, Y. Papadopoulos, E. Armengaud, M. Zeller and K. Höfig, "WAP: Digital dependability identities," in *2015 IEEE 26th International Symposium on Software Reliability*

Engineering (ISSRE), Gaithersbury, USA, 2015.

[60] J. Reich, D. Schneider, I. Sorokos, Y. Papadopoulos, T. Kelly, R. Wei, E. Armengaud and C. Kaypmaz, "Engineering of Runtime Safety Monitors for Cyber-Physical Systems with Digital Dependability Identities," in *International Conference on Computer Safety, Reliability, and Security. In: Casimiro A., Ortmeier F., Bitsch F., Ferreira P. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2020. Lecture Notes in Computer Science, vol 12234. Springer, Cham*, Lisbon, Portugal, 2020.

[61] J. Novet, "Everyone keeps talking about A.I.—here's what it really is and why it's so hot now," 17 6 2017. [Online]. Available: https://www.cnbc.com/2017/06/17/what-is-artificial-intelligence.html. [Accessed 2 12 2021].

[62] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development,* vol. 3, no. 3, pp. 210-229, 1959.

[63] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer and C. Barrett, "The Marabou Framework for Verification and Analysis of Deep Neural Networks," in *In: Dillig I., Tasiran S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science, vol 11561. Springer, Cham.*, New York, USA, 2019.

[64] S. Gerasimou, H. F. Eniser, A. Sen and A. Cakan, "Importance-driven deep learning system testing," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, Seoul, South Korea, 2020.

[65] J. Cameron, Artist, *The Terminator.* [Art]. Hemdale; Orion Pictures, 1984.

[66] X. Zhao, W. Huang, S. Schewe, Y. Dong and X. Huang, "Detecting Operational Adversarial Examples for Reliable Deep Learning," 13 4 2021. [Online]. Available: https://arxiv.org/abs/2104.06015. [Accessed 2 12 2021].

[67] A. Kurakin, I. Goodfellow and S. Bengio, "Adversarial examples in the physical world," 8 7 2016. [Online]. Available: https://arxiv.org/abs/1607.02533. [Accessed 2 12 2021].

[68] K. Lee, H. Lee, K. Lee and J. Shin, "Training Confidence-calibrated Classifiers for Detecting Out-of-Distribution Samples," 26 11 2017. [Online]. Available: https://arxiv.org/abs/1711.09325. [Accessed 1 12 2021].

[69] R. Salay, R. Queiroz and K. Czarnecki, "An analysis of ISO 26262: Using machine learning safely in autonomous systems," 7 9 2017. [Online]. Available: https://arxiv.org/abs/1709.02435. [Accessed 11 12 2021].

[70] X. Zhao, W. Huang, A. Banks, V. Cox, D. Flynn, S. Schewe and X. Huang, "Assessing the Reliability of Deep Learning Classifiers Through Robustness Evaluation and Operational Profiles," 2 6 2021. [Online]. Available: https://arxiv.org/abs/2106.01258. [Accessed 2 12 2021].

[71] C. Paterson, H. Wu, J. Grese, R. Calinescu, C. S. Păsăreanu and C. Barrett, "DeepCert: Verification of Contextually Relevant Robustness for Neural Network Image Classifiers," 2 3 2021. [Online]. Available: https://arxiv.org/abs/2103.01629v1. [Accessed 1 12 2021].

[72] C. H. Cheng, C. H. Huang and G. Nührenberg, "nn-dependability-kit: Engineering Neural Networks for Safety-Critical Autonomous Driving Systems," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Westminster, CO, USA, 2019.

[73] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri and M. Vechev, "AI2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[74] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang and M. Vechev, "DI2: Training and querying neural networks with logic," in *International Conference on Machine Learning, PMLR, 2019*, 2019.

[75] M. Mirman, T. Gehr and M. Vechev, "Differentiable Abstract Interpretation for Provably Robust Neural Networks," in *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, 2018.

[76] M. Muller, G. Makarchuk, G. Singh, M. Puschel and M. Vechev, "Precise Multi-Neuron Abstractions for Neural Network Certification," 5 3 2021. [Online]. Available: https://arxiv.org/abs/2103.03638v1. [Accessed 1 12 2021].

[77] X. She, P. Saha, D. Kim, Y. Long and S. Mukhopadhyay, "SAFE-DNN: A Deep Neural Network With Spike Assisted Feature Extraction For Noise Robust Inference," in *2020 International Joint*

Conference on Neural Networks (IJCNN), Glasgow, UK, 2020.

[78] K. Aslansefat, I. Sorokos, D. Whiting, R. T. Kolagari and Y. I. Papadopoulos, "SafeML: Safety monitoring of Machine Learning classifiers through statistical difference measures," in *7th International Symposium on Model-Based Safety and Assessment (IMBSA 2020); Proceedings in Springer Nature, Vol 12297, 2020, p197*, Lisbon, Portugal, 2020.

[79] M. Kläs and L. Jöckel, "A framework for building uncertainty wrappers for AI/ML based data driven components," in *International Conference on Computer Safety, Reliability, and Security; Springer, 2020, pp 315-327*, 2020.

[80] M. Kläs and L. Sembach, "Uncertainty Wrappers for data-driven models," in *International Conference on Computer Safety, Reliability, and Security; Springer, p 358-364*, 2019.

[81] N. Carlini, G. Katz, C. Barrett and D. L. Dill, "Provably Minimally-Distorted Adversarial Examples," 29 9 2017. [Online]. Available: https://arxiv.org/abs/1709.10207. [Accessed 3 12 2021].

[82] K. Aslansefat, S. Kabir, A. Abdullatif, V. Vasudevan and Y. Papadopoulos, "Toward Improving Confidence in Autonomous Vehicle Software: A Study on Traffic Sign Recognition Systems," *Computer,* vol. 54, no. 8, pp. 66-76, 2021.

[83] D. M. Powers, "Evaluation: From Precision, Recall, and F-Measure to ROC, Informedness, Markedness, and Correlation," *Journal of Machine Learning Technologie,* vol. 2, no. 1, pp. 37-63, 2011.

[84] M. Kläs, R. Adler, I. Sorokos, L. Joeckel and J. Reich, "Handling Uncertaintties of Data-Driven Models in Compliance with Safety Constraints for Autonomous Behaviour," *17th European Dependable Computing Conference (EDCC),* pp. 95-102, 2021.

[85] M. T. Ribeiro, S. Singh and C. Guestrin, ""Why should I trust you?" Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, USA, 2016.

[86] D. Slack, S. Hilgard, E. Jia, S. Singh and H. Lakkaraju, "Fooling LIME and SHAP: Adversarial attacks on post-hoc explanation methods," in *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 2020.

[87] S. M. Lundberg and S. I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, USA, 2017.

[88] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. Prutkin, B. Nair and S. I. Lee, "From local explanations to global understanding with explainable AI for trees," *Nature Machine Intelligence,* vol. 2, no. 1, pp. 56-67, 2020.

[89] Y. Shimada, Y. Hirata, T. Ikeguchi and K. Aihara, "Graph distance for complex networks," *Scientific Reports,* vol. 6, p. 34944, 2016.

[90] A. Abid, M. T. Khan and J. Iqbal, "A review on fault detection and diagnosis techniques: basics and beyond," *Artificial Intelligence Review,* vol. 54, pp. 3639-3664, 2020.

[91] B. G. Buchanan and E. H. Shortliffe, Rule-based Expert Systems: the Mycin Experiments of the Stanford Heuristic Programming Project, New York, USA: Addison-Wesley, 1984.

[92] W. R. Nelson, "REACTOR: An expert system for diagnosis and treatment of nuclear reactors," in *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, USA, 1982.

[93] A. Nairac, N. Townsend, R. Carr, S. King, P. Cowley and L. Tarassenko, "A system for the analysis of jet engine vibration data," *Integrated Computer-Aided Engineering,* vol. 6, no. 1, pp. 53-66, 1999.

[94] L. Felkel, R. Grumbach and E. Saedtler, "Treatment, Analysis, and Presentation of Information about Component Faults and Plant Disturbances," in *Symposium on Nuclear Power Plant Control and Instrumentation (IAEA-SM-266/40), p340-347*, UK, 1978.

[95] C. Lee, R. L. Alena and P. Robinson, "Migrating fault trees to decision trees for real time fault detection on the International Space Station," in *IEEE Aerospace Conference*, Big Sky, USA, 2005.

[96] J. B. Dugan and T. Assaf, "Diagnosis based on reliability analysis using monitors and sensors," *Reliability Engineering & System Safety,* vol. 93, pp. 509-521, 2008.

[97] M. Kramer and F. E. Finch, "Fault Diagnosis of Chemical Processes," in *Knowledge-based system*

diagnosis supervision and control, New York, USA, Plenum Press, 1989, pp. 249-263.

[98] N. H. Ulerich and G. Powers, "Online Hazard Aversion and Fault Diagnosis in Chemical Processes," *IEEE Transactions on Reliability,* vol. 37, no. 2, pp. 171-177, 1988.

[99] S. Guarro and D. Okrent, "The logic flowgraph: a new approach to process failure modelling and diagnosis for disturbance analysis applications," *Nuclear Technology,* vol. 67, pp. 348-359, 1984.

[100] M. Yau, S. Guarro and G. Apostolakis, "Demonstration of the Dynamic Flowgraph methodology using the Titan II space launch vehicle digital flight control system," *Reliability Engineering & System Safety,* vol. 49, no. 3, pp. 335-353, 1995.

[101] L. W. Chen and M. Modarres, "Hierarchical Decision Process for Fault Administration," *Computers and Chemical Engineering,* vol. 16, pp. 425-448, 1992.

[102] J. deKleer and B. Williams, "Diagnosis of Multiple Faults," *Artificial Intelligence,* vol. 32, no. 1, pp. 97-130, 1987.

[103] B. J. Kuipers, "Qualitative Simulation," *Artificial Intelligence,* vol. 29, pp. 289-338, 1986.

[104] P. Bunus and K. Lunde, "Supporting model-based diagnostics with equation-based object oriented languages," in *2nd International Workshop on Equation-based Object Oriented Languages & Tools (EOOLT)*, Paphos, Cyprus, 2008.

[105] K. Lunde, R. Lunde and B. Munker, "Model-based failure analysis with RODON," in *Proceedings of ECAI 2006 - 17th European Conference on AI*, Riva del Garda, Italy, 2006.

[106] Y. Zhou, J. Hahn and M. S. Mannan, "Process monitoring based on classification tree and discriminant analysis," *Reliability Engineering & System Safety,* vol. 91, pp. 546-555, 2006.

[107] J. Eggert, "Risk estimation for driving support and behavior planning in intelligent vehicles," *at - Automatisierungstechnik,* vol. 66, no. 2, p. 119–131, 2018.

[108] M. Machin, J. Guiochet, H. Waeselynck, J.-P. Blanquart, M. Roy and L. Masson, "SMOF: A Safety Monitoring Framework for Autonomous Systems," *IEEE Trans. Syst. Man Cybern, Syst. (IEEE Transactions on Systems, Man, and Cybernetics: Systems),* vol. 48, no. 5, p. 702–715, 2018.

[109] C. Pek, S. Manzinger, M. Koschi and M. Althoff, "Using online verification to prevent autonomous vehicles from causing accidents," *Nat Mach Intell (Nature Machine Intelligence),* vol. 2, no. 9, pp. 518-528, 2020.

[110] S. Shalev-Shwartz, S. Shammah and A. Shashua, "On a Formal Model of Safe and Scalable Self-driving Cars," Intel/Mobileye, http://arxiv.org/pdf/1708.06374v5, 2017.

[111] M. Trapp, D. Schneider and G. Weiss, "Towards Safety-Awareness and Dynamic Safety Management," in *14th European Dependable Computing Conference (EDCC)*, 2018.

[112] C. Hartsell, S. Ramakrishna, A. Dubey, D. Stojcsics, N. Mahadevan and G. Karsai, "ReSonAte: A Runtime Risk Assessment Framework for Autonomous Systems," in *16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2021.

[113] S. Khastgir, H. Sivencrona, G. Dhadyalla, P. Billing, S. Birrell and P. Jennings, "Introducing ASIL inspired dynamic tactical safety decision framework for automated vehicles," in *IEEE Intelligent Transportation Systems Conference (ITSC)*, 2017.

[114] R. Johansson and J. Nilsson, "The need for an environment perception block to address all ASIL levels simultaneously," in *IEEE Intelligent Vehicles Symposium*, Gotenburg, Sweden, 2016.

[115] P. Feth, "Dynamic Behavior Risk Assessment for Autonomous Systems," Germany, 2020.

[116] M. Schreier, V. Willert and J. Adamy, "An Integrated Approach to Maneuver-Based Trajectory Prediction and Criticality Assessment in Arbitrary Road Environments," *IEEE Trans. Intell. Transport. Syst. (IEEE Transactions on Intelligent Transportation Systems),* vol. 17, no. 10, pp. 2751-2766, 2016.

[117] S. Lefevre, C. Laugier and J. Ibanez-Guzman, "Intention-Aware Risk Estimation for General Traffic Situations and Application To Intersection Safety," [Research Report] RR-8379, INRIA, 2013.

[118] J. Dahl, G. R. d. Campos, C. Olsson and J. Fredriksson, "Collision Avoidance: A Literature Review on Threat-Assessment Techniques," *IEEE Trans. Intell. Veh. (IEEE Transactions on Intelligent Vehicles),* vol. 4, no. 1, pp. 101-113, 2019.

[119] S. Lefévre, D. Vasquez and C. Laugier, "A survey on motion prediction and risk assessment for intelligent vehicles," *ROBOMECH Journal 1,* pp. 1-14, 2014.

[120] L. Westhofen, C. Neurohr, T. Koopmann, M. Butz, B. Schütt, F. Utesch, B. Kramer, C. Gutenkunst

and E. Böde, "Criticality Metrics for Automated Driving: A Review and Suitability Analysis of the State of the Art," https://arxiv.org/abs/2108.02403, 2021.

[121] J. Reich and M. Trapp, "SINADRA: Towards a Framework for Assurable Situation-Aware Dynamic Risk Assessment of Autonomous Vehicles," in *16th European Dependable Computing Conference (EDCC)*, Munich, Germany, 2020.

[122] J. Reich, M. Wellstein, I. Sorokos, F. Oboril and K.-U. Scholl, "Towards a Software Component to Perform Situation-Aware Dynamic Risk Assessment for Autonomous Vehicles," in *Dependable computing - EDCC 2021 Workshops. DREAMS, DSOGRI, SERENE 2021*, 2021.

[123] M. Wellstein, "Development of a Bayesian Network for Situation-Aware Lane Change Prediction based on the highD Dataset," Technical University Kaiserslautern, 2021.

[124] D. Schneider and M. Trapp, "Conditional Safety Certification of Open Adaptive Systems," *ACM Trans. Auton. Adapt. Syst. (ACM Transactions on Autonomous and Adaptive Systems),* vol. 8, no. 2, pp. 1-20, 2013.

[125] D. Schneider, "Conditional Safety Certification for Open Adaptive Systems," Technical University Kaiserslautern, 2014.

[126] D. Schneider and M. Trapp, "A Safety Engineering Framework for Open Adaptive Systems," in *5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2011.

[127] D. Schneider and M. Trapp, "Engineering Conditional Safety Certificates for Open Adaptive Systems," in *IFAC Proceedings*, 2013.

[128] P. Feth and R. Adler, "Service-based Modeling of Cyber-Physical Automotive Systems: A Classification of Services," in *Matthieu Roy (Ed.): CARS 2016 - Critical Automotive applications : Robustness & Safety*, 2016.

[129] J. Reich, "Systematic engineering of safe open adaptive systems shown for truck platooning," Technical University Kaiserslautern, 2016.

[130] J. Reich, D. Schneider, I. Sorokos, Y. Papadopoulos, T. Kelly, R. Wei, E. Armengaud and C. Kaypmaz, "Engineering of Runtime Safety Monitors for Cyber-Physical Systems with Digital Dependability Identities," in *39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16-18, 2020*, 2020.

[131] S. Kabir, I. Sorokos, K. Aslansefat, Y. Papadopoulos, Y. Gheraibia, J. Reich, M. Saimler and R. Wei, "A Runtime Safety Analysis Concept for Open Adaptive Systems," in *Model-Based Safety and Assessment. 6th International Symposium, IMBSA*, 2019.

[132] W. Van der Aalst, "Data science in action," in *Process Mining*, Berlin, Germany, Springer, 2016, pp. 3-23.

[133] Y. Gheraibia, S. Kabir, K. Aslansefat, I. Sorokos and Y. Papadopoulos, "Safety + AI: A Novel Approach to Update Safety Models Using Artificial Intelligence," *IEEE Access,* vol. 7, pp. 135855-135869, 2019.

[134] S. Kabir, K. Aslansefat, I. Sorokos, Y. Papadopoulos and Y. Gheraibia, "A conceptual framework to incorporate complex basic events in HiP-HOPS," in *International Symposium on Model-based Safety and Assessment*, 2019.

[135] K. Aslansefat and G. Latif-Shabgahi, "A hierarchical approach for dynamic fault trees solution through semi-Markov process," *IEEE Transactions on Reliability,* vol. 69, no. 3, pp. 986-1003, 2019.

[136] K. Aslansefat, F. Marques, R. Mendonca and J. Barata, "A Markov Process-based approach for Reliability Evaluation of the Propulsion System in Multi-rotor Drones," in *Doctoral Conference on Computing, Electrical, and Industrial Systems*, 2019.

8. APPENDIX: USE CASE INVESTIGATIONS

This appendix provides a brief overview of some of the ongoing investigations into how to apply the techniques & technologies described in this document as part of some of the SESAME use cases.

8.1 LOCOMOTEC HOSPITAL DISINFECTION ROBOT USE CASE

As part of our investigations into the Locomotec use case, Hull and Fraunhofer IESE have been experimenting with SafeML for person detection, thus far with limited success. However, even failures can teach valuable lessons and through this process we have identified several weaknesses in the SafeML approach and have begun to address them. For example, new Tensor-based SafeML implementations have been developed that can be run on GPUs to achieve much higher performance.

This work has contributed to a recently published paper (see below) in which we have improved SafeML using bootstrap-based p-Value calculation for ECDF-based statistical distance measures. In addition, SafeML results have been compared with Wilson Interval Uncertainty calculated by Uncertainty Wrappers.

SESAME Publication | SafeML II
Aslansefat, K., Kabir, S., Abdullatif, A., Vasudevan, V., & Papadopoulos, Y. (2021). Toward Improving Confidence in Autonomous Vehicle Software: A Study on Traffic Sign Recognition Systems. IEEE *Computer*, *54*(8), 66-76.

Another publication is also in preparation (see below) to discuss how SafeML results can be used for runtime reliability evaluation of ML classifiers with operational profiles.

SESAME Publication | SafeML for ML Reliability Evaluation
Aslansefat, K., Akram M. N., Walker, M., Sorokos, I., Schneider, D., Papadopoulos, Y. (to be published in 2022), Towards Assessing Run-time Reliability of Machine Learning Algorithms Under Operational Profile Uncertainty.

Regarding the person detection problem, Locometec has provided a set of videos, including cases of successful person detection and cases of person misdetection. We have re-implemented YOLOv4 to test with SafeML and measure the results. The following figure provides two sample frames from a video that was labelled as a misdetection. The right image shows that in an early frame where the camera adjusts its brightness, the YOLOv4 Algorithm has detected the Person with 76.9% confidence. The left image shows a frame from the middle part of the video where YOLOv4 has failed to detect the person. This represents one of the more challenging cases, where the brightness is low and the person is partially obscured by another object. Hull is developing a new version of SMILE (see Section 3.2.5.2) to experiment with explainability vs. perturbed brightness and also to improve SafeML to include the brightness shift detection in its calculation.
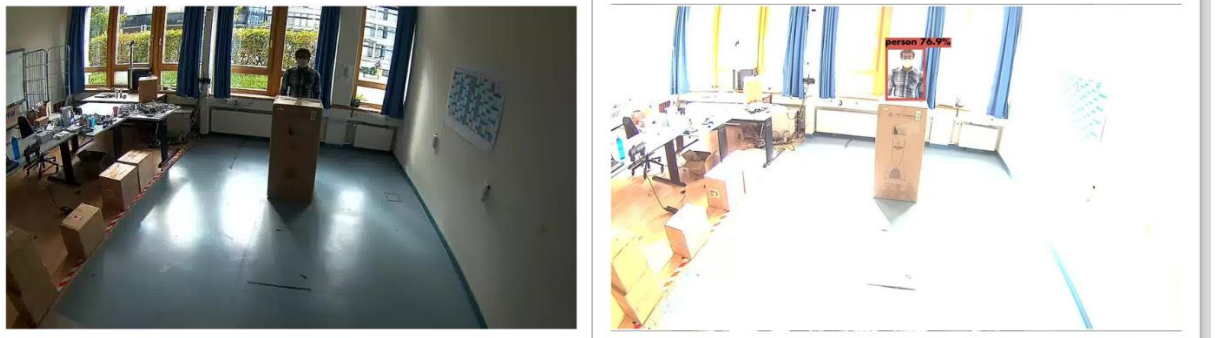
Figure 70 - Two sample frames from a video that LOCOMOTEC has labelled as misdetection – (Left) the person is not detected; (Right) The camera adjusts its brightness and the YOLOv4 Algorithm detects the person with 76.9% confidence

The next figure below illustrates the basic framework of SafeML as applied to the use case. First the embedded camera(s) capture an image from the scene. In the next step the software inside the robot loads the captured images and does the pre-processing. The YOLOv4 algorithm is used for human/person detection. On top of the person detection algorithm, SafeML can be applied where the detected person will be compared statistically with a certified or trusted dataset that the algorithm has been trained with.

Based on the statistical distance measures and an embedded kernel function, confidence values will be generated. When the calculated confidence is very low, the robot will turn off the lamps and notify the operator to check the person detection. The decision of the lamps will be stored to improve the detection algorithm and to be added to the certified or trusted database. If the confidence is only just below the threshold, SafeML will ask for more frames to re-evaluate the results. When SafeML provides high confidence, the decision will be accepted, and the robot's software can direct the required actions automatically.
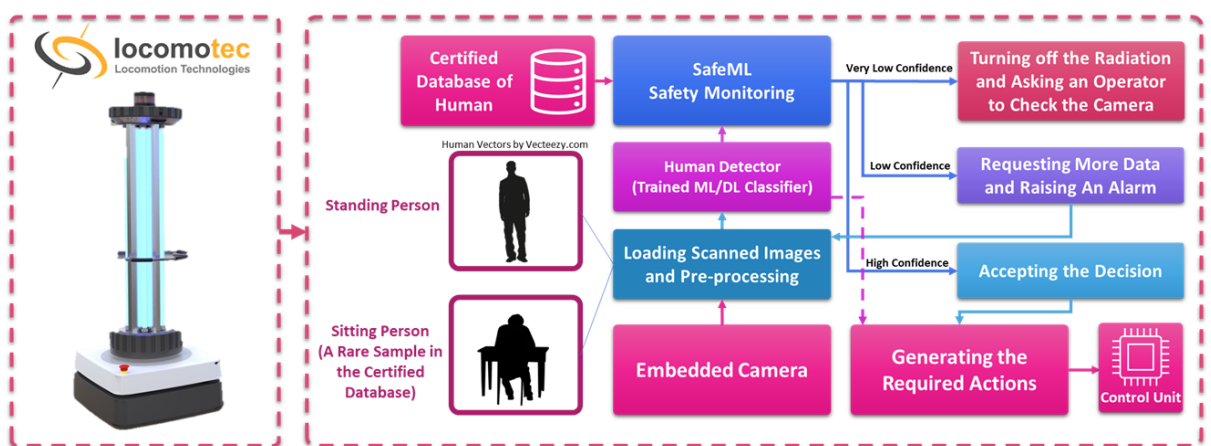

Figure 71 - The overall SafeML framework proposed for the LOCOMOTEC case study in the SESAME Project

Other than the above confidence evaluation using statistical distance measures, the new version of SafeML will also be able to evaluate the reliability of the YOLOv4 algorithm. The calculated reliability can later be integrated in the fault trees of the robot to enable overall reliability evaluation, taking into account both AI-based components and physical components.

This fault tree can be linked with symptoms (from event monitoring) and improved with Markov process based complex basic events (BEs). The following figure depicts a possible Markov process to be used as a complex BE in the robot's fault tree. Different top-level hazards can be considered; for example, a failure in one of the bulbs or switches can cause an area to be left infected (failure in disinfection). A reverse scenario can also be modelled where we ask the robot to turn off the UV-C lamps and it fails to do so. By adding small UV light detectors for each bulb, the robot could then provide evidence of the required symptoms. These symptoms determine the current state in the Markov model, and based on the current state, the probability of failure can be calculated to be used by the EDDI at runtime. It should be noted that the proposed model has been validated through Monte Carlo simulations and also algebraic calculations.
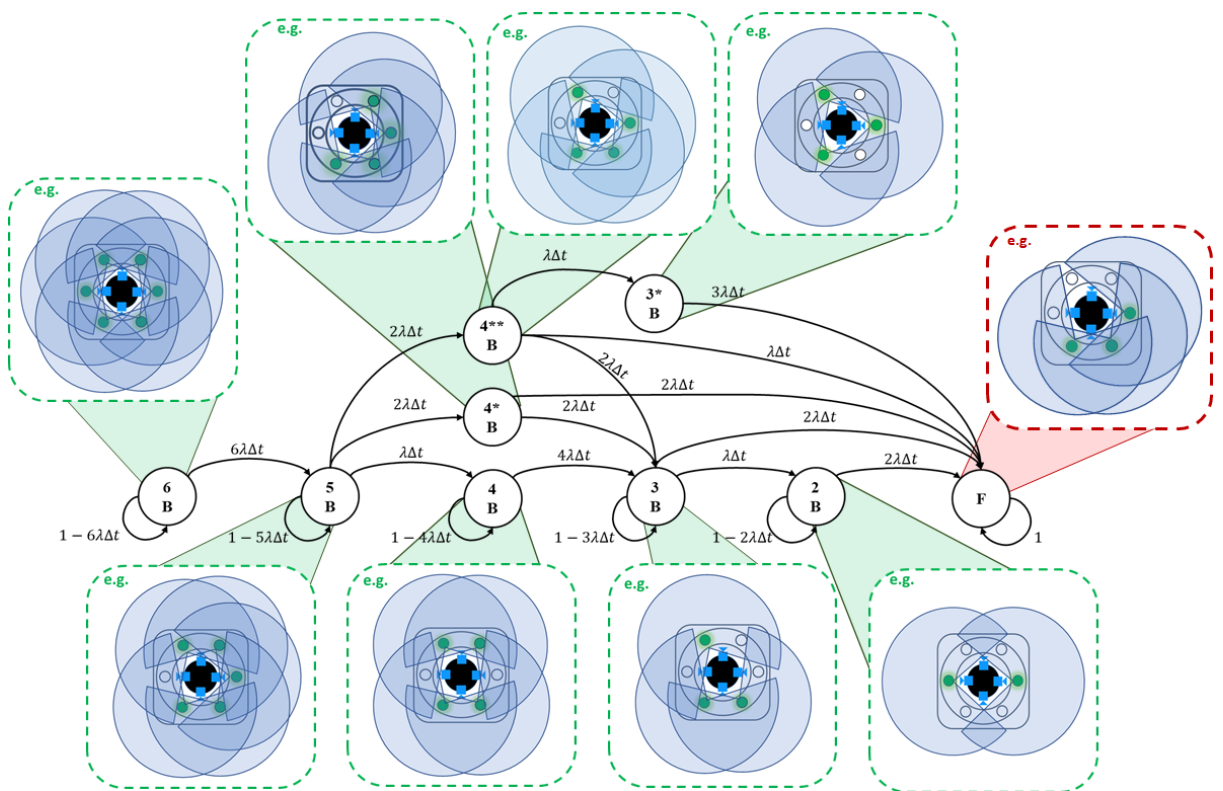

Figure 72 - The proposed Markov model for reliability evaluation of the robot's lamp and/or switch failures (note that each beam illustration for each state is representing just one possible scenario for that state and there are many other scenarios are not illustrated).

Confidentiality: Public Distribution

8.2   POWER STATION INSPECTION USE CASE

As part of our investigation into the Cyprus (power plant inspection) use case, the following section is intended to show how some of the technologies described in this report can merge to form part of an EDDI concept. Some of the ideas may also be applicable to the viticulture use case as well.

Consider a scenario in which three UAVs are tasked to scan three areas in parallel when an internal fault occurs in UAV #3. Although it does not cause immediate failure, the onboard EDDI can be used to re-evaluate the reliability at runtime and determine whether it increases the risk above a dangerous threshold. In order to reduce the risk of collision (or e.g. spraying out of bounds), UAV #3 activates a fail-safe mechanism and returns to base. If UAV #2 has the required dependability (quantitative values) and can offer guarantees of such, its mission can then be updated to cover the third area. The following figure illustrates this using two scenarios: scenario 1 when no fault has occurred yet and scenario 2 when an internal fault has occurred in UAV #3.
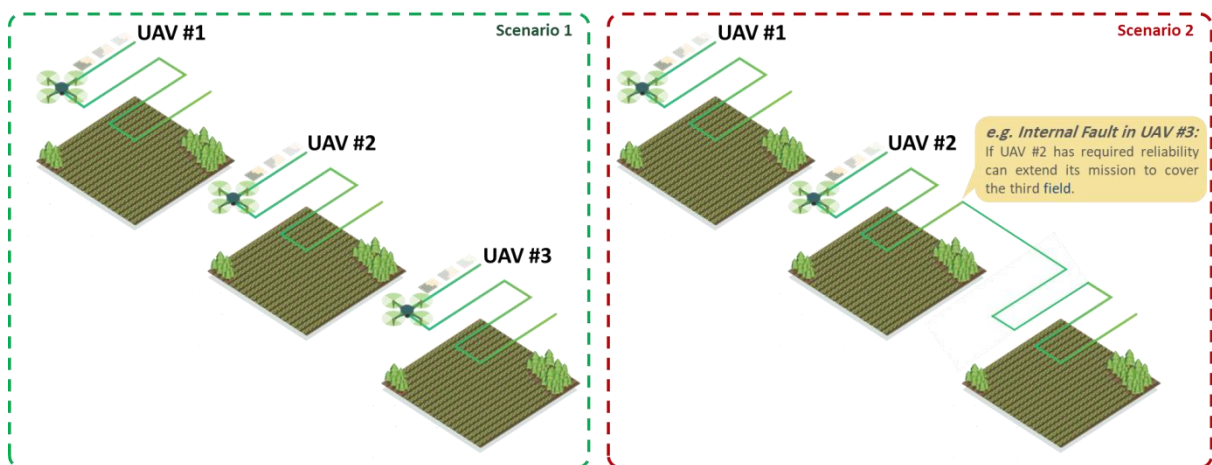

Figure 73 - Three UAVs are tasked to scan three fields in parallel; (Left figure) Scenario 1 when there is no fault occurred, (Right figure) scenario 2 when an internal fault occurs in UAV #3.

8.2.1   The Challenge of Complexity

To address the challenge on Complexity in this case study, both Hull and Fraunhofer offer MBSA tools like HIP-HOPS, Dymodia, and safeTbox to help experts provide a hierarchical failure model for each robot (i.e., UAVs in this case) as described in Section 2.

Hull has proposed the following fault tree model for each drone and are collaborating with KIOS to improve the fault tree and add more detail. This fault tree has different intermediate events such as communication failure, navigation failure, computer system failure, and propulsion system failure. The basic events are divided into two main categories:

- Simple basic events (BEs) that have an equivalent two-state Markov chain

- Complex basic events that have multi-state Markov chain like the propulsion system

For more information about fault trees with complex BEs, consult [134] and [135].
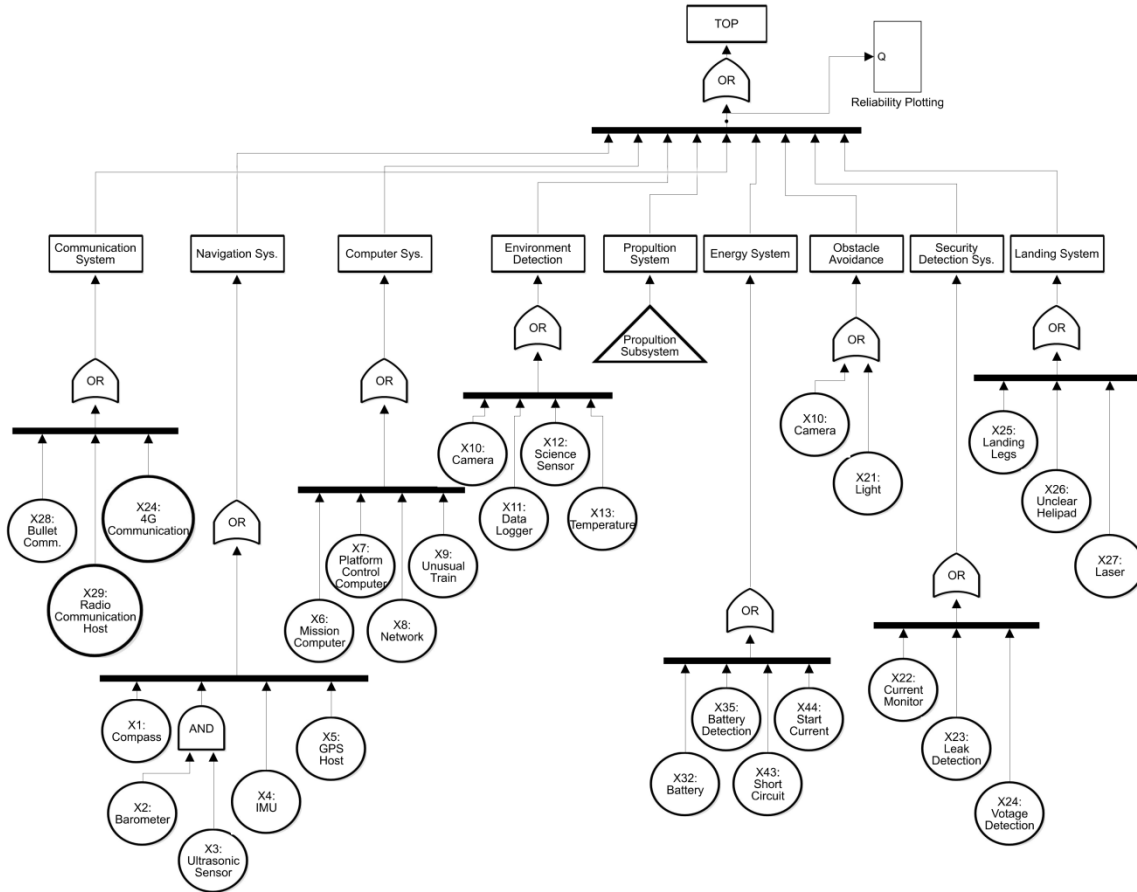

Figure 74 - UAV fault tree

As an example of a complex BE, consider a hexa-copter with identical rotors and PNPNPN configuration (P stands for clockwise rotation and N stands for anticlockwise rotation). It can be modelled with Markov chain as described in the figure below (for more information regarding the model construction please check [136]).
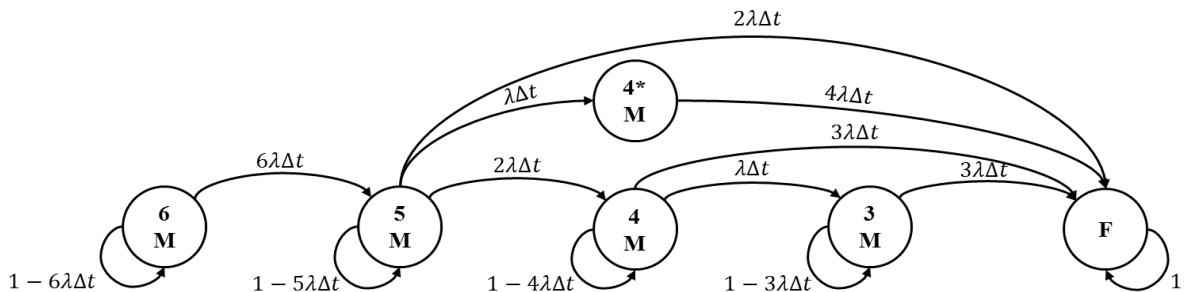

Figure 75 - A simplified Markov model of hexa-copter with identical rotors and PNPNPN configuration.

It is then necessary to have safety experts identify probabilistic links between system failure symptoms and different Markov states in each BE (simple or complex BEs), as illustrated below.
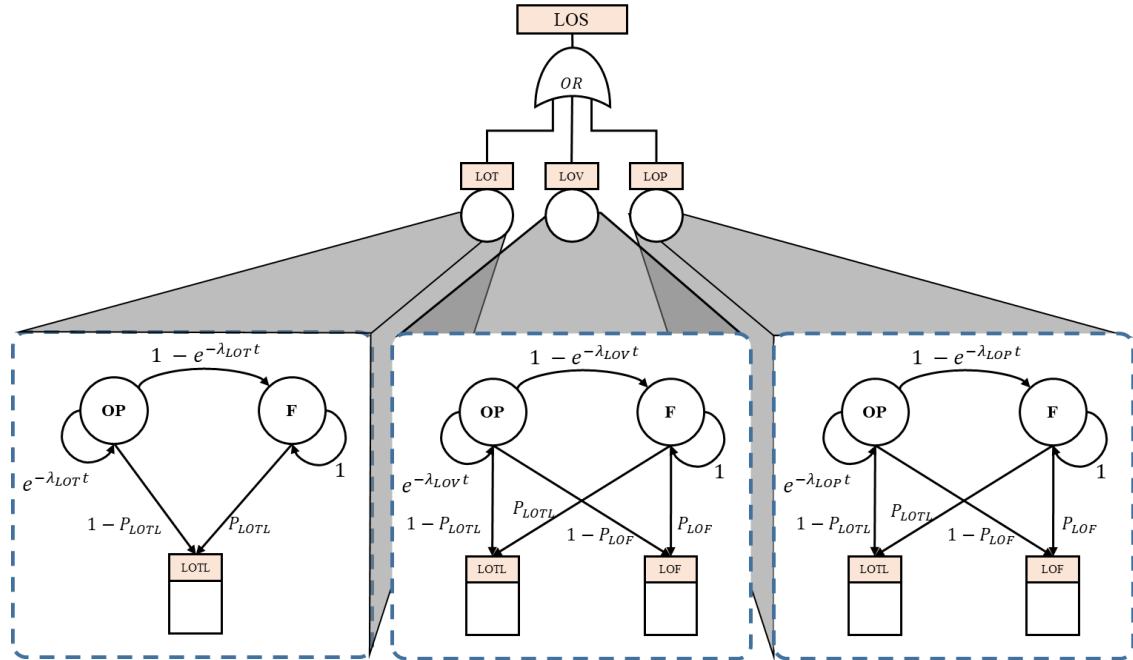
Figure 76 - Basic Event in FT and their connection to failure symptoms

8.2.2 The Challenge of Intelligence

As for intelligence, Uncertainty Wrappers, SafeML, and SMILE can all be applied to measure the reliability of AI-based software components as described in Section 3. It should be noted that SafeML has potential to provide a runtime reliability evaluation for the robots. Thus, the reliability of AI-based software components can be updated continuously during the mission and can form the basis of a basic event in the fault tree shown earlier.

For an overview of the potential framework, observe the figure below. In the top-right, the fault tree of the robot can be created using expert knowledge and MBSA tools like HiP-HOPS, linked to the failure symptoms. In addition, in the lower-right part of the figure, the generated model can be checked with ML algorithms both offline and at runtime to make sure it has the right logic as part of a possible model validation & repair mechanism (see Section 4.2.4).
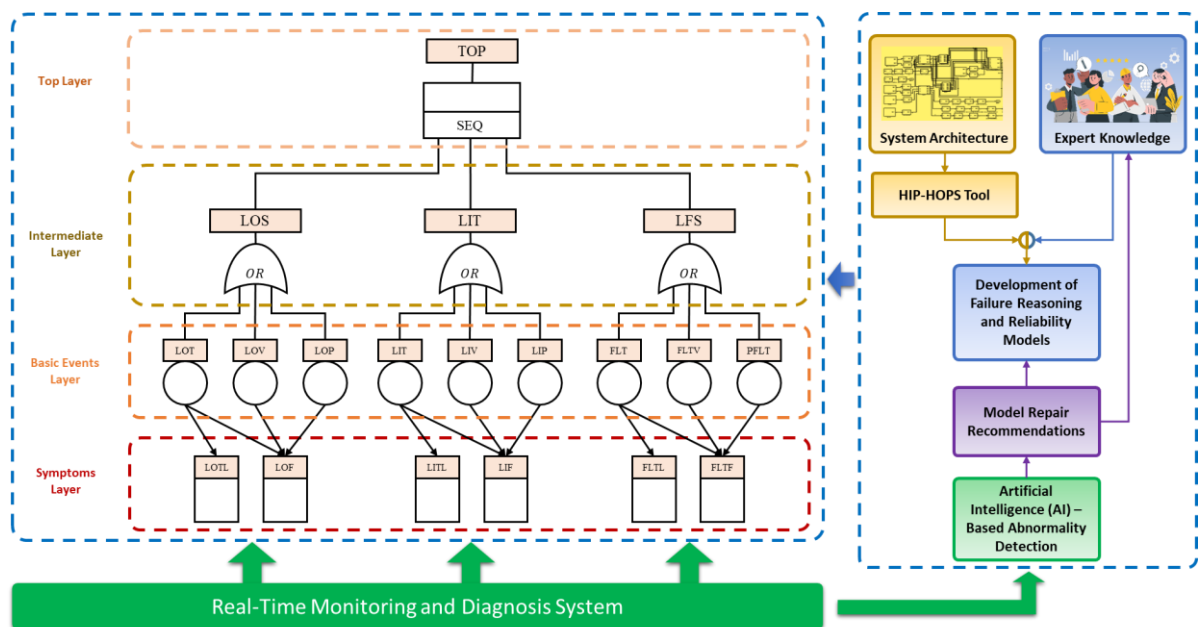
Figure 77 - An overview of the FT framework with symptoms and model repair

8.2.3 The Challenge of Autonomy & Openness

The generated fault tree, the Markov chains of the simple and complex BEs, and their links to the failure symptoms can all be stored as part of an EDDI and thus shared as necessary between all active robots. The EDDI can read the stored safety model at runtime and re-evaluate the robot reliability estimates during operation.

To illustrate this with an example, consider the same PNPNPN hexa-copter as earlier. An API called "Motor_Failure_Risk_Calc" can be used to be called inside the executable component of the EDDI onboard the robot. In this Python API, the first input is "MotorStatus" that can be 0 for failed and 1 for operational for each rotor (a vector of six digits). For instance, for a fully operational hexa-copter, the MotorStatus would be [1,1,1,1,1,1]. A MotorStatus like [0,1,1,1,1,1] means a motor has failed.

The second input is # Motors_Configuration. It can be 'PNPN' for quadcopters, 'PNPNPN' and 'PPNNPN' for hexa-copters and 'PPNNPPNN' for octa-copter.

The third input is Lambda, i.e. the failure rate of the Propulsion System including rotors, motors' drivers and propellers.

The final input is the time or system clock or mission current time. In the following figure, based on symptoms the drone's controller can say that all motors are healthy and send [1,1,1,1,1,1] as MotorStatus (e.g. in ArduPilot this information can be obtained from Dataflash and tlogs[40]).

Having obtained the MotorStatus, motors configuration, failure rate and mission time, the function calculates the probability of failure and MTTF. Based on the output, the reliability of the robot can be re-evaluated during the mission by the EDDI's high-level

---

[40] https://ardupilot.org/copter/docs/common-diagnosing-problems-using-logs.html

reasoning component. In addition, this information could be used by a ConSert to determine robot- or system-level safety guarantees (as discussed in Section 4).
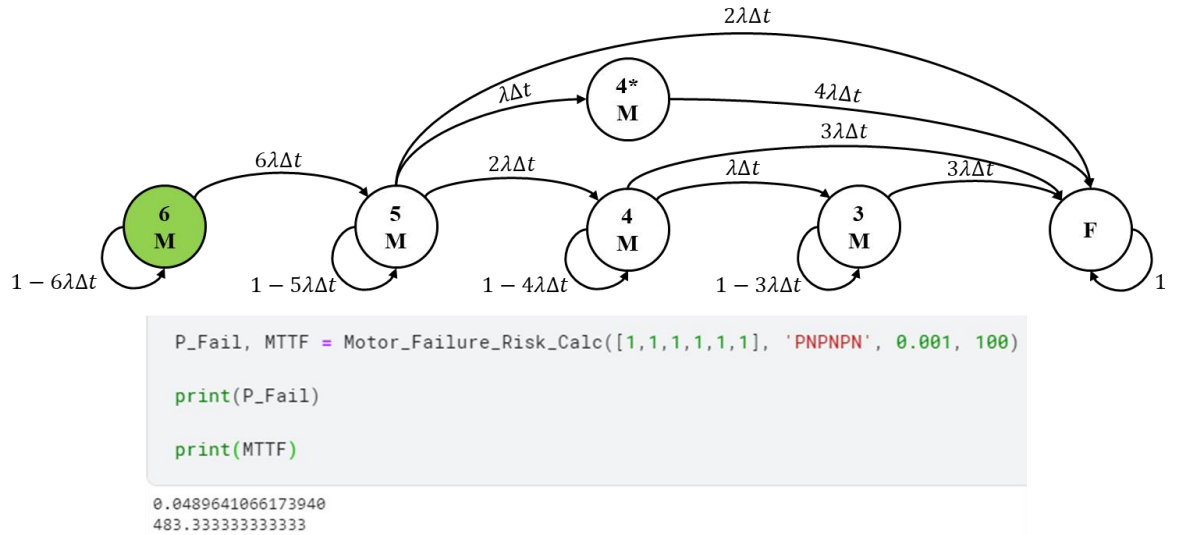


```
P_Fail, MTTF = Motor_Failure_Risk_Calc([1,1,1,1,1,1], 'PNPNPN', 0.001, 100)

print(P_Fail)

print(MTTF)
```

```
0.0489641066173940
483.333333333333
```

Figure 78 - Markov model of a hexa-copter propulsion system with identical rotors and PNPNPN configuration. The Python API calculates the probability of failure and MTTF of the drone.

In case of a rotor failure, the MotorStatus can be changed and the probability of the failure will also change accordingly as shown in the following figure. Based on the probability of failure, the whole system reliability can be re-evaluated.


```
P_Fail, MTTF = Motor_Failure_Risk_Calc([0,1,1,1,1,1], 'PNPNPN', 0.001, 100)

print(P_Fail)

print(MTTF)
```
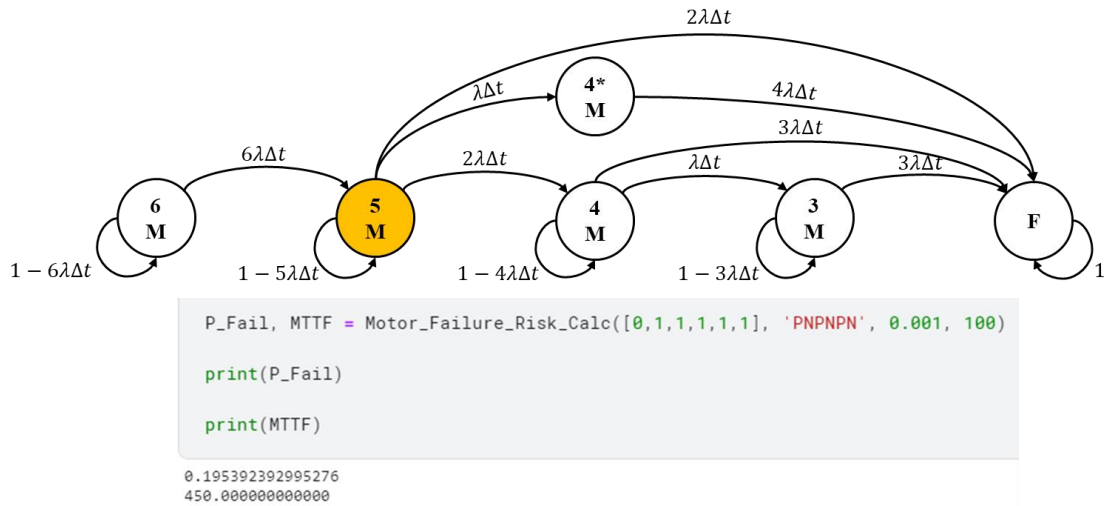
```
0.195392392995276
450.000000000000
```

Figure 79 - The same Markov model with one rotor failure. The Python API shows the updated probability and MTTF.