TU WIEN Informatics

# Semi-Automated Verification of Functional Safety Configurations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Technische Informatik**

eingereicht von

**Donald Acker, MSc**
Matrikelnummer 12202148

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn. Thomas Frühwirth, BSc
Mitwirkung: Dipl.-Ing.(FH) Dr.techn. Dieter Etz, MBA

Wien, 20. März 2025

| Donald Acker | Thomas Frühwirth |
|---|---|

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

# Semi-Automated Verification of Functional Safety Configurations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Computer Engineering**

by

**Donald Acker, MSc**
Registration Number 12202148

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn. Thomas Frühwirth, BSc
Assistance: Dipl.-Ing.(FH) Dr.techn. Dieter Etz, MBA

Vienna, March 20, 2025

| Donald Acker | Thomas Frühwirth |
|---|---|

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Declaration of Authorship

Donald Acker, MSc

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed. I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, March 20, 2025

Donald Acker

# Acknowledgements

First, I want to thank Thomas Frühwirth and Dieter Etz for their guidance, feedback, and support throughout this thesis. I am also grateful to Wolfgang Kastner and Valentin Just whose courses on automation introduced me to the field, and whose contagious passion for the subject led me directly into this project. I also want to thank all my other teachers at TU Wien, in whose classes I built the technical skills that made this paper possible. And, I owe special gratitude to the Fachschaft Informatik, whose VorlesungsWiki made it possible for me to learn enough informatics-related German to pass exams in my first year of admission-requirement classes: without this resource, I never would have gotten anywhere near writing my thesis.

Finally, and most of all, I want to thank my wife Theresa, who inspired me to shake up my life and follow my curiosity into a whole new academic degree, and who has supported me every step of the way with endless kindness, warmth, and love, and also with patience every time I have talked about ontology engineering and emergency stop buttons.

This work was conducted using Protégé [62].

# Kurzfassung

Die schnell rekonfigurierbare Fertigung ist ein Eckpfeiler des Industrie-4.0-Programms. Neben der flexiblen Anpassung an unsichere Marktsituationen kann es die Nachhaltigkeit einer Produktionsanlage verbessern, indem es die Umweltbelastung, den Abfall, den Energieverbrauch und den Verbrauch natürlicher Ressourcen minimiert. Die Verifizierung der funktionalen Sicherheit einer rekonfigurierbaren Fabrik stellt jedoch eine wesentliche Herausforderung dar, da herkömmliche Sicherheitsverfahren auf umfangreichen, manuellen Überprüfungen durch Sicherheitsingenieure basieren. Diese Dissertation wendet eine Design-Science-Research-Methodologie an, um einen beschleunigten Ansatz zu entwickeln: einen Prozess zur semi-automatisierten Verifizierung, bei dem automatisiertes Schließen die Konformität eines Modells eines industriellen Systems mit seinen Sicherheitsanforderungen überprüft. Der Prozess beginnt mit der Erstellung eines Systems Modeling Language (SysML)-Modells des Systems und seiner Anforderungen, die anschließend in eine Web Ontology Language (OWL)-Wissensbasis übersetzt werden, um dem automatisierten Schließen zugänglich zu sein. Schließlich wird ein Satz von Semantic-Web-Schlussfolgerungssprachen, verwendet, um die Sicherheitsanforderungen zu verifizieren. Als durchgängiger Proof-of-Concept wird dieser Prozess auf eine beispielhafte Produktionseinheit angewendet. Die Evaluierung des Prozesses zeigt, dass, obwohl die manuellen Systemmodellierungsschritte eine vollständige Automatisierung behindern, die Automatisierung nach der Investition in die Modellierung der rekonfigurierbaren Fabrik die Sicherheitsverifizierung jeder Konfiguration erheblich unterstützt.

# Abstract

Rapidly reconfigurable manufacturing is a pillar of the Industrie 4.0 program. In addition to flexibly adapting to uncertain market situations, it can improve a production facility's sustainability by minimizing the environmental impact, waste, energy consumption, and use of natural resources. However, verifying the functional safety of a reconfigurable factory is a major challenge, since standard safety practices rely on extensive manual reviews by safety engineers. This thesis applies a Design Science Research methodology to develop a faster way: a process for semi-automated verification, in which automated reasoning verifies the compliance of a model of an industrial system with its safety requirements. The process begins with the creation of a Systems Modeling Language (SysML) model of the system and its requirements, which are then translated into a Web Ontology Language (OWL) knowledge base in order to be amenable to automated reasoning. Finally, a set of Semantic Web reasoning languages are used to verify the safety requirements. As an end-to-end proof of concept, this process is applied to an example production unit. Evaluation of the process shows that while the manual system-modeling steps hold it back from full automation, once the time is invested in modeling the reconfigurable factory, the automation greatly assists in verifying the safety of any configuration.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

Industrie 4.0 promises that networked, rapidly reconfigurable automated factories can make manufacturing highly efficient [10]. However, established practices for manufacturing safety have little scope for reconfiguration: creating and validating a safety scheme takes a long time and a lot of highly specialized labor. That means that rapid reconfiguration would require decidedly un-rapid safety changes. To realize both functional safety and Industrie 4.0 rapid reconfiguration goals, it will be necessary to automate the creation and validation of safety measures.

Model-Based System Engineering (MBSE) offers a set of tools which could solve this problem: it is possible to include both safety requirements and safety systems in a system model [47]. Theoretically, this would allow for automated verification of the safety conditions on the system. Extending this system model to a digital twin of a real-world system would thus allow for automated verification of safety conditions in a shorter amount of time. A further, implicit benefit here is the possibility of including safety alongside systems design, rather than as a second stage [60].

To begin work on this, researchers have developed conceptual frameworks for automating the functional safety process. Some have applied these frameworks in limited example facilities as well. However, at present, all have been forced to subjectively interpret relevant safety regulations in the contexts of their examples in order to create requirements that their system models can parse. In practice, this manual step is a severe bottleneck, preventing any practical gains from automation.

Etz et al. identify a first step towards bridging this gap: in addition to the configuration of the system model, it is necessary to have a knowledge base encoding knowledge about both the manufacturing system and the regulatory requirements, represented in a manner that allows for automated reasoning on the knowledge [31]. Following from this work,

Reitgruber has developed a prototype knowledge base for manufacturing systems [66]. The other half remains open: developing a formal representation of safety requirements. With this in place, it should be possible to apply automated reasoning to a system model and verify that it satisfies safety requirements.

## 1.2 Research Question

In this work, we demonstrate a process for automatically verifying the compliance of an industrial system with its safety requirements. We will produce this demonstration as we explore our main research question: **How can we automatically verify that an industrial system is consistent with all of its safety requirements?**

We should immediately recognize: functional safety is a property of the real physical system, while we can only perform automated verification on a model. Thus, this top-level research question can be broken into two related sub-questions:

- How can we effectively model an industrial system for the purpose of verifying functional safety requirements?
- How can we effectively use automated reasoning to verify a system model's consistency with its safety requirements?

We make special note here of two semantic separations: keeping these concepts distinct is necessary if our discussion is to be precise at each point.

First, we draw a distinction between our subject industrial system (a production unit in a factory) and the *model* of our system. The industrial system is "the real thing", or at least, "thing itself", while the model is a *structured simplifying representation* of this thing. The distinction is tricky here since we will not be working from an actual physical factory in the real world, but from an imaginary factory which is itself already quite simplified. However, the conceptual distinction remains: our system model *represents* the subject industrial system, and any automated reasoning solution can be applied to the *model* (which is made of 0s and 1s), not to the system itself (which is made of steel and plastic, or at least of pictures and narrative descriptions).

Our second semantic separation concerns the scope of the subject *system*. Our industrial system comprises physical objects (rooms, machines, and so on), as well as configurations (in wiring or software) and also requirements (in our case, derived from safety standards). When we write "system model", we are referring to the highest-level concept of our industrial system, encompassing all elements which are relevant to our inquiry. Elements which are not relevant, such as external stakeholders, financial details, and so on are omitted for practical reasons, though in theory they would be present at this level of abstraction. This leads to a careful choice of language: as in the research questions above, we will say "the system is consistent with its own safety requirements", not, "the

system meets the requirements". This is a recognition of the fact that the requirements are a part of the system. This philosophical choice becomes practical when we enter the system-modeling process: physical and configuration elements of the system have direct links to requirement elements, and changes in one realm interact with changes in another. So we are careful to maintain this understanding: the requirements are part of the system.

## 1.3 Methodology

To explore our research questions, we will follow a design science research methodology. We adopt the six-step cycle described by vom Brocke et al. in "Introduction to Design Science Research" [25].

1. Problem identification and motivation: We refine the above research question into a clear problem statement about checking safety requirements using automated reasoning.
2. Define the criteria for a solution: In addition to a set of criteria for our artifact (this chapter), we will develop criteria for each of the major decisions we make along the way:
   a) Choice of scenario: What representative example of an industrial system will we choose, to which we may apply our modeling and verification processes? (Chapter 3)
   b) Choice of system modeling approach: What technologies and strategies will we use in our system-modeling process? (Chapter 4)
   c) Choice of verification approach: What automated reasoning technologies and strategies will we use in our verification process? (Chapter 5)
3. Design and development: We will create each of the components needed for our ultimate artifact:
   a) An example scenario to which we may apply our modeling and verification processes, presented in narrative descriptions and pictures.
   b) A process for creating a system model of our scenario.
   c) A system model of our scenario, in formal modeling language(s).
   d) A process for automatically verifying that our system model is consistent with its safety requirements.
   e) A report confirming the system's compliance or else precisely identifying its shortcomings.
4. Demonstration: We will demonstrate our processes on multiple configurations of the model scenario.

5. Evaluation: Based on the objectives we have created, we will evaluate our artifact.

6. Communication: We present our work in this thesis paper.

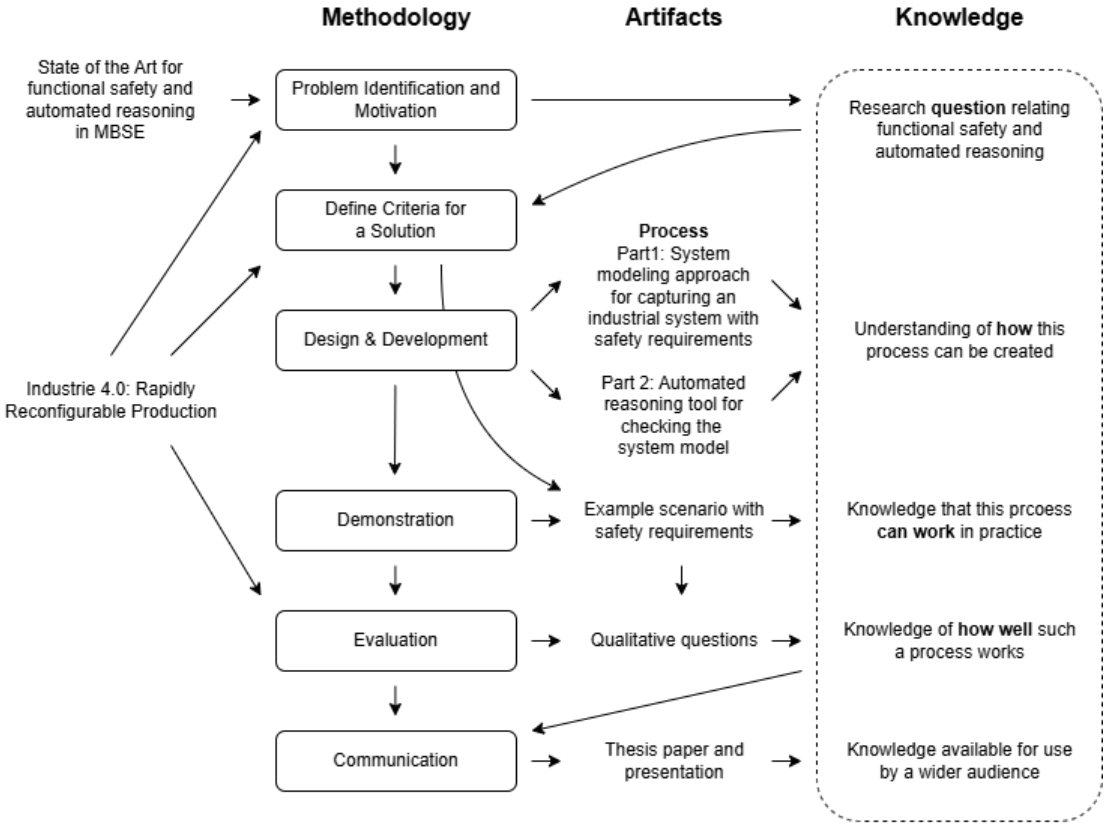


Figure 1.1: Design Science Reseach methodology, as adapted for use in this thesis from [25]

We can see in Figure 1.1 that "thesis paper and presentation" appears only as the tail end of the process. As a result, the iterative loops under "design and development" which actually comprise most of the work are hidden from the reader. However, the reality of iterative design guides certain high-level choices in this thesis. For example, while the SPARQL Protocol and RDF Query Language (SPARQL) queries in Apache Jena are used to check the safety requirements, this came only after some days of effort attempting to use other solutions involving the Semantic Query-Enhanced Web Rule Language (SQWRL), the Shapes Constraint Language (SHACL), and Snap-SPARQL. These unsuccessful efforts are not presented alongside the successful solution, but are nonetheless a core part of following the methodology. We summarize the reasons that these other technologies were unsuccessful in "Choice of Reasoning Language".

Of particular note: we will spend significant effort producing a top-level system model which encompasses our requirements. This system model may at first seem irrelevant in

our final product. However, it is a critical part of our process for system modeling, as it allows us to carefully track the effects each iterative change we make: as we find we need to update a requirement or rethink how a system element is modeled, the system model helps us trace the impact of these changes and maintain consistency throughout our illustration of our process.

## 1.4 Artifact

Our methodology will result in the production of an *artifact*: **a process for evaluating an industrial system for consistency with its safety requirements**. It is helpful to view the process in two parts. The first is a process for system modeling. The second is a process for automated verification. Linking these two processes is an intermediate artifact: a particular system model, which our first process produces and our second process consumes. We illustrate this in Figure 1.2.

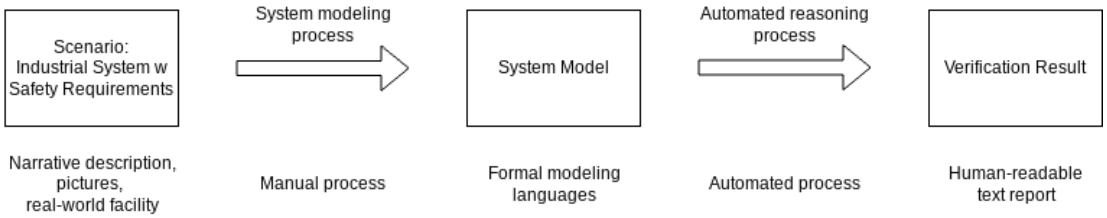


Figure 1.2: Process for evaulating an industrial system for consistency with its safety requirements

It is this overall process which forms the response to our research question, and so it is by examining this that we will evaluate the success of our project.

This diagram should make clear the somewhat qualified title of this thesis: our process is *semi-automated*, with a manual first part and an automated second part. The title may also be viewed as a concession to the ultimate truth that safety verification, at the highest level, cannot be fully automated, as it is ultimately human engineers and executives who must take responsibility for the safety of their systems.

## 1.5 Criteria for a Successful Artifact

Following our division of our artifact into two subprocesses, we distill our aims for the project into two sets of criteria.

A successful **modeling process**:

1. Allows the modeler to accurately reflect the target system.
2. Unites physical and configuration elements as well as requirements.

3. Allows for iterative change as the target system or the modeler's understanding of it changes.
4. Makes use of existing technologies in order to reduce the modeler's effort.
5. Makes use of existing modeling languages in order for different modelers' separate work to be mutually intelligible and functionally compatible.

All of these except for the second are essential criteria for *any* modeling process in any domain. However, that second criteria has a more specific relevance to this thesis: since we are performing verification of safety configurations, it is necessary that our model encompass all three of these domains: physical devices, configurations, and safety requirements,

A successful **verification process**:

1. Uses automated reasoning tools to ensure correct results.
2. Detects common human errors from the modeling stage.
3. Produces a clear answer, either affirming the system's consistency or specifically identifying each fault.
4. Provides human-readable explanations for each fault.
5. Can be conveniently applied to multiple test cases or to entirely different system models.

While these are mostly self-evident, the last demands some clarification. Since our goal is to verify safety in order to enable rapid reconfiguration, we need to be able to readily apply our process to new variations on the same model.

We will return to these criteria in Chapter 8.

## 1.6 Contributions of This Thesis

The design artifact comprises the main original contribution of this thesis. Existing work has explored the feasibility of various aspects of the safety verification problem. However, no existing work presents a precise end-to-end process which starts with an industrial system with its safety requirements and ends with a detailed verification result in human-readable form. This thesis provides such a process, and validates it against an example scenario.

The exact choice of which reasoning technology to use for which purpose is probably not entirely new, but is not represented in the literature. In particular, the approach to using the Semantic Web Rule Language (SWRL) and SHACL together to perform inference subject to closed-world conditions is original, if somewhat inelegant.

## 1.7 A Preview of the Thesis

With all of this in mind, we offer a brief overview of the thesis, to help the reader understand what each chapter is doing.

1. Introduction: We present our research question and the methodology with which we will explore it. We then present the criteria by which we will evaluate the artifact resulting from our exploration.

2. State of the Art: We give an overview of the current research publications related to our research question. Here, we look at what is being done, from a high level, rather than the details of how the technologies work.

3. Scenario: We present a narrative description of an idealized production unit and a simplified set of safety standards which this must meet. A detailed modeling process for this scenario, together with an automated process for verifying its safety compliance, will be the product of our exploration.

4. System Modeling: We introduce the main approaches to system modeling, evaluate their suitability to our purposes, and choose the specific modeling technologies we will use to create the detailed model of our scenario.

5. Automated Reasoning: We introduce the main approaches to automated reasoning in the context of our system model, evaluate their suitability to our purposes, and choose the specific reasoning technologies we will use to verify the safety compliance of our model.

6. Design Artifact: We present the result of our design methodology: our detailed step-by-step process for evaluating an industrial system for consistency with its safety requirements.

7. Proof of Concept: We apply the step-by-step process from Chapter 6 to four different configurations of the industrial system and safety requirements from Chapter 3.

8. Evaluation: We assess the performance of our modeling process and automated reasoning tool against the criteria we presented in the first chapter.

9. Conclusion: We discuss the implications of our results and potential directions for future work.

Let's go!

CHAPTER 2

# State of the Art

In this chapter, we examine the current state of the technologies we will use and the research that informs our exploration. This chapter focuses on the "what": what are current researchers accomplishing? Our detailed explanation of the "how" of our relevant technologies waits for Chapters 4 and 5.

## 2.1 Technical Background

**Functional safety** is the part of overall safety focused on automatic changes in the behavior of a machine or system to prevent injury. One example is the control circuit that prevents an automatic door from closing on a person standing in the doorway [11]. This example is relatively passive: a more active example would be a controller on a wind turbine that angles the blades into a neutral position if wind speed increases above a safe limit. A number of standards guide functional safety: the most prominent is IEC 61508, which describes whole-lifecycle safety for electrical and electronic systems. A modification of this, IEC 62061, describes functional safety of safety-related control systems on machinery [61]. A range of other standards may also be relevant, depending on the industry and the jurisdiction. Alongside standards, governments also mandate industrial safety measures, to varying extents. In the EU, the the Machinery Directive (MD) sets the legal requirements for safety in manufacturing processes [3]. The relationship between IEC standards and the MD is complicated. For example, compliance with IEC 61508 does not give the presumption of compliance with the MD [7], but IEC 62061 does [2]. But regardless, in practice, standards and legal requirements are closely related, both in their development and their application. So, the IEC standards, together with the MD, provide an important and representative example of a regulatory framework for functional safety.

**Model-Based System Engineering (MBSE)** is the use of system models to support requirements, analysis, verification, and validation throughout a design process [5]. The

essential goal of MBSE is to cope with the exponentially growing complexity of designed systems, in which component interactions, rather than component failures, become the primary causes of failure. The most common modeling tool for MBSE applications is the Systems Modeling Language (SysML): while it is not necessarily a standard, it is widely used and required in some contexts, such as US Department of Defense contracts [8]. In automation, AutomationML is another widely-used open standard [1]. There is increasing recognition that MBSE offers the possibility of solving safety engineering problems with which older safety methodologies are unable to cope. The question of how, exactly, to realize this is at the forefront of systems engineering research today. For example, in her keynote at the INCOSE International Workshop 2023 MBSE Workshop, Nancy Leveson outlined the challenges of modern safety engineering and argued that advances in MBSE offered a way past them [6].

**Knowledge engineering** is the use of representations of real-world knowledge in structured frameworks to enable automated reasoning [54]. In the context of functional safety and MBSE, knowledge models are necessary in order to "ask" whether a system (model) is safe–or, equivalently, in order to state what, exactly, it means for a system to be safe. Knowledge models are commonly structured as ontologies: "set[s] of representational primitives with which to model a domain of knowledge or discourse" [41]. The most prevalent tool for describing ontologies is the World Wide Web Consortium (W3C)'s the Web Ontology Language (OWL) [76]. However, any given ontology itself is generally domain-specific. In this work, we will extend Reitgruber's Reconfigurable Safety System Ontology, which is in turn developed from several domain-specific ontologies [66]. Attaching knowledge modeling to a MBSE workflow is the subject of current research in systems engineering [50] [9] [58] [4].

**Automated reasoning** is the use of software to check the truth of a statement [65]. In our context, we intend to form statements about the safety of our system model using our knowledge model, and then check the validity of such statements. While automated reasoning is, in full generality, an uncomputable problem, and is in virtually any practical application at least NP hard, practical approaches nonetheless exist. Most of these amount to reducing a given problem to a better-understood problem–boolean satisfiability is a common target–and then applying a well-developed solver for that problem–for example, Microsoft's open-source Z3 solver, for satisfiability. These approaches are mature enough to solve such otherwise-intractable problems as software library dependency resolution in practical amounts of time. In particular, it is possible to use automated reasoning on knowledge models and system models, as our goals require [53].

## 2.2 Related Work

Taking these four identified areas as a guide, we examine relevant current work on each topic. We examine each topic in turn, presenting a handful of relevant papers.

### 2.2.1 Functional Safety

We approach functional safety directly from the context of Industrie 4.0, a promotional label which has come to encompass a wide range of interconnected trends in the evolution of factories into cyber-physical systems [52]. The term is also politically specific: it identifies trends being actively advanced in Germany (as the spelling suggests) and, more broadly, in the EU. As Kagermann and Wahlster, two original proponents of the term, write in their 2022 review of the first decade of the phenomenon, "For the first time in the high-tech world, we have once again been able to establish an innovative concept from Germany internationally, after they had mostly come from North America or Asia for many years. Industrie 4.0 has made Europe the most innovative factory supplier of the world. There does not exist any 'smart factory' anywhere in the world where a large number of software and hardware components does not come from European companies." [52] As this originates in the German and EU context, it is natural that a strong emphasis on safety regulation has been a thread throughout the developments described as Industrie 4.0. At the same time, the move towards broad adoption of emerging technologies creates significant safety challenges.

Polak-Sopinska et al. give a comprehensive overview of the opportunities and challenges which Industrie 4.0 presents for safety in their 2020 paper "Impact of Industry 4.0 on Occupational Health and Safety" [64]. On the opportunities side, they identify a clear safety benefit in the simple fact that automated "lights off" factories running without humans present no injury risks during those hours. They identify a wider range of potential benefits: the greater use of sensors and intelligent technologies creates the possibility of more quickly and accurately detecting unsafe situations. They also see potential for "smart" personal protective equipment to reduce injuries. And they note that about a quarter of sick leave days in Germany result from lifting-related injuries, and highlight the potential for robotic coworking and even exoskeletons to reduce this source of injuries. At the same time, they identify significant challenges. At a high level, they note that qualified labor shortages likely to result from the increasing complexity of work coupled with an aging workforce would generally exacerbate safety risks. With a closer look at the functional safety of machines, they note that while AI solutions for factory management are a major aspect of Industrie 4.0, AI is essentially unable to cope with new situations–that is, with precisely the sort of unsafe situations which a system needs to able to mitigates. This essentially amounts to the black swan problem: all serious safety failures are previously-unknown until they happen, and the statistical approach of AI cannot detect them in advance. (This is very much the motivation for our desire to instead employ formal reasoning!)

Despite the wide range of challenges which the Industrie 4.0 transformation brings, there is so far only limited systematic research on this issue. In a 2023 review of the topic, Hutchins et al. perform a keyword- and citation-based analysis of existing publications, and find only the most limited treatment of the issue, recommending further research [46]. Much of the existing work on the topic focuses on high-level organizational approaches to safety management. For example, Torrecilla-García et al. produce a four-level conceptual

framework for classifying organizational readiness for implementing occupational safety and health in industries pairing products with services ("servitization") [74]. Junior et al. examine the question from an enterprise management point of view, identifying frameworks to support top-level development of functional safety in the automotive sector [51]. Meany directly addresses functional safety in his 2017 paper "Functional safety and Industrie 4.0" [59]. However, overall, all of these papers are preliminary and high-level.

One particularly relevant paper is Dieter Etz's dissertation "Flexible Safety Systems: Use Cases, Requirements, System Design, and Software Architecture": this thesis is one of eight thesis projects carried out under Etz's supervision, in conjunction with his project [30]. In contrast to the high-level, management-oriented approaches of the papers above, Etz lays out the architecture and requirements for "Flexible Safety Systems", a method by which factories could achieve functional safety while also achieving the key Industrie 4.0 goal of reconfigurable manufacturing. The project addresses every level of the functional safety process: "The design utilizes existing base technologies for safe and reliable communication and provides services for device discovery, configuration generation, and automatic deployment. It also provides a basis for further features such as automatic risk assessment, automatic safety verification, safety validation, and safety evaluation. Additionally, assistive features such as legal regulations checks and AI-supported configuration generation could be envisioned." While many issues of occupational safety and health lie outside the scope of functional safety, Flexible Safety Systems present a promising path forward for taking full advantage of Industrie 4.0 developments in the realm of functional safety.

### 2.2.2 Model-Based System Engineering

Research on integrating safety verification into MBSE (and, hence, into Industrie 4.0 rapidly reconfigurable systems) is still in early stages. We highlight some papers from the last several years showing development of frameworks for reaching this goal, and some offering limited demonstrations of MBSE approaches which point in the direction of this goal. We then dig deeper into recent criticism of pre-automation safety approaches, which serves to highlight the fundamental importance of automating safety validation in MBSE, even beyond Industrie 4.0 goals.

A number of researchers have produced conceptual frameworks for the project. Etz et al. propose a framework of five Service Groups for automated functional safety: our work fits into the first of these, Knowledge Representation [32]. Lee et al. highlight the importance of developing common ontologies and language for safety in systems engineering, with a focus on safety in process industry [56]. Salado presents several approaches to capturing requirements in model building–a necessary part of modeling safety requirements [69].

Alongside these theoretical development efforts, there are a number of implementation examples. Mhenni et al. demonstrate the possibility of introducing safety requirements analysis in the initial system design phase for an aircraft wheel brake subject to the

ARP 4761 standard [60]. Häring et al. demonstrate a full process of introducing safety requirements at the system engineering stage and checking their fulfillment with formal verification: in this case, their example is a malfunction indicator lamp for a car [47]. Javed et al. demonstrate the design of safety requirements for an autonomous guided vehicle using a design-by-contract approach [49]. And Bdiwi et al. develop a set of safety requirements for an industrial robot and demonstrate the real-time checking of those requirements using camera-based safety systems in an experimental production unit [22].

In all cases above, however, the safety requirements are derived from the standards "manually", by the engineers' direct interpretation. Without knowledge modeling, interpreting safety standards in context remains a critical bottleneck in the engineering process. Leveson highlights the essential problem with formulating safety requirements one-by-one: the complexity of designed systems is simply too high, and individual safety rules are unlikely to prevent emergent phenomena [57]. For example, the 1993 landing accident of a Lufthansa A320 in Warsaw was attributed in part to the design of the safety logic: the engine reversers and ground spoilers would not deploy unless both main landing gear detected enough load for the plane to be "on the ground" [87]. Due to other errors, the plane touched down with only one landing gear under load for nine seconds: in that time, commands to these braking features were blocked by the safety logic, and the plane overran the runway. Leveson presents this as an example of the limitations of component-focused safety design: while each component system performed according to its safety requirements, the emergent state was unsafe–fatally so for one crew member and one passenger. She argues for a "paradigm shift" from safety as reliability to safety as control: positively asserting the properties of globally safe states. This accords with Hollnagel's much more philosophical treatment of safety nine years earlier: he argues that safety science must focus not on the unsafe states to be avoided, but on the characteristics of positively safe states [44].

The development frameworks mentioned above highlight the importance of knowledge modeling and automated reasoning for integrating safety into changeable system configurations. The example projects show, implicitly, the limitations on Industrie 4.0 design without these tools. And the demonstrated need to conceptualize global safety properties makes clear that reasoning on knowledge models will be necessary to connect the global level to the component level in a reliable way.

### 2.2.3 Knowledge Engineering

Etz's Flexible Safety System concept uses a Safety Network Controller (Figure 2.1) informed by a knowledge-based system: this consists of all the relevant knowledge for both the production system and external requirements, together with an inference engine. The knowledge-based system is thus a key component. In his related master's thesis [66], Reitgruber develops a proof-of-concept knowledge base for this application. He selectively combines existing ontologies for industrial systems and networking to create a suitable ontology for this. Our own exploration focuses on the inference and automated reasoning

side of this system. With this in mind, we review recent developments in knowledge engineering related to functional safety.

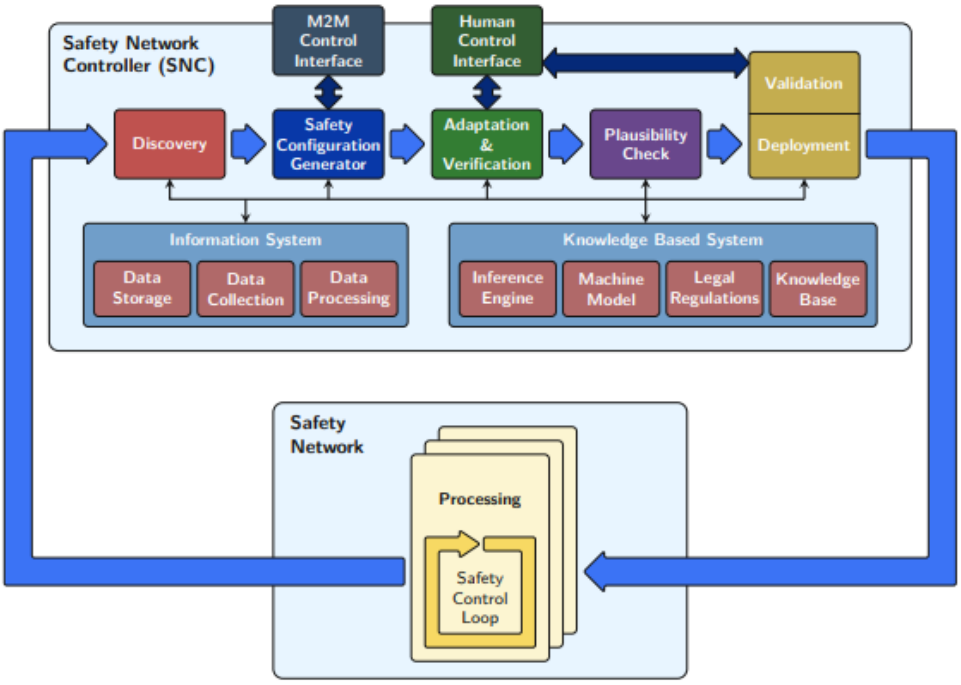


Figure 2.1: Architecture of the Safety Network Controller, from [30]

Some other authors have also worked on developing knowledge bases in the functional safety context. De Galizia uses a custom-built knowledge base for electrical systems to permit probabilistic, rather than formal-methods, safety assessment [38]. More recently, in "Mitigation Ontology for Analysis of Safety-Critical Systems", Ali et al. provide an ontology focused on identifying potential failures and mitigations [18]. Like most papers presenting ontologies, it offers this Mitigation Ontology as a potential common language to provide interoperability for safety systems. However, both the relative scarcity of papers on this topic and the lack of a widely adopted standards in practice show that the use of knowledge engineering in safety systems is still in its early phases of development.

It may be that the greatest knowledge engineering challenge in functional safety is in the requirements themselves. Etz identifies this as a key direction for future work: "it would be of great value if the system could provide a semantic model of the applicable safety regulations or standards." [30] In his thesis "Unambiguous requirements in Functional Safety and ISO 26262: dream or reality?", Sternudd explores the fundamental challenge: ISO 26262, the international standard for functional safety in vehicles, requires that safety requirements be *unambiguous*, and also that they be expressed in natural language: the inherent contradiction here is one of the oldest paradoxes in philosophy, and not likely to permit easy resolution [73]. Sternudd instead presents a proof of concept in which an

ontology is used to constrain the statements of requirements and to produce controlled natural-language versions of them. This satisfies the letter of ISO 26262 if not the spirit: the formalized version of the requirement expressed in an ontology language is the primary source of truth in this model. Realistically, the opposite direction will probably be more valuable: translating written safety requirements into machine-comprehendible representations. The extreme complexity of existing standards is both the biggest obstacle and the biggest motivation for automated comprehension. At the same time, a certain degree of flexibility in safety standards is necessary: faced with absolute, deterministic standards, it would often be the case that the only fully safe strategy is to never turn the machine on.

The interaction of two institutional factors also presents an obstacle to recasting safety standards into machine-readable ontology languages. First, in the EU, privately published safety standards gain the force of law under the MD when they are recognized by the European Commission as Harmonized Standards and published in the Official Journal of the EU [3]. This is a *legal* mechanism: the *text* of the standard has force of law. Imagining that standards were instead machine-readable raises an immediate question: how exactly would a judge interpret an RDF file with, itself, the force of law? These challenges show that modernization of existing standards faces even more complex hurdles than one might initially suspect.

### 2.2.4 Automated Reasoning

The complexity of cyber-physical systems is explosive. In turn, ensuring functional safety becomes a monumental task. In order to support reconfigurable systems, some form of automated checking would be necessary. In Etz's Flexible Safety Systems concept, the automated reasoner serves as a plausibility check on the system, while the ultimate safety verification is still manual: this human-in-the-loop system is probably necessary both due to the challenges noted above of expressing safety requirements in a machine-readable way and due to legal and institutional norms in which some particular human engineer certifies the compliance of the system.

Other current research examines alternate strategies for using formal reasoning to support functional safety. Fetzer et al. introduce the idea of using a "safety auditor" equipped with a comprehensive system model to monitor data inputs from a car in real time and correct faults as detected [34]. Bernardini et al. present formal methods as a tool to support probabilistic fault tree analysis [23]. Both of these papers engage with the fundamentally probabilistic nature of functional safety standards. However, both the plausibility check in Flexible Safety Systems and these probabilistic systems with an auditor or fault injection still require that formal reasoning be able to check at least simple cases.

Beyond the question of proving compliance, formal methods can still provide value, allowing systems to use inference on an ontology to structure their own safety requirements. Gačnik et al. have demonstrated this approach on an automobile guidance system, using

an OWL ontology [37]. Vincentini et al. used temporal logic-based models to describe the possible sequences of tasks in a human-robot collaboration scenario, and then exhaustively checked the state space in order to quickly identify safety risks [77].

So, we can see that while there are serious obstacles to safety verification using formal methods, there are still a variety of use cases in which formal methods may add real value.

# CHAPTER 3

# Scenario

In this chapter, we will choose a realistic scenario for our system model and automated verification process. The scenario will allow us to construct a specific system model and to then apply our automated verification approach. In order to choose a suitable scenario, we will look at existing research, consider the factors that went into others' choice of scenario, develop our own criteria, and then settle on a suitable example. When we define our example, we will need to specify multiple aspects. The system itself comprises physical configurations, configurations, and functional safety requirements. These plurals are quite important: in order to evaluate our product in the Industrie 4.0 context, our system needs to be reconfigurable: we need to have multiple, related configurations of both physical components and control flows. Our automated verification process will check the conformity of the physical and control configurations with the safety requirements in each of the variants. Thus, these three elements, in an identified number of variations, comprise our scenario.

## 3.1 Criteria for the Choice of Scenario

In order to develop and evaluate our system model and our verification approach, we will need to choose an appropriate test scenario. In the existing literature on functional safety, researchers have investigated a wide variety of examples. We will look at these examples and see what factors they considered in their choices of use case.

We see functional safety research across a wide variety of domains. Häring et al. apply a Model-Based System Engineering (MBSE) approach to an automobile's malfunction indicator lamp [47]. Mhenni et al. look at aircraft braking systems [60]. Baumgart et al. use automobile steering in their discussion of a model-based design approach for functional safety [21]. Bdiwi et al. examine automatically configuring safe zones around a working robot and enforcing them using cameras and image recognition [22]. Catelani et al. consider safety configurations in process industry [26]. They provide a detailed

methodology for assessing electromechanical compliance based on Reliability Block Diagram methodology. Functional safety and verification are broad topics which could be examined in a large number of contexts–although automotive design and industrial automation seem to be the most well developed. We will focus on industrial automation.

Our scenario should relate to real-world standards. In the automotive realm, Häring et al. connect their work on modeling the safety systems of a malfunction indicator lamp to the ISO 26262 standard. Adhikari et al also work in relation to ISO 26262, in their case analyzing an automobile emergency braking system [16]. In both of these cases, the use case is tied to specific safety standards. For Mhenni et al., that is ARP 4761; for Baumgart et al., it is again ISO 26262. That said, they are not dealing with the full complexity of those standards. For our scenario to be reasonably relevant, we would like to have a clear relation to real-world standards, but for it to be practical, we would like to be able to use a simplified version of those standards.

Our scenario should present an obvious safety problem which our safety system can solve. And, just as the scenario should have an obvious relation to safety, the safety measure itself should be clear in its function. In Bdiwi et al.'s example with the industrial robot, the safety requirement is fairly straightforward: the robot needs to stop if an object enters its exclusion zone. Similarly, in Häring et al., the warning light turns on. In both of these examples, we see that, while the operation of the system itself is very complex, the safety intervention is simple: "turn off the robot", or, "turn on the light", although in the robot example, the safe zone itself changes as the robot and human move in the work area. In our earlier examples, actually achieving a safe state is more complicated: enforcing safe automotive braking is much more than "apply the brakes"; angle of attack correction is far more nuanced than "pitch down the elevators"! And indeed, in Adhikari et al.'s braking example, they chose a very, very simplified subset of safe behavior for their study. A contrasting approach is Catelani et al.'s process industry example, in which the scenario and the safety mechanism are not actually described, presumably as they are too complicated. Instead, the Reliability Block Diagram examples are simply presented as "the safety functions that are present in a complex system designed for process industry application". We would really like to not fall into this situation, and as we are inventing our own scenario out of whole cloth, there is no reason that we should. What we can take from both the simple safety behaviors in the industrial examples and the conscious choice to simplify in the automotive examples is: our safe behavior should be very simple. Ideally, it should be boolean: on-or-off, yes-or-no.

At the same time, our scenario should present a significant level of complexity and interrelation. Real-world safety systems are vulnerable to unintended consequences: when safety systems don't perform their functions, unsafe situations won't be prevented; but, the operation of a safety system to correct one unsafe condition may violate another safety requirement. A dramatic example of unintended consequences in aerospace control systems illustrates this latter failure mode vividly: the angle-of-attack sensor on the Boeing 737 MAX. The US Federal Aviation Administration report on the 737 MAX control system failures describes a series of faults in which the flight control software

would continually adjust the elevator trim in response to a single sensor, even overriding pilots' control inputs [33]. The possibility of unexpected behavior causing safety violations drives home the value of formal verification: an automated logical process can detect problems arising as unintended consequences which human design intuition might miss. Implicitly, the report also underlines the potential value of formal verification, as the faulty system had already passed traditional functional safety review processes! From this, we can see that it would be valuable for our safety rules to contain a non-obvious requirement which an unsafe configuration might violate. Constructing a scenario with a "toy" unintended consequence may not be entirely feasible, but would definitely be desirable.

All of the examples we have talked about have been very concrete: one could easily picture, or draw a picture of, the example. We should choose an example with a similar level of physical simplicity.

Finally, we see one major contrast between these examples and our own requirements. The core idea of Industrie 4.0 is that of rapid reconfiguration: our aim is to produce a verification tool to support this. The existing scenario examples are all essentially unitary. In contrast, our example scenario needs to be essentially reconfigurable: both the physical configuration and the control configuration need to support multiple changes. On the other hand, the safety rules will remain the same throughout.

So, our criteria for a good scenario are:

1. The system should offer the potential to use a simplified version of real-world standards.
2. The system should have a clear relation to safety.
3. The system should be complex enough to demonstrate robustness against "unintended" consequences.
4. The safety controls and the rules chosen should be amenable to "on-or-off" interventions for simplicity.
5. The physical system should be simple enough to readily visualize.
6. The physical components and control flow should be reconfigurable.

## 3.2 Emergency Stop Buttons

With these factors in mind, we will examine a system based on emergency stop controls. Emergency stop functionality contains an appealing balance of simplicity and complexity: the controls themselves and their intended safe behavior are simple; however, the interaction of different e-stop controls is potentially complex.

E-stop functionality is governed by a number of different standards [75]. ISO 13850 specifically describes the requirements of e-stop buttons themselves. Other standards such as ISO 13849, IEC 60204-1, and IEC 62745 present broader safety standards which make critical use of e-stop functionality. The complexity arises from a few subtle points in the safety standards. In particular:

- An e-stop must bring ALL equipment within the operator's view to a safe stopped condition, not just its attached equipment. This set of equipment is referred to as the "span of control of emergency stop device(s)" in ISO 13850 [12]. The standard actually requires the span to consider not just visibility but the possibility of sound or smell alerting an operator to an unsafe situation. However, even just the visibility requirement introduces a gigantic level of complexity, especially if wireless e-stop devices are permitted.

- An e-stop is not the same, semantically, as a normal stop button: while triggering an e-stop brings the machine(s) into a safe state, resetting the e-stop button it must NOT start them again. Instead, the reset button must be on the machine itself (per ISO 13850 4.1.1.2). This "one-way-door" behavior creates room for complexity in a seemingly simple use case.

- Both connected and disconnected e-stop buttons are significant: any e-stop present has to actually be connected. This is a major issue for moveable control devices with built-in e-stop buttons. One such example is the "Safety Hot Swap" system produced by Sigmatek: these allow modules with e-stop functionality to be disconnected during operation by first pressing a "disconnect" button [71]. They use powered lights to color the e-stops red and yellow: without power, the buttons are instead a neutral gray. This color system is in line with that described in ISO 13850:2015, 4.3.8: "at least one of the following measures shall be applied to avoid confusion between active and inactive emergency stop devices: device colour changing through illumination of the active emergency stop device" [12]. The manufacturer further asserts that this system complies with IEC 62061 and ISO 13849-1 and -2.

## 3.3 Scenario Description

We will develop a scenario which allows us to consider these complexities while keeping the overall complication low. Below, we describe our scenario setup. Then, we evaluate it against the requirements set in the previous section.

### 3.3.1 Physical Components

We will consider a robot assembly area spanning three rooms. The northeast and northwest spaces are a single control room, with a permanently installed control panel including an integral e-stop. The southeast and southwest rooms are production rooms.

From the control room, a door and a glass window open into each of the production rooms.

Each production room contains:

- An operator work area
- One or more robot assembly stations, each comprising:
  - A robot assembly machine
  - An integrated e-stop
  - A control station with a port for a removable control panel.

The two production rooms are separated by a removable opaque barrier, which may be present or absent in different variations on the scenario. By default, it is present. We show the entire setup in Figure 3.1.

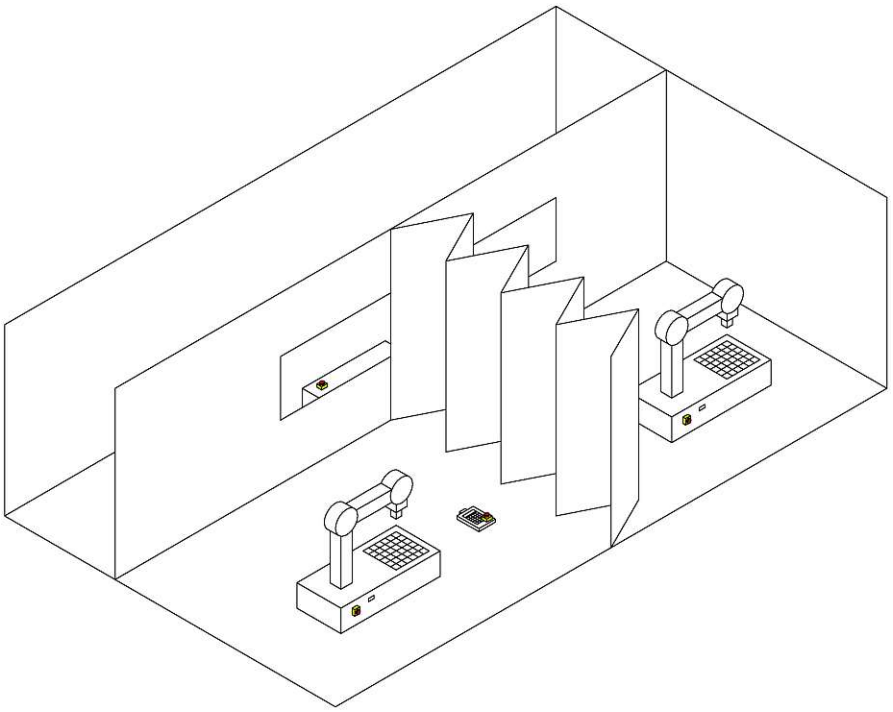

Figure 3.1: The physical components of our scenario

Finally, the scenario includes a number of removable control panels: each control panel can be used with any robot station, and contains an integral e-stop. Any number of operators may access the rooms, using any number of removable control panels. Window and wall sections introduce the challenge of visibility: recall that each e-stop button must

span all visible machines. In our default configuration, one panel is present, and is not connected to anything.

We also choose *not* to model certain equipment. We will assume other safety equipment like light barriers is present, but as it is independent of the e-stop functionality we are studying, we will not model it. As a matter of model simplification, we do not consider doorways for operators. We also do not consider whether windows can be opened and/or traversed.

The main physical reconfiguration we will examine is the movement and connection and disconnection of removable control panels with integrated e-stops. The addition or removal of the dividing wall between the production rooms, changing visibility, also offers a range of configurations. We can also consider the addition or removal of robot assembly stations.

### 3.3.2 Rules Definition

We will use a simplified set of rules taken from existing standards for emergency stop buttons. In particular, we are basing our rules on ISO 13850:2015 "Safety of machinery - Emergency stop function - Principles for design". Our first two rules, on the presence of e-stop buttons, are simplified from section 4.3.2. The actual standard requires e-stops on control panels, entry-exit areas, operator intervention locations, and human-machine interaction zones. In addition, section 4.6.1 requires e-stop buttons on portable control panels. Our rule for visibility is based on section 4.1.2. The actual standard is broader than merely visibility, and allows for exceptions based on risk assessment; however, we greatly simplify for our own purposes.

Our basic rules:

- An e-stop button stops all connected machines
- An e-stop button is red and yellow
- One-way door: disengaging an e-stop does not turn on machines.
- Each machine has an integrated e-stop
- Each control panel has an integrated e-stop
- Each emergency stop button must span all machines **visible** from its location.
- Each machine and each e-stop is connected to the safety network
- Machines and e-stops registered on the safety network must not disconnect.

When we build our system model, we will elaborate on these rules and develop a structured set of requirements with test cases.

### 3.3.3 Scenario Variations

We will also introduce three variant test cases. Each of these represents a plausible reconfiguration which should violate at least one of the requirements above.

1. A machine disconnects from the safety network.
2. The opaque wall is removed from the facility.
3. A new machine is added to the network.

Each of these changes would demand significant reconfiguration of the safety measures in the facility: introducing the change without suitable safety reconfigurations should produce an unsafe state.

## 3.4 Evaluation of the Choice of Scenario

Our use case satisfies the criteria we developed above:

1. *The system should offer the potential to use a simplified version of real-world standards:*
   In our model, we use a simplified set of rules derived from ISO 13850:2015 "Safety of machinery - Emergency stop function - Principles for design".
2. *The system should have a clear relation to safety:*
   E-stops are obviously safety-related.
3. *The system should be complex enough to demonstrate robustness against "unintended" consequences:*
   The requirement that each e-stop span each *visible* machine creates significant complexity.
4. *The safety controls and the rules chosen should be amenable to "on-or-off" interventions for simplicity:*
   We focus on boolean variables: is the e-stop on or off? Is it connected or not? Is a given machine visible from a given e-stop?
5. *The physical system should be simple enough to readily visualize:*
   We provide a visualization in Figure 3.1.
6. *The physical system and control flow should be reconfigurable:*
   Physical reconfiguration can happen by moving removable control panels, by adding/removing robots, and by adding/removing the wall divider. Control flow reconfiguration can happen by requiring operators to be at certain stations or not.

## 3.5 Summary

We now have a good example scenario on which we can test our design artifact. We will put it aside for the next two chapters, as we begin designing our verification process. We will return to this scenario in Chapter 7, when we apply our process to it.

CHAPTER 4

# System Modeling

Now that we have a narrative description of our example scenario, we begin moving towards the technical part of our project. We begin by choosing appropriate tools for system modeling and automated reasoning. In Chapter 7, we will produce suitable system models for our scenario according to these plans.

Our technical work divides into two major parts: representing the system, and performing reasoning on that representation. In this chapter, we focus on the representation. We will introduce the important ideas related to system modeling and the well-known approaches and technologies. We will evaluate them to determine the extent to which each approach will help us explore our research question.

## 4.1 What is a System Model?

Our overarching aim is to be able to verify that a system model satisfies a set of safety requirements. But even this simple sentence reveals a semantic wrinkle: the requirements are, after all, part of the system, and hence part of the system model. So, it would be more accurate to say that the system model, which includes its requirements, is *consistent* or *correct*. As a check against this kind of semantic ambiguity, we will carefully define the terms we intend to use.

When we write *system*, we mean the physical components, the control flows by which they interact, the use cases by which our imagined operators interact with these, and the requirements which must be satisfied. By "system model", we mean a simplified representation of the underlying system which allows for easier understanding or reasoning.

In our scenario, the system is already a simplified "toy", so there is little practical difference between the system and its model; however, the distinction is conceptually important, and so we preserve it. Imagine, if you will, that the elements of our scenario

actually exist in a factory somewhere on the outskirts of Vienna, and our system model is confined to these pages and the associated code repository.

There is then an underlying assumption, without which nothing we are doing matters: *If the model is accurate, then a verification of consistency with safety requirements in the model implies consistency with safety requirements in the actual system.* In this thesis, we accept this assumption to a limited extent: we consider our semi-automated process to be an assistant system, catching obvious mistakes and allowing learned knowledge to be saved by updating the model. But we note: for a real-world implementation, "if the model is accurate" implies a monstrous amount of work, and the validity of the assumption always demands close scrutiny.

## 4.2 Criteria for the Choice of System Modeling Approach

What do we want out of our system model?

1. The model should be able to **represent all types of entity** in our scenario:
   a) Physical components
   b) Control flows
   c) Use cases
   d) Requirements

   This is necessary because the requirements are our primary subject matter, and they could conceivably make reference to all other types of entity.
2. The model should make our system design **precise and legible**: this is a pragmatic necessity for us as authors.
3. The model should support **extension and modification** of our scenario: this is necessary in order to support the "rapid reconfiguration" goal of Industrie 4.0.
4. The model should support **automated reasoning**: this is necessary since we need to implement automated verification. Inference-based reasoning may not be strictly necessary.

## 4.3 Approaches to System Modeling

How, then, should we model our system? We need to represent a complex system, which includes radically different kinds of entities. At a minimum, we have:

- Physical components: *things* in our factory which can *do actions* or *have actions done to them*

- Control flows: *relationships between the actions* which objects do, separated in time.
- Use cases: tasks which users want to complete, and the sequences of subtasks and actions which may comprise a larger task.
- Requirements: *statements* about various entities in the system which demand to be true.

(We purposefully leave open one question: can requirements refer to other requirements? On the one hand, this would allow categorization and greater understandability. On the other hand, this could *dramatically* increase the logical complexity of our model, and may kick it beyond the capabilities of realistic automated reasoning solutions. We return to this question in the next chapter.)

We consider two approaches. One is to view the model as a representation of entities and relationships: any kind of boxes-and-arrows diagram would be an example of this. An alternate approach is to view the model as a *knowledge base*: a collection of statements. The first approach leads directly to Model-Based System Engineering (MBSE); the second to ontology engineering. The distinction at the high level is very subtle. However, as we will see in the next two sections, the very delicate distinction in what each model *is* leads to significant differences in *how each model can be used*, and it is this latter question which really interests us.

### 4.3.1 Model-Based System Engineering

We first introduce MBSE. We start with a high-level overview of the concept, and then look at languages which can be used in this methodology.

#### What is MBSE?

MBSE is a methodology for managing requirements and design in the development of complex systems [70]. The term stands in contrast to document-based engineering: where traditionally a large variety of documents have been used to record requirements, design decisions, and everything that results over the course of an engineering project, in MBSE a single model (in a computer program) collects all of these elements and defines their interrelationships. As a methodology, MBSE proposes to offer several improvements over document-based processes:

- Requirements traceability: Defined relationships between all elements allow each element of the system to be traced back to the requirements it is involved in satisfying. Moreover, if a requirement is changed, all other related entities can be decisively identified.
- Single source of truth: Where different documents may conflict, in an MBSE project, the model is the ultimate source of truth for design facts and decisions.

- Transparency: By collecting all design-related information in one model in one common format, different stakeholders on the project can see all information reflected in a common language.

MBSE has also attracted criticism, most of which boils down to disputing the extent to which these improvements materialize in practice, and noting the expense and friction introduced by the task of building and maintaining the model [67]. These critics admit, however, that these benefits are largely realized for clients and funding agencies, who see their requirements tied to the whole of the design in a complete and unambiguous way; the criticism focuses on the loss of time for engineers who may continue to maintain document-based practices as a pragmatic necessity in parallel to performing MBSE practices as mandated from above.

For our purposes, as we are in effect the contracting party and the engineer at the same time, we are able to realize the main benefits: **an MBSE-style system model will allow us to set our safety requirements in unambiguous terms and tie them to a complete description of all entities in our system.**

### MBSE Languages

One of the benefits listed above is *transparency*: MBSE provides a model written in a "language" which is (intended to be) common to all stakeholders. So, what language? In order to use a system model to represent our requirements and their relation to the other entities in our system, we will need to choose a language.

A variety of MBSE modeling languages exist. The International Council on System Engineering (INCOSE), identifies six examples on their wiki page, last updated in 2021: Arcadia, EAST-ADL, IDEF, OPM, the Systems Modeling Language (SysML), and the Unified Modeling Language (UML) [48]. In their overview paper, De Saqui-Sannes et al. identify around a dozen prominent examples as of 2022, including SysML, UML, OPM, and Arcadia, among others [28]. They distinguish the languages by their capabilities in different aspects of MBSE, and by their support by available tools. It is evident that they view SysML as the most prominent example, as they continually present it first and present other languages as competitors or alternatives. SysML owes much of its prominence to UML, the international standard modeling language for software architecture: SysML is a direct expansion of UML, especially developing its capabilities for requirements analysis. They also show that SysML enjoys dramatically better support from available software tools than its competitors. Even in the niche field of biology, SysML is the dominant modeling language: Fudge and Reeves show that since 2016, it has been the most popular choice [36].

Should we choose SysML? Unsurprisingly, choosing an MBSE language is a complex task. Basnet et al. illustrate this clearly when they create their decision-making framework for MBSE language [20]. They use questionnaires with Likert scales, pilot model designs, and multi-criterion Bayesian inference to determine the MBSE language best suited to

a client's project. In their case study, considering a range of criteria and their client's stakeholders' questionnaire results, they find that SysML is the best fit.

We will forgo this complexity, and choose SysML for three main reasons:

- It is generally the most prominent and representative modeling language for MBSE
- Free and open source tools allow us to build and manipulate a model for free
- It is especially focused on requirements, which are the main subject matter of our modeling.

Thus, a SysML model will allow us to clarify our requirements and tie them to all other elements of the system.

### Strengths and Weaknesses of SysML

The main strengths of SysML are its explicit structure and its focus on requirements traceability. The explicit structure allows for a clear separation of concerns within the design process, with requirements, user behavior, and components held conceptually separate. Traceability means that any element of the design can be traced back to the requirement or requirements which it satisfies. Conversely, any change to the requirements can be followed down through the model to identify all affected components.

The weaknesses of SysML are its rigidity and limited scope, its weak amenability to automated reasoning, and the relatively poor quality of available software. The rigid model means that the details of the design can be written down and perhaps linked to parameters or interfaces, but that the full set of domain-specific interrelationships cannot be represented. SysML does not produce a full digital twin for the system so much as a digital twin (or full replacement) for *the documentation of the system*. SysML also has very limited automated reasoning options. For SysML, while there is some scope to perform verification, manual steps are still necessary where the model is unable to represent domain-specific information [42]. SysML 2 expands the reasoning capabilities: for example, the AI tool Imandra Automated Reasoning is able to perform verification tasks on a SysML 2 State Machine diagram, relative to its parameters [72]. However, a *general* formal verification solution is not yet present.

### Applicability of SysML to Our Project

For our project, we can use a SysML model to produce an exact specification of our scenario. When we change the configurations, we will refer to our SysML model to see what elements are affected. However, we will be using this essentially on a "soft" level: it will provide an overview and a sanity check, but will not be used as a detailed automated reasoning tool. It may tell us that a certain robot station is affected by a requirement change, but it may not tell us how, and it cannot be relied upon for an exhaustive accounting of all consistency conditions.

For greater opportunities for automated reasoning, we turn to our second system modeling approach, ontology engineering.

### 4.3.2 Ontology Engineering

Our first approach was to model our system as a set of entities and relations: for this we have settled on SysML as a modeling language. Our second approach is to conceive of our model as *knowledge about the system*. For this, we turn to the discipline of ontology engineering.

#### What is Ontology Engineering?

The term ontology originates in philosophy, and can be very (very!) roughly summarized as the study of the existence of things and how things that exist might be related to each other [43]. In philosophy, these questions might be examined in any number of contexts: from theology to causality, from mathematics to human nature, and so on. However, the technical domains have adopted the term to describe something much more particular: a computer science object which specifies, in a precise way, and in a certain context, *what things are* [68]. Note the double meaning in English: "what things are" as existence: "to be or not to be?", and also "what things are" as type: "what (kind of thing) is it?" Ontology concerns both questions. Depending on context, this might be a thesaurus, an XML schema in a database, or a model for linked data.

We are interested in producing a model of knowledge about our industrial system. This points us in the direction of a knowledge base: a structured collection of facts. In the modern conception of a knowledge base, the nature of both the facts and the structure is more precisely characterized: the facts are "sentences": subject-predicate-object triples, such as `clifford hasColor red` and `clifford isInstanceOf Dog`. The structure follows an object-oriented paradigm, where class relationships are also defined by s-p-o triples: `Dog isSubclassOf Animal`. A knowledge base may sometimes be called a *triplestore*, as its data is presented entirely as a collection of s-p-o triples.

In our Clifford example [24], our immediate inclination may be to draw the conclusion `clifford isInstanceOf Animal`. But, it is very important to note: a knowledge base is simply a set of sentences. The logic by which we might infer this last sentence to be true is NOT necessarily part of that knowledge base, and there is NO guarantee that a knowledge base containing the first example sentences also contains this one. We will see below in *Ontology languages* that different technologies provide for different degrees of reasoning.

#### Concepts in Ontology Engineering

Knowledge bases bring an almost daunting level of flexibility. In modeling languages like SysML, the distinct categories and relationships are baked in as diagram types and

specific named relationships; in contrast, the only strict structure in a knowledge base is the triple itself. In order to understand any particular ontology language, it is useful to develop some key concepts first.

One is the idea, taken originally from formal reasoning, of the "T-box" and the "A-box". The T-box contains the "terminology" that defines a given reasoning context, while the A-box contains the "assertions": the specific statements in the context. The exact level at which one draws the distinction depends on one's needs. For example, we might consider the hierarchical relationship `CanisFamiliaris isSubclassOf Carnivora`to be part of our T-box, if we are principally interested in individual animals and the more specific properties that describe them. On the other hand, if we are interested in species taxonomy, then the same statement might properly belong to our A-box, and more structural relations like `Species isSubTaxonRankOf Order` might belong to our T-box. Reasoning rules also belong to the T-box, about which more below. In SysML, the rigidly defined types and relationships make the T-box/A-box relation explicit; in a knowledge base, as everything is stored as a s-p-o triple, it is up to us to use this concept to recognize the two parts of the ontology which have qualitatively different roles.

Somewhat related to the T-box/A-box distinction is the idea of classes and instances, taken from object-oriented programming. If classes are "kinds of things", then instances are the things themselves. In ontologies, this distinction is fairly soft and not necessarily enforced programmatically—we will see this in more detail in the following section as we trace the development of RDF/RDFS/OWL. In the very broadest terms, we can see classes as naturally related to the T-box and instances as naturally related to the A-box. However, there is a third type of entity which complicates this.

In addition to classes and instances, ontologies also include *properties* – predicates which relate subject and object. So, for example, `isSubclassOf` and `isInstanceOf` are properties. However, since *everything* is defined within the ontology itself, these properties may also be subjects. Consider the simplified fragment of an ontology in Listing 4.1.

```
hasColor isInstanceOf Property
hasColor hasDomain Object
hasColor hasRange Color
Color isSubclassOf Class
isInstanceOf isInstanceOf Property
isInstanceOf hasDomain Object
isInstanceOf hasRange Class
hasRange isInstanceOf Property
hasRange hasDomain Property
hasRange hasRange Class
hasDomain isInstanceOf Property
hasDomain hasDomain Property
hasDomain hasRange Object
```

Listing 4.1: Simplified ontology

In order to indicate that `hasColor` is a property, and to define its domain and range, we are forced to treat it as a subject. The already-spiraling complexity of this very simple

example illustrates two key points. First: both the power and the difficulty of ontologies results from the gigantic complexity of saying even something as simple as `clifford hasColor red`.

But the second key point is: we haven't even managed to say what we *mean* by `hasColor`! That is, we want `clifford hasColor red` to be valid because `red isInstanceOf Color`. But we should not accept `clifford hasColor apple` because `apple isInstanceOfColor` isn't true! And we *really* shouldn't accept `clifford hasColor Dog` because "Dog" is a class, not an instance of anything, let alone of `Color`! In fact: while we have said that `hasColor hasRange Color`, we haven't actually said that `x prop y` and `prop hasRange Range` **implies** `y isInstanceOf Range`, which is what we really want in all of this. And, if we have that, we want it to trickle down through subclasses and instances so that `x hasColor y` and `hasColor hasRange Color` implies `y isInstanceOf Color`.

In short, we want **reasoning**. But reasoning is actually a lot to ask–we're devoting the entire next *chapter* to it and really barely scratching the surface. Still, at this point we can see that we want three things:

- To **verify** that certain conditions are true throughout our ontology. For example, we might wish to require that no object can be both a class and an instance of a class.
- To **query** for elements of our ontology which meet certain conditions.
- To **infer** new sentences. For example, we would almost certainly want a transitive property to hold if "A isSubclassOf B" and "B isSubclassOf C" are in our knowledge base, then we would want to automatically add "A isSubclassOf C". Or, we might wish to do this only if "isSubclassOf C hasPropertyType transitive".

And we may immediately note that the processes are quite different: starting with the rather problematic sentence `clifford hasColor Dog`, a verification approach might identify the error we described above, while a query approach might (less helpfully) tell us there is no match. But, an inferential approach might instead add `Dog isInstanceOf Color` to our knowledge base!

And anyway, look again at the bold-faced *implies* two paragraphs above: is that really what we mean? Do we mean *if*, *only if*, or *if and only if*? To what extent does it depend on the reasoning process that we plan to run?

All of this is to say: reasoning is very difficult, but reasoning is necessary if we want our knowledge base to be anything more than a monstrous list of sentences which may or may not make any sense. In the next section, we will look at different ontology languages, where we will see that the level of reasoning ability is often the key differentiating factor.

### Ontology Languages

The development of ontology languages begins in the context of the Semantic Web, a project supported by the World Wide Web Consortium (W3C) aiming to make the *meaning* of information machine-readable, in contrast to the HTML-based web, in which meaning is purely human-readable [78]. A series of early ontology languages were based on XML, such as Ontology Exchange Language, Ontology Markup Language, Simple HTML, Ontology Extensions, Ontology Inference Layer, and Darpa Agent Markup Language + OIL [13]. At present, these have been essentially superseded by the Web Ontology Language (OWL). In order to understand OWL, it is necessary to first understand the Resource Description Framework (RDF) and RDF Schema (RDFS), the ontology languages upon which it is based. OWL arises from a step-by-step expansion of the capabilities of RDF.

RDF is a data exchange model originally developed by W3C as a serialization format for metadata [85]. It provides the essential subject-predicate-object structure on which an ontology may be built. Entities in an RDF knowledge base are specified with a namespace prefix, either user-defined or built-in. So, in our examples going forward, we will be writing `ex:clifford ex:hasColor ex:red` and `ex:clifford rdfs:member ex:Dog`, where `ex:` abbreviates our example namespace, and `rdfs:` is one of several special built-in namespaces, along with `rdf:`, `owl:`, and `xsd:` (for XML Schema data types for literals). An additional special namespace is `_:`, which is used for ad-hoc graph nodes generated by automated processes within the knowledge base. Our reification example below uses this.

RDF has an extremely limited vocabulary of special terms [85]. `rdf:type` supports a very limited ability to place individuals in classes, including a very short list of built-in classes like `rdf:Statement` and `rdf:Property`. However, there is no support for class-class relationships.

`rdf:subject`, `rdf:predicate`, and `rdf:object` allow for reification of statements: a statement like `ex:clifford ex:hasColor ex:red` can be represented as in Listing 4.2.

```
_:statement42 rdf:type rdf:Statement
_:statement42 rdf:subject ex:clifford
_:statement42 rdf:predicate ex:hasColor
_:statement42 rdf:object ex:red
```

Listing 4.2: Reification

In this way, the statement itself is an object to which other statements can refer: “Emily says that Clifford is red” can be encoded `ex:Emily ex:says _:statement42`. In this way, the expressiveness of RDF is actually far greater than the s-p-o structure initially suggests. The special `_:` unnamed namespace allows for newly-generated statements to be well-defined even though they aren’t defined in any particular external namespace.

We can see that while RDF is expressive, it has virtually no built-in structure, and is under no obligation to "make sense". RDFS begins to address these two limitations [84]. It extends RDF with three related elements:

- Classes: Subclass relationships and inheritance are now supported
- Domain and range: Predicates can now be defined to have domain and range in certain classes.
- Inference: Class inheritance can be automatically inferred from subclass relations. In addition, class can be inferred from the domain and range of a predicate.

For example, if `ex:clifford ex:isFrom ex:Dog` and `ex:isFrom rdfs:Range ex:Country`, RDFS would infer the new statement `ex:Dog rdfs:subclassOf ex:Country`. Note that this is aggressively open-world reasoning: there is no syntax for "not", and no reason that this new statement might raise any automated eyebrows.

Still, the reasoning permitted by RDFS is extremely limited. OWL dramatically expands the reasoning capabilities. Now, we can ensure, in Listing 4.3, that Clifford is not a cat.

```
DisjointClasses( ex:Cat ex:Dog )
ClassAssertion( ex:Dog ex:clifford )
```

Listing 4.3: Ensuring that Clifford is not a cat

If we were to additionally assert `ClassAssertion( ex:Cat ex:clifford )`, our knowledge base would be inconsistent, and OWL's automated reasoning support could detect this conflict. This reasoning example follows a verification approach. Alternately, we might entail a new fact from this: `ClassAssertion( ObjectComplement( ex:Cat ) ex:clifford )`: Clifford is not a cat.

One immediately notices that the syntax above no longer matches RDF—if anything, statements like `ObjectPropertyAssertion( ex:hasColor ex:clifford ex:red)` look like LISP! However, W3C defines a complete mapping between OWL and RDF, allowing an OWL knowledge base to be fully represented in an RDF graph database [81].

The rich reasoning capability of OWL makes it a very attractive tool for both representing and checking a system model. However, there are significant limitations to automated reasoning over knowledge bases. In particular, the problem in its fullest generality is undecidable. As a result, multiple versions of OWL exist, such as OWL Lite, OWL DL, and OWL 2. These differ in the richness of their available axioms and assertions, and their amenability to automated reasoning techniques. For example, "DisjointClasses" in the example above is new to OWL 2 [82]. We will examine the limitations of reasoning options for different flavors of OWL in the next chapter.

Using an OWL ontology to model our system will give us direct access to automated reasoning. At the same time, its flexibility will ensure that we can model all aspects of our system.

#### Extensibility

One of the main advantages of an RDF-based knowledge base is its extensibility. Using the namespace prefixes, it is possible to use two or more knowledge bases together – in fact, it is necessary to, since the built-ins of RDF, RDFS, XML Schema Datatypes, and OWL are all separate namespaces. But in practice, the real advantage of this is that it is not necessary to reinvent the wheel: one can use an established domain-specific ontology for some aspect of a model, while adding necessary concepts in a new project-specific namespace. For example, it is common to use the Friend of a Friend (FOAF) ontology for any application in which personal profiles or social network information are relevant [35]. In theory, a well-developed public ontology for a specific domain would enable a high level of interoperability, for example, between industrial components from different manufacturers. However, this is more a current topic of research than a mature technical reality.

OWL also supports extensibility with such properties as `owl:sameAs`, which allows one to bridge across ontologies, identifying common elements between them [79].

#### Strengths and Weaknesses of OWL

OWL offers three major strengths: expressiveness, reasoning capability, and extensibility. Its semantic flexibility allows it to express any kind of relationship between any kind of entities, including second-order relationships between relationships. At the same time, its class structure allows one to encode semantic relationships between different types of entity. OWL also has well-developed support for reasoning. Finally, its extensibility is attractive for any project: one can pull in any suitable ontology.

On the other hand, the flexibility of OWL is something of a drawback: the complexity needed to capture even fairly simple relationships makes a system model using OWL absolutely huge and, as a result, somewhat opaque.

#### Applicability of OWL to Our Project

We need to use automated reasoning to check that our system model is consistent with its safety requirements. OWL will support this. At the same time, OWL is flexible enough to describe any system at all, so long as we are willing to go to the trouble of creating a suitable ontology.

### 4.3.3 Evaluation of the Choice of System Modeling Approach

To what extent do our two identified modeling languages meet the criteria from the beginning of this chapter? We evaluate our four criteria on a qualitative scale of “very bad”, “bad”, acceptable”, “good”, and “very good”.

**Suitability of SysML**

1. Very good fit: each type of entity corresponds to explicit diagram types in SysML.

2. Good fit: In practice, SysML finds its main use in precisely this role: making complex systems precisely specified and legible for contracting parties and other top-level stakeholders. At the same time, the difficulty of using SysML software places practical limits on how legible this can actually be.

3. Acceptable fit: SysML supports strong traceability, allowing us to see how changes in one domain affect components in another domain. Extensibility is more limited: anything new needs to be introduced manually.

4. Bad fit: While SysML 2 supports some types of automated reasoning, we cannot implement automated reasoning with any kind of generality or flexibility.

Overall, we can see that SysML provides strengths in describing and specifying our system. It can also help us see the effects of our configuration changes, though this does not extend to actually checking their effects rigorously.

**Suitability of OWL**

1. Good fit: OWL is completely open-ended and can represent whatever we want. We will, however, have to provide the structural architecture ourselves.

2. Acceptable fit: OWL can make things as precise as we can write them. However, its graph databases have limited legibility, as they are crowded with many many statements in order to fully specify even fairly simple relationships.

3. Good fit: OWL supports extensibility very, very well with its namespace function and the existence of already-established domain-specific ontologies. It supports modification as well as anything else, although again the architecture underlying modifications is up to us.

4. Very good fit: OWL provides strong automated reasoning support. The limitations are really in the mathematics of formal reasoning itself, rather than in any shortcoming of OWL.

Overall, we can see that OWL is very well-suited to our automated reasoning application. However, it is a bit clumsy for the simpler goal of specifying and understanding our system.

## 4.4 Choice of System Modeling Technology

While a number of software packages for SysML exist, our aim is principally to understand our own requirements and our necessary system components. For this, it will suffice to use the SysML modeling *language* to create our diagrams and document our design process. To produce readable diagrams using standard symbols, we will use the online drawing platform draw.io. draw.io supports SysML diagrams natively. Free users can download their saved files in an XML format, allowing for practical workflows.

To build and manipulate our knowledge base, we will use the Protégé software, a free tool for ontology management developed by the Stanford Center for Biomedical Informatics Research. In a comparison of ontology tools, Alatrish finds that Protégé offers the highest level of usability, interoperability, and extensibility [17]. For our purposes, its relatively clean UI, its support for multiple reasoning tools, and its built-in visualization tools will all be very helpful. While Alatrish finds that TopBraid Composer offers a similar range of features, we are happy to choose Protégé as it is open source software and enjoys wide use.

With OWL, we have a further decision to make: should we use pre-existing ontologies in our process? In theory, the extensibility of OWL knowledge bases is one of their main strengths. One can bring in virtually any already-existing ontology in a new namespace, and even use `owl:equivalentClass` and `owl:equivalentProperty` to knit them together. Reitgruber uses this approach in his "Knowledge Base for Reconfigurable Safety Systems" [66]. He evaluates nine potentially relevant ontologies before choosing five to partially incorporate into a bespoke Reconfigurable Safety System ontology. However, problems arise in this kind of approach: the use of so many namespaces makes queries and reasoning difficult. For example, something as simple as searching for all parts of a certain machine quickly breaks down if several different ontologies all use a `hasPart` property in their namespaces, and the system model draws unpredictably from these different namespaces. A second problem with bringing in outside ontologies, as Reitgruber also acknowledges, is bloat: each existing ontology is gigantic, and makes it hard to see the relevant elements in the proof of concept. For these reasons, we are choosing to make our own ontology for the proof of concept. In real-world applications, it may be advantageous to use an existing ontology if, for example, the ontology is already widely used inside a firm or in a certain industry. FOAF, the ontology which Facebook developed for personal contact and relationship information, may be such an example: it is widely used and allows for meaningful data interchange between organizations and knowledge bases [35]. However, its maturity and wide adoption are more the exception than the rule. As a result, our process will be based on creating our own ontology.

## 4.5 From SysML to OWL

The main purpose of our SysML model is to understand our own requirements. Once we have a solid set of requirements, we will follow their implications to plan the system

elements which satisfy them. In this way, we will develop a base system model whose own requirements are well-posed and make sense. However, one major weakness of SysML is that, unlike UML, it has no way to model *instances* of classes [45]! That means that while we can model the abstract relationships between types of elements, we can't actually create a complete model realization of our system. To overcome this, we then develop an OWL knowledge model for our system, comprising both the T-box and A-box. With named individuals instantiated in the A-box, it will be possible for us to actually change the configuration of our system.

How do we translate from the SysML model into the OWL knowledge model? Some research exists into automating this process. Graves provides a theoretical framework for translating Block Definition diagrams into OWL classes: his paper is essentially a formal confirmation of the correctness of the manual approach one would naturally take [40]. Wardhana et al. present a more detailed framework for automatically translating a Requirements diagram into OWL [86]. However, their approach does not meet our needs. They translate the hierarchical structure of the Requirements diagram into a class hierarchy in OWL. However, what we need is a knowledge model which provides a context to which our requirements can refer. So we take another approach.

First, we translate the Block Definition diagrams into OWL classes, essentially as in Graves. We then enrich these classes with relations, starting with the subclass and association relations in the Block Definition diagram. We translate cardinality annotations in SysML into functionality or cardinality constraints in OWL. We then turn to the requirements in the SysML model. We do not represent the requirements directly in OWL, but will instead represent them externally in a query language which we apply *to* the knowledge model. However, we will need to represent anything to which the requirements refer so that our queries will have something to look for. We represent state using data properties, and create additional facts (ideally using automated inference) to represent anything else mentioned in a requirement. We summarize this correspondence in Table 4.1.

| **SysML element** | **OWL element** |
|---|---|
| Block Definition diagram | Classes |
| Block Definition diagram subclass and association relations | Object Properties |
| Block Definition diagram relation cardinalities | Cardinality and functinality constraints |
| Requirement | (Not represented in the knowledge model) |
| State Machine «refines» Requirement | Data properties modeling the state of an individual |
| (Anything else referred to in a requirement) | Inference rules |

Table 4.1: Correspondence between SysML and OWL elements

While this semi-manual process is cumbersome, it reflects the potential complexity of requirements: we are asked to represent what a requirement *means*, and that interpretive task is far too complex and open-ended to automate reliably.

## 4.6 Summary

We have identified two promising approaches for system modeling. The first is a structured system model using SysML. This has the advantage of foregrounding our requirements and allowing us to draw explicit connections between our requirements and the other system elements which realize them. When we change our system, we will be able to see the effects of changes in one domain on other domains. It will give us a model which is useful for our understanding of the system. The second is a knowledge base using OWL. This has the advantage of permitting rich automated reasoning, and flexibly describing the details of our system. When we change our system, we will be able to automatically check if the system model is still consistent (i.e., satisfying its safety requirements), or else see the problems. In essence, we will use the SysML model for our own design process and sanity checking, and OWL for our actual technical processes.

Comparing these capabilities to our criteria, we can see that these two technologies together should allow us to achieve all of our goals.

We will see system modeling in action in Chapter 7, when we actually create our system model and apply our chosen verification approach to its different configurations. However, before that, we will need to learn about automated reasoning and choose the approach or approaches which will best suit our project.

CHAPTER 5

# Automated Reasoning

In this chapter, we will first introduce high-level concepts about automated reasoning in general. Then we will look at automated reasoning technologies directly applicable to the Web Ontology Language (OWL). Finally, we evaluate the different technologies, and choose the ones we will use.

## 5.1 About Automated Reasoning

We begin with a high-level look at automated reasoning. We first look at the broad capabilities of automated reasoning, and then examine limitations of automated reasoning which are relevant to our work.

### 5.1.1 What is Automated Reasoning?

Automated reasoning is the domain of computer science focused on building computer systems which can prove or disprove given statements relative to some specified set of axioms [65]. It has a wide range of application fields, including hardware and software design, mathematics, and philosophy. For our purposes, the statement of interest is "the industrial system satisfies its safety requirements". Our process concludes by either proving or disproving this claim.

### 5.1.2 Kinds of Reasoning Solution

It will help us in our discussion of automated reasoning to distinguish three kinds of reasoning task:

- Query: Check (perhaps a very large number of) items to see if they fit a certain pattern. While a simple glob expression like `*.txt` doesn't seem to warrant the

"automated reasoning" title, Structured Query Language (SQL) database queries and regular expressions can reach gigantic levels of complexity, and the details of the underlying reasoning engine become relevant.

- Verification: Given a system, prove that it satisfies certain properties. This is perhaps most encountered in computer programming: produce a proof that a certain computer program is *correct* – that is, that it always satisfies certain post conditions for all allowed preconditions.

- Inference: Given a set of facts, generate new facts. This can range from basic properties like reflexive and transitive relations to much more complicated rules-based inference.

All three types of task are potentially useful for us. Inference greatly streamlines the creation of an OWL knowledge base: we can use properties on relations like reflexivity and inversion to quickly fill out appropriate domains and ranges; we can use transitivity to automatically expand our class structure and make sure that all class relations are represented as OWL facts. Verification is ultimately our goal: the property we wish to verify is the system's consistency with its safety requirements. However, we may find it useful to think of verification as a query task: identify all instances of rule violations. As we can expect to encounter all three types of task, it will be good to have the distinction in mind. This will be especially relevant when we examine particular OWL reasoning solutions, each of which has its own purpose.

With the capabilities of automated reasoning in mind, we now turn to the limitations.

### 5.1.3 Limitations on Automated Reasoning

There are a number of limitations on what automated reasoning tools can do, both in theory and in practice. We examine both the general theoretical limitations and the limitations specific to OWL.

#### Decidability

We would like our automated reasoning tool to always work. However, in general, this is too much to ask. Gödel's Second Incompleteness Theorem makes this painfully clear in mathematics; the Halting Problem is the most famous example in informatics. Even practical reasoning problems are undecidable in general, such as determining whether two SQL queries are equivalent [15]. It should be obvious why we want our automated reasoning solution to be decidable. However, we may compromise by limiting the scope of problems that we feed to our solution, or by accepting that in certain marginal cases, failure becomes likely, and building a system which can approach such failures of automated reasoning by other means.

#### Speed

Our tool is useless to us if every query takes a year (or a trillion trillion years) to run. Of course we can make our knowledge base and our queries as complex as we wish, and arbitrarily increase run time. What we really want is for our solution to be fast in practical cases.

#### Expressiveness

We would like to be able to express complicated relationships and queries and have our automated reasoning tool answer difficult questions. However, as we will see in the next section, there is a direct tradeoff between expressiveness and decidability: a logical language sophisticated enough to say everything we might think of definitely won't be decidable in general.

#### OWL-Specific Limitations

**Open-world assumption**: When we look into the details of reasoning with OWL, we will have to take note of two special limitations. First, OWL uses an open-world assumption: the absence of a property P(x) among the facts of the ontology does not imply the fact not-P(x). To borrow from the Pizza Tutorial [29], we may find it difficult to usefully define a vegetarian pizza, as the absence of a `hasTopping MeatTopping` fact doesn't imply any other fact that we can use in the definition. This is less a bug than a feature: OWL is designed for a web environment in which the absence of yet-unknown information does not imply falsehood, and new information may be added at any time.

**Monotonic reasoning**: The addition of information over time also relates to the second major limitation we will encounter in OWL: it does not support **non-monotonic reasoning**. That is, the logical values of individual elements cannot change. For example, while we can define `clifford hasColor Red`, there is no way we can then say `clifford not-hasColor Red` and negate the first fact. This also derives from the incomplete information assumption of the semantic web: what if we later find out that clifford is in fact red? We can always add new facts, but we can't negate or change facts as we go. This has wider implications: OWL struggles to handle uncertainty.

Lack of a **unique name assumption**: In many reasoning contexts, we assume that each named individual is distinct. However, as OWL is built for knowledge integration, it does *not* make this assumption. This means that we cannot assume difference. Instead, where distinguishing individuals is important, it must be encoded with `owl:sameAs` and `owl:differentFrom`.

## 5.2 Reasoning With OWL

Within the the Resource Description Framework (RDF)/RDF Schema (RDFS)/OWL knowledge engineering stack, a wide variety of automated reasoning solutions are avail-

able, each tailored to different applications. We will first look at the formal reasoning underpinnings of OWL and explore their implications for the design of OWL itself. Then, we will examine a list of available reasoning languages and tools.

### 5.2.1 Choice of Logic(s) for OWL

Where RDF is simply a format for a graph database, OWL is a language with support for reasoning. In order to understand its capabilities, we need to also understand the capabilities of two well-known types of logic: propositional logic and first-order logic. This is because the logic(s) used by various (flavors of) OWL arises a compromise between the tractability of the former and the expressiveness of the latter.

#### Propositional Logic

Probably the simplest and best-known formal reasoning language is propositional logic: statements composed of atomic statements (a, b, c, and so on) and connecting operators (and, or, implies, and so on). Despite its simplicity, propositional logic underlies important mathematical results with surprising complexity. The satisfiability problem (SAT) is a good example. Given a sentence in propositional logic using atoms a, b, c, and so on, find an assignment of truth values for a, b, c, and so on, which make the sentence true. SAT was the first problem to be determined to be NP-hard: it is the prototype for the class of problems widely believed (though not proven) to admit no solution algorithm which runs in polynomial time.

Despite its computational difficulty, the SAT problem makes clear a fundamental fact about propositional logic: the satisfiability of its sentences is definitely, unambiguously *computable*: the naive solution algorithm of testing every value for every atom runs in $O(2^n)$ time. Of course we would like the logical underpinning of our knowledge base to be decidable: our queries should have answers!

However nice propositional logic is, it has significant weaknesses for knowledge engineering. In particular, it lacks existential quantifiers (“there exists” and “for all”) and is unable to form statements like “there is a dog such that the color of the dog is red”. It also lacks a concept of equality. Considering both of these, we might see that more deeply, propositional logic lacks the concept of *variables*. To reach this level of expressivity, we move on to first-order logic.

#### First-Order Logic

Extending the symbols of propositional logic with variables, quantifiers for variables, and, potentially, equality relations, gives us First-Order Logic (FOL) (or, potentially, FOL with equality). This is, in essence, the logical system underlying all of mathematics and programming. This immediately highlights its strength and its weakness. The strength is expressiveness: nearly anything we would be interested in saying can be said in FOL. However, its most fundamental weakness is decidability: there is no general decision procedure for FOL statements.

It is worth noting that, as the name implies, higher-order logics are possible. For example, while FOL sentences can use quantified variables, they cannot use quantified properties. So, for example, contrast the following two statements:

1. "There exists x such that isBig(x) and isRed(x) both hold."

2. "There exists a property P such that P(x) holds if and only if isBig(x) and isRed(x)."

The first sentence is expressible in FOL; the second is not, since there is no syntax for quantifying over properties. (The second is, in fact, quantifiable in *second*-order logic.) Since we may actually wish to define properties by inference in some knowledge-engineering cases, this has the potential to be a problem.

### Description Logics

The contrast between propositional logic and first-order logic sets the stage for our question: what logics might we find which lie "in between" these two extremes and capture both of their benefits? In particular, we want a high level of expressiveness together with decidable and relatively tractable solution algorithms. Note that "in between" should not be taken too literally: in light of the limitations of first-order logic, we may actually wish to capture certain types of expressiveness which exceed even its abilities. But as a basic motivating description, it is a good way to think about where the different Description Logics (DL) come from.

And, indeed, there are *different* description logics! Many of them, generally characterized by the specific reasoning capabilities which they provide. The most basic (and first) DL is usually denoted ALC: it allows union and intersection of classes, as well as negations of classes, but does not support negation of relations. It is a fragment of two-variable FOL, which is known to be decidable. We will see many other examples in the next section, as we introduce the different types of OWL: indeed, they are different precisely because of their choice of DL.

A theoretical note: The various flavors of OWL actually use DL which include an additional capability that goes beyond FOL: this is *transitive closure*: the ability to specify the smallest transitive relation which contains a given relation. For example, the relation from City to City "canReachByDirectFlight" is clearly not transitive; however, the relation "canReachBySequenceOfFlights" is transitive, if nothing else by concatenation. This second relation contains the first: any ordered pair of cities related by "canReachByDirectFlight" are definitely related by "canReachBySequenceOfFlights", if nothing else by the sequence consisting of only one flight. In this way, "canReachBySequenceOfFlights" is the transitive closure of "canReachByDirectFlight". OWL flavors support this, as it is a very useful property in knowledge engineering, even though FOL does not.

### 5.2.2 Flavors of OWL

OWL is defined as a standard by the World Wide Web Consortium (W3C), and, as such, has multiple defined versions and subsets. At the top level, a newer OWL 2 replaced the original OWL in 2012 [82]. Most of the changes related to usability rather than logic. For example, OWL 2 uses IRIs instead of URIs for namespace identifiers, permitting non-English characters such as Chinese *hanzi*. However, there are significant logical updates. In particular, OWL 2 supports declaring classes as disjoint: this allows a significant level of built-in error checking, as it gives the default open-world class reasoning in OWL something to conflict with.

At a more granular level, both OWL and OWL 2 include defined language subsets, each of which corresponds to a certain DL with a defined level of expressiveness and a proven level of tractability.

The original OWL specification defines three *variants*: OWL Full, OWL DL, and OWL Lite. OWL Full comprises the complete set of OWL structures, with no additional restrictions. It is thus the most expressive variant. However, reasoning problems are not necessarily decidable: OWL Full makes no computational guarantees.

OWL DL is based on the SROIQ description logic. To achieve this set level of expressivity and tractability, it enforces several restrictions on OWL Full [80]. The most notable of these:

- Top-level types are disjoint. So, for example, an element cannot be both a class and an individual. Contrast this with plain RDFS, where there are *no* restrictions on what triples can be formed.
- Object properties and datatype properties are disjoint. That means built-ins like `owl:inverseOf` which map object property to object property cannot be used on datatype properties.
- All classes referred to must be explicitly declared.
- Equality and difference facts can only apply to named individuals.

OWL Lite dramatically reduces the available expressiveness, resulting in a minimum usable language which can be processed very quickly. Among the losses are `owl:oneOf`, `owl:unionOf`, `owl:complementOf`, `owl:hasValue`, `owl:disjointWith`, `owl:DataRange`. Basically, OWL Lite allows for reasoning about class hierarchies, property restrictions, and optional/required properties [80].

With OWL 2, the standard shifts focus from *variants* to *profiles*. While the variants were defined in expectation of reasoner development, the profiles support specific use cases and reasoning tools developed in the interval between the two standards. Each profile is a subset of OWL 2 functionality which has already been demonstrated to be tractable in its intended use case [83]:

- OWL 2 EL: Ideal for complex class and property hierarchies. Scales well with T-box complexity.
- OWL 2 QL: Ideal for query-style tasks – scales well with A-box complexity.
- OWL 2 RL: A subset of OWL 2 DL optimized for polynomial-time reasoning with respect to the size of the ontology.

OWL Lite and OWL DL can also be considered profiles of OWL 2.

### 5.2.3 Reasoning Languages for OWL

A number of reasoning languages have been developed for use with OWL and other Semantic Web technologies. We examine a selection of those which are potentially relevant to our project.

#### Description Logics

As we have described above, OWL is based on certain flavors of DL. Thus, it is automatically amenable to DL. We can use DL both to form queries and for inference. The query use case is straightforward: for example, in the Pizza Tutorial ontology [29], the query `hasTopping some (hasSpicinessValue Hot)` does exactly what one might expect, returning all pizzas with a spicy topping. However, DL can also be used for inference through *restrictions*: logical expressions defining classes. So, we could use a DL expression to define the class `MargheritaPizza` as the subclass of `Pizza` with some topping from `CheeseTopping`, some topping from `TomatoTopping`, and only toppings from `(CheeseTopping or TomatoTopping)`. Note that because of the open world assumption, we need to specify both necessary and sufficient conditions.

#### The SPARQL Protocol and RDF Query Language

The SPARQL Protocol and RDF Query Language (SPARQL) is a query language for RDF knowledge bases. Since OWL is built on top of RDF, SPARQL is naturally suited to OWL. Like SQL, SPARQL allows for construction of arbitrarily complex queries. It even offers the potential for non-monotonic reasoning, as it can add facts using the `INSERT` statement. However, Protégé does not support `INSERT`, so we don't consider this. Another significant limitation is in Protégé, SPARQL queries do not recognize facts asserted by the reasoner, such as domains for inverse properties. It is possible to work around this, for example by exporting asserted facts as axioms. But it is nonetheless important to note.

SPARQL is highly expressive: in fact its expressivity is equal to that of Relational Algebra—that is, as expressive as one might ask a query language to be [19]. However, it has potentially high complexity: in general, deciding whether a SPARQL query has a result is PSPACE-hard (harder than NP) with respect to the size of the query or the

combined size of the query and knowledge base [55]. However, the query problem is in the far more tractable class when the query is simple: it is NL-complete (easier than P) with respect to the size of the knowledge base only. In practice, queries are not very complex, despite the gigantic complexity of knowledge bases. Thus, practical query problems fall within this very tractable problem class.

#### The Semantic Web Rule Language

The Semantic Web Rule Language (SWRL) is a rule language: it asserts new facts according to rules. For example, we could assert that if `x hasParent y` and `y hasBrother z`, then `x hasUncle z` [63]. A reasoner using SWRL would then add this last fact to the ontology whenever the antecedent pattern matched.

#### The Semantic Query-Enhanced Web Rule Language

While SWRL is not a query language, an antecedent in SWRL can be viewed as a query. The Semantic Query-Enhanced Web Rule Language (SQWRL) takes this approach by replacing the consequent with a SQL- or SPARQL-style command like `sqwrl:select` or `sqwrl:orderBy`. The syntax is dramatically simpler than that in SPARQL, and offers the possibility to quickly turn inference conditions expressed in SWRL into queries, giving flexible transition between inference and query.

#### The Shapes Constraint Language

The open world assumption means that OWL lacks good mechanisms for data validation: all facts are valid unless very, very specifically invalidated. The Shapes Constraint Language (SHACL) provides a mechanism for validating an RDF knowledge base. In addition native SHACL syntax, a SPARQL extension, SHACL-SPARQL is defined, allowing the use of more complex SPARQL queries in a validation process. In terms of our identified types of task, we can view SHACL as a tool for verification: ensuring that a knowledge base meets certain requirements.

## 5.3 Automated Reasoning Approach

We now need to choose exactly how we will use automated reasoning in our project. We first consider the choice of the reasoning tool itself, and then the suitability of different reasoning languages to different kinds of tasks.

### 5.3.1 Choice of Reasoner

A bewildering variety of reasoners have been developed for OWL. In 2020, W3C listed 22 reasoners on the OWL wiki. Fortunately, we may immediately limit our pool to those with open source licenses and, since we have already chosen to use Protégé, support for Protégé. Abburu compares nine of the more notable reasoners, giving us a helpful

overview of their capabilities [14]. Our first two crite1ria reduce us to five: Pellet, FACT++, HermiT, CEL, and ELK. Examining their capabilities, we reject FACT++, CEL, and ELK because they do not support SWRL rules. Then, between Pellet and HermiT, we note that HermiT does not provide justifications for its results, while Pellet does. Thus, we choose Pellet: it is open-source, supported in Protégé, supports SWRL, and provides justifications.

### 5.3.2 Choice of Reasoning Languages

We want a set of tools which can test whether an instance of our system meets our safety requirements. For this, we will need to:

1. Make our safety requirements expressible by inferring the facts and applying constraints.
2. Formalize each safety requirement using a formal reasoning language.
3. Check each safety requirement using a reasoner, for each test case.

This translates into a number of automated reasoning tasks:

- Rule inference
- Validation
- Query for violations
- Automation

What technology should we use for them? We can see we have several general needs:

- Expressiveness sufficient to model safety requirements
- Ability to infer new facts corresponding to concepts used in the safety requirements
- Ability to validate against facts – in particular, negation and closed-world reasoning
- Ability to cleanly use existential constructs
- Readability and ease of use
- Practical tractability
- Good integration with Protégé

Because each of these needs is only applicable to some contexts, and because the decision to use any given technology will come with qualifiers based on the context, we will proceed with a holistic analysis of the available technologies in each context, rather than a point-by-point evaluation.

### Rule Inference

By *inference*, we mean introducing new facts to our knowledge base according to rules. We apply a qualitative pros-and-cons analysis to our available inference technologies in Table 5.1.

| **Inference technology** | **Pros** | **Cons** | **Decision** |
|---|---|---|---|
| Pellet (Protégé built-in DL) | Easy class and property inference in Protégé. | Limited to defined tasks related to class relationships and object properties. | Use to simplify data entry (e.g. inverse relationships for object properties). |
| SPARQL | Highly expressive, like SQL. | `INSERT` command not supported in Protégé. | Do not use. |
| SWRL | Readable syntax. Well-suited to existential constructs. Protégé reasoner incorporates inferred facts directly. | Open-world assumption: can't support any kind of negation. | Use to infer new facts. |
| SHACL | Highly expressive. Closed-world assumption. | New facts are confined to the SHACL process itself: Protégé can't update using them. Difficult syntax. | Do not use. |

Table 5.1: Analysis of relevant inference technologies

Beyond the very basic capabilities of Pellet (built in to Protégé), what do we use for inference? SWRL, not SHACL. Why? SHACL can also be used to infer new facts. However, the cost of this is that SHACL rules are considerably more complicated, as, like SPARQL, it operates at the lower level of RDF triples. And, from the practical point of view, the Protégé reasoner does not update the ontology with inferred facts from SHACL. In contrast, SWRL is easily readable, and the reasoner adds its inferred facts along with the rest.

We will also use one other Protégé feature: because not all technologies recognize inferred facts under Protégé's built-in inference engine, we will use the Export Inferred Axioms feature to preprocess each OWL knowledge model, ensuring that each inferred fact receives a literal representation.

### Validation

We wish to validate that our knowledge model is consistent and represents our real-world system in an acceptable way. We consider both validation and query technologies, since queries could be used to identify validation violations. We apply a qualitative pros-and-cons analysis to our available validation technologies in Table 5.2.

| **Validation technology** | **Pros** | **Cons** | **Decision** |
|---|---|---|---|
| Pellet (Protégé built-in DL) | Simple and readable. Decidable. | Limited expressiveness. Open-world assumption in OWL. Must use Protégé to specify all classes disjoint, all individuals distinct. | Use during data entry to prevent mistakes. |
| SPARQL (Protégé built-in) | Strongly expressive and tractable in real-world use cases. | Unable to consider inferred facts. High complexity and low readability. | Do not use. |
| Snap-SPARQL | As above, and accesses inferred facts from Protégé reasoner. | Does not support the `FILTER NOT EXISTS` construct. | Do not use. |
| SHACL (Protégé built-in) | Closed-world assumption. Highly expressive. | Difficult syntax. | Use for most validation tasks. Use to apply closed-world filter over inferred rules. |
| Jena SHACL | Convenient for command-line execution. | No editor. | Use for batch processing. |

Table 5.2: Analysis of relevant validation technologies

Most of our validation tasks amount to preventing user error at the data entry stage. We will encounter an example of this during our Proof of Concept application. However, a more critical task for validation relates to closed-world inference. For example, if we wish to have a "disconnected" relation, we cannot use SWRL to infer that client A and server B are disconnected from their lack of a "connected" relation. A workaround is to make related inferences and filter failures using SHACL:

1. Assign “disconnected == true” to all clients pairs.
2. Use a SHACL rule to invalidate all “connected” relations where A is connected to B and A has “disconnected == true”.
3. Manually fix the model until only valid “disconnected” values remain.

While this method is a bit ugly, it gets us around the open-world restrictions of SWRL. In the Proof of Concept application, we will apply this approach to modeling visibility.

So, for validation, we use Protégé’s built-in DL reasoning over disjoint classes and distinct individuals, and we use SHACL shapes for more complex tasks, especially those requiring closed-world reasoning. We use Protégé’s SHACL tab while creating our shapes, and Jena to execute SHACL validation at scale.

### Query for Violations

Our main query task is to identify violations of our safety requirements. We apply a qualitative pros-and-cons analysis to our available query technologies in Table 5.3.

| **Query technology** | **Pros** | **Cons** | **Decision** |
|---|---|---|---|
| SPARQL (Protégé built-in) | Strongly expressive and tractable in real-world use cases. | Unable to consider inferred facts. High complexity and low readability. | Do not use. |
| Snap-SPARQL | As above, and accesses inferred facts from Protégé reasoner | Does not support the `FILTER NOT EXISTS` construct. | Use to interactively build SPARQL queries. |
| Jena SPARQL | Convenient for command-line execution. | No editor. | Use for batch processing. |
| SQWRL | Easy SWRL-style syntax. | Strictly open-world, like SWRL. | Do not use. |
| SHACL-SPARQL | Closed-world assumption. Highly expressive. | Difficult syntax. No use of `MINUS` construct. | Do not use. |

Table 5.3: Analysis of relevant query technologies

We are tempted to use SHACL to enforce safety requirements. However, its limitations make it too rigid for this task. And while SHACL can actually get around this by including SPARQL queries, in practice this proved to be too cumbersome for us to

actually use. Further, SHACL-SPARQL does not support the `MINUS` construct, closing off our last hope for closed-world reasoning.

SPARQL instead shows its strength here. We can recast each requirement as a query for its violation, and write a SPARQL query for this. SPARQL also provides us limited support for closed-world reasoning. We use the `MINUS` construct to provide for atomic negation: to select for all Object which do *not* have propertyX, we select (Object) `MINUS` (Object and propertyX).

We choose Snap-SPARQL for use within Protégé because it can automatically detect inferred facts. This will be especially convenient for interactively designing our queries. For batch processing, we will use Jena SPARQL.

### Automation

We find it useful to introduce Apache Jena into our tech stack. While Protégé and its plugins provide a good user interface for building the knowledge base, and a convenient way to run SWRL rules generation, SHACL validation, and SPARQL queries, Protégé does not have a command line interface and is thus challenging to automate. In particular, Protégé can only run one SPARQL query at a time. We will need more. Jena, an open-source knowledge engineering framework, provides command line tools sparql and shacl which will let us perform our validation and verification tasks in simple Bash scripts. To ensure that Jena can detect the inferred facts, we will use Protégé's Export Inferred Axioms option to create each of our test case knowledge base files.

This automated section is essentially functional, as we can see in Table 5.4.

| **Inputs** | **Outputs** |
|---|---|
| `test_case.rdf`<br>`shacl_shapes.ttl`<br>`safety_requirements.rq` | `STDOUT`<br>Return code |

Table 5.4: Inputs and outputs of automated process

Of the three input files, only the RDF for the configuration actually changes from test case to test case. The SHACL shapes and the SPARQL queries are fixed. We can automate this in a simple bash script. This part of the process only requires Jena and Bash.

It is important to use that whenever a violation occurs, our automated reasoning solution provides a human-readable explanation. With SHACL, this is easy: we use `sh:message` and write the appropriate message. SPARQL is much trickier: there is no in-built capacity for this kind of metadata. We use comments as a workaround: in our Bash script, comment lines beginning `# Requirement` will be printed for each test, while comment lines beginning `# Violation` will be printed in the event of a violation. We accomplish this with a simple grep call.

### Technology Choices

We summarize our choices of technology in Table 5.5.

| Task | Technology |
|---|---|
| Rule inference | SWRL, Pellet (Protégé built-in DL), Protégé "Export Inferred Axioms" function |
| Validation | SHACL, Pellet (Protégé built-in DL), Protégé disjoint classes and "Make all individuals different" |
| Query for violations | SPARQL: Snap-SPARQL for design; Jena SPARQL for batch processing |
| Automation | Jena, Bash |

Table 5.5: Technology choices for automated reasoning

## 5.4 Summary

Having developed a deeper understanding of how automated reasoning can help with our task, and the major limitations we face, we have chosen a set of tools which will allow us to develop our system model with requirements and then check its consistency with its requirements. In the next short chapter, we put the pieces together, giving the detailed step-by-step process which our design methodology has produced.

CHAPTER 6

# Design Artifact

With our design decisions made, we are ready to detail our design artifact—our **process for evaluating an industrial system for consistency with its safety requirements**. Our process takes an industrial system and a set of safety requirements as its inputs and produces a human-readable verification result as its output.

1. Input: Identify input industrial system and safety requirements.
2. System Model (Pen & Paper): Produce a model using the Systems Modeling Language (SysML), including its requirements:
   a) Model the requirements in a **Requirements diagram**.
   b) Introduce **State Machine diagrams** to clarify state-base requirements.
   c) Identify interrelations between requirements using **«derives» and «refines» relations**.
   d) Model the components of the industrial system in a **Block Definition diagram**.
   e) Elaborate on **subclass and association relations with cardinalities**.
   f) Identify relations between components and requirements using **«satisfies» relations**.

3. Knowledge Model (Protégé): Produce a knowledge base using OWL, with classes based on our SysML model and named individuals based on our input industrial system.
    a) Create a new ontology.
    b) Introduce **classes** for the components, based upon the SysML model.
    c) Based on the Block Definition diagram subclass and association relations, introduce **object properties** needed to relate components to each other.
    d) Use OWL inverse and cardinality properties to control the object properties and allow for model validation.
    e) Based on the requirements and their State Machine diagrams and satisfying components, introduce **data properties** to track the states of component instances relevant to requirements.
    f) Using SHACL, introduce **SHACL shapes** to ensure the validity of the knowledge model.
4. Inferring facts (Protégé): Enhance the OWL knowledge base to infer facts relevant to the requirements
    a) Identify necessary facts, based on the requirements.
    b) Using SWRL, introduce **SWRL rules** to infer new facts.
    c) Filter the inferred facts using further **SHACL shapes** when closed-world reasoning is strictly necessary.
5. Query design (Protégé):
    a) Translate each requirement into a **query** for SPARQL.
    b) Comment each query with a human-readable description and a human-readable violation message.
6. Instantiation (Protégé):
    a) Introduce **named individuals** to instantiate the model, based on the input industrial system.
    b) When comparing multiple configurations, produce a separate instantiated RDF file for each test case.
    c) Use Protégé's **Export Inferred Axioms** feature in each RDF file to make sure all inferred facts are reflected in the knowledge base.

7. Collect files:

    a) `test_case_[n].rdf`, our fully developed OWL models, including A-box named individuals. One model per configuration being evaluated.

    b) `shacl_shapes.ttl`, our model validation SHACL shapes.

    c) `safety_requirements.rq`, our safety requirements expressed as SPARQL queries.

8. Execution (Jena, Bash):

    a) Use **Jena SHACL** to validate each test case against the SHACL shapes.

    b) Use **Jena SPARQL** to test each test case for compliance with the safety requirements.

9. Output: **Text report** confirming consistency or else detailing violations.

We also provide a visual overview of this process, in the form of a SysML Activity diagram, in Figure 6.1.

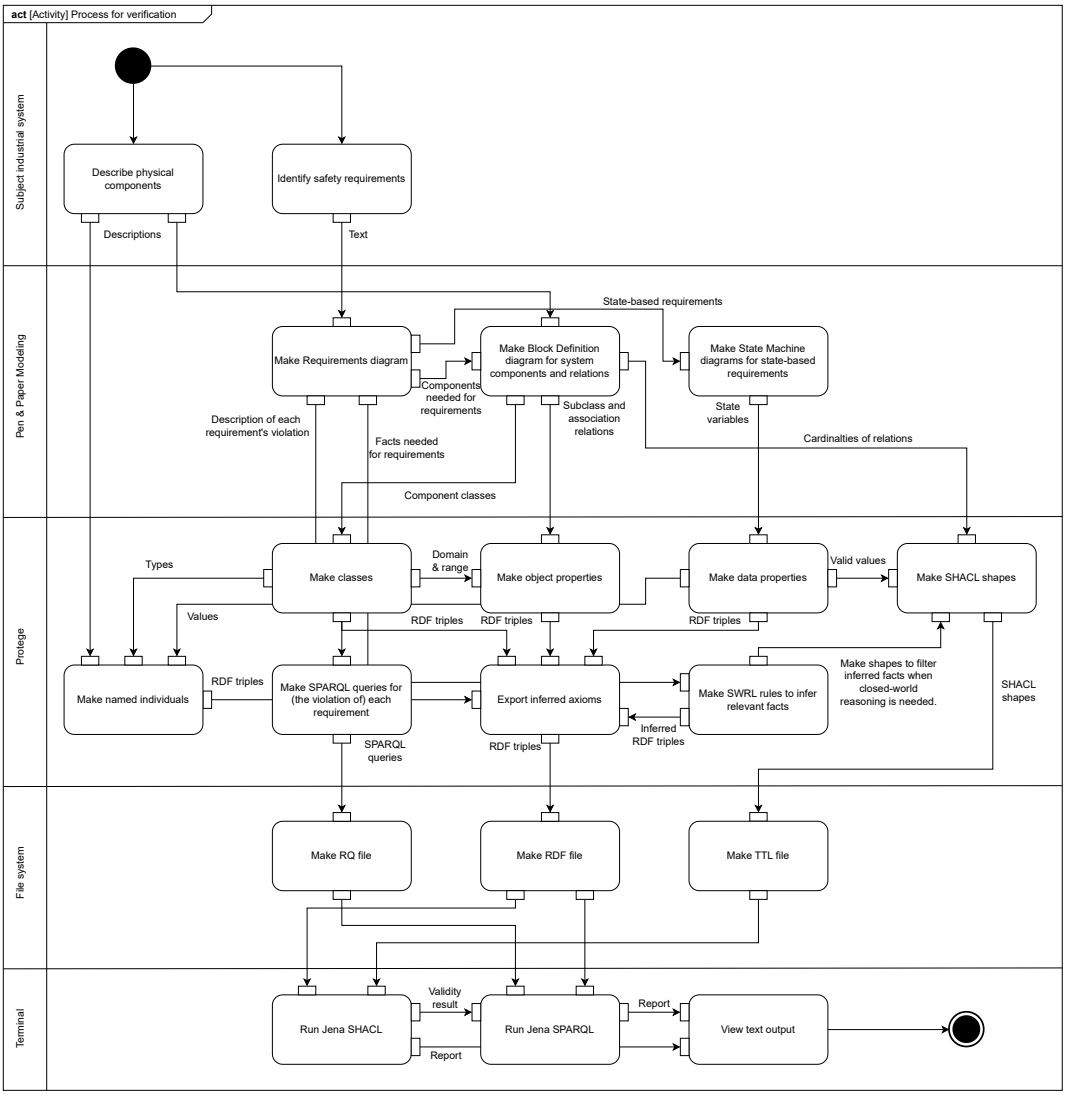


Figure 6.1: Activity diagram: Process for verification

# CHAPTER 7

# Proof of Concept

In this section, we apply the nine-step process developed as our artifact to the industrial system described in our scenario. We will apply the process to several test cases so that we can evaluate it in terms of *reconfigurable* systems, per the Industrie 4.0 goals.

## 7.1 Test Cases

We will apply our reasoning application to a series of test cases. A test case changes the instances of our system into a safe or unsafe configuration. We list them in Table 7.1.

| **Test case** | **Description** | **File name** |
|---|---|---|
| Base configuration | | `test_case0.rdf` |
| Disconnected machine | Machine 1 is disconnected from the safety network. | `test_case1.rdf` |
| Wall removal | Now each e-stop can see each machine. | `test_case2.rdf` |
| New machine introduced | Adding a new machine without any other configuration should violate all sorts of requirements. | `test_case3.rdf` |

Table 7.1: Test cases

These four test cases allow us to see if our system really does detect safety violations resulting from reconfiguration steps.

## 7.2 Verification Process

Having defined our test cases, we now carry out the steps of our process. We carefully follow the outline in Chapter 6.

### 7.2.1 Input

We will take the industrial system described in Chapter 3 as our input, along with the three variant test cases listed above. We take the four simplified safety "standards" listed with our scenario as our safety requirements, which we will refine as we build our model.

### 7.2.2 System Model

We begin by building a model using SysML to reflect our scenario. We first model the requirements, and the the components of the industrial system.

#### Requirements Modeling

In our scenario description, we outlined the definition of an e-stop, and we set requirements for the use of e-stops in our system:

- An e-stop button stops all connected machines
- An e-stop button is red and yellow
- One-way door: disengaging an e-stop does not turn on machines.
- Each machine has an integrated e-stop
- Each control panel has an integrated e-stop
- Each emergency stop button must span all machines **visible** from its location.
- Each machine and each e-stop is connected to the safety network
- Machines and e-stops registered on the safety network must not disconnect.

We will start our SysML model with these requirements. We will also add a dummy functional requirement: "assemble product", to represent the value-generating process of our system. We group these requirements for logic and readability. A real model would also link to the specific standards, design decisions, documents, and so on, that lead to these requirements, but we will omit all of this. Thus we get:

- RQ 1: Assemble product
- RQ 2: Correct e-stop functionality

  - RQ 2.1: E-stop stops all connected machines
  - RQ 2.2: E-stop is red and yellow
  - RQ 2.3: One-way door: disengaging e-stop does not turn on machines.
- RQ 3: Correct use of e-stops in system
  - RQ 3.1: Each machine must have an integrated e-stop
  - RQ 3.2: Each control panel must have an e-stop.
  - RQ 3.3: Each e-stop must span all machines visible from its location
- RQ 4: Use of safety network
  - RQ 4.1: Each machine and e-stop must be connected to the safety network
  - RQ 4.2: Devices registered to the network must not disconnect while registered.

In Figure 7.1, we present these requirements in a SysML Requirements diagram.

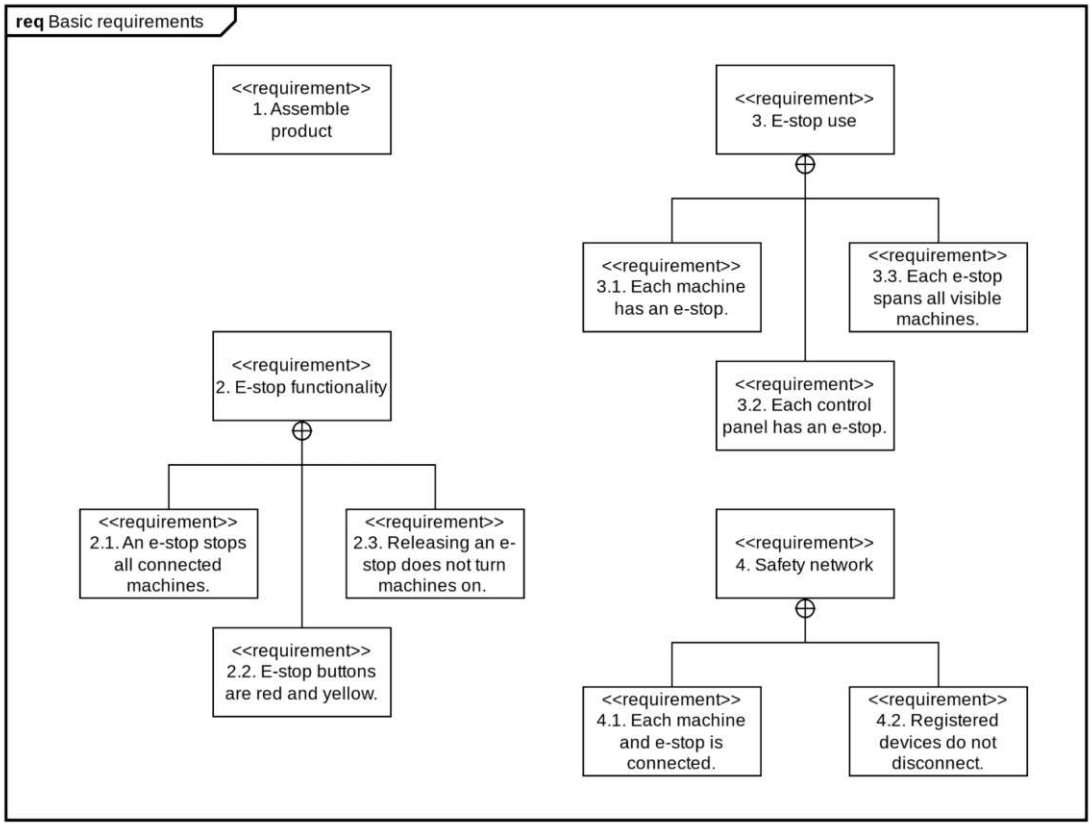


Figure 7.1: Basic requirements for our example system

From this set of requirements, we notice three things: First, this implies specific domains which our model must cover:

- Physical devices: machines, controllers, and e-stop buttons.
- Safety network: abstracted as one object which monitors connected devices and is capable of routing e-stop signals. It includes a safety configuration table: many-to-many assignment of machines to e-stops. It detects connected devices and registers devices that should be connected.
- Locations: semantic container for physical devices. Each physical device belongs to exactly one location.
- Visibility domains – each location belongs to one or more visibility domains. Each visibility domain contains one or more locations.

Second, we may notice that allowing for removable control panels with integrated e-stop leads to some requirements violations. 2.1, 2.3, 3.3, and 4.1 need to be refined to allow for color-changing e-stops; 2.2 and 3.2 need to consider connected/disconnected state. We introduce a new top-level requirement for removable control panels, and «refine» and «derive» our new requirements. Whereas the SysML diagram was basically optional for the numbered outline above, we now actually need one to show the level of interrelation between the requirements: see Figure 7.2.

Third, we see examples of state-based behavior:

- the relationship between engaging/disengaging an e-stop and the on/off state of a machine
- the process of correctly connecting and disconnecting the control panel

We will use State Machine diagrams to illustrate these: see Figures 7.3 and 7.4.

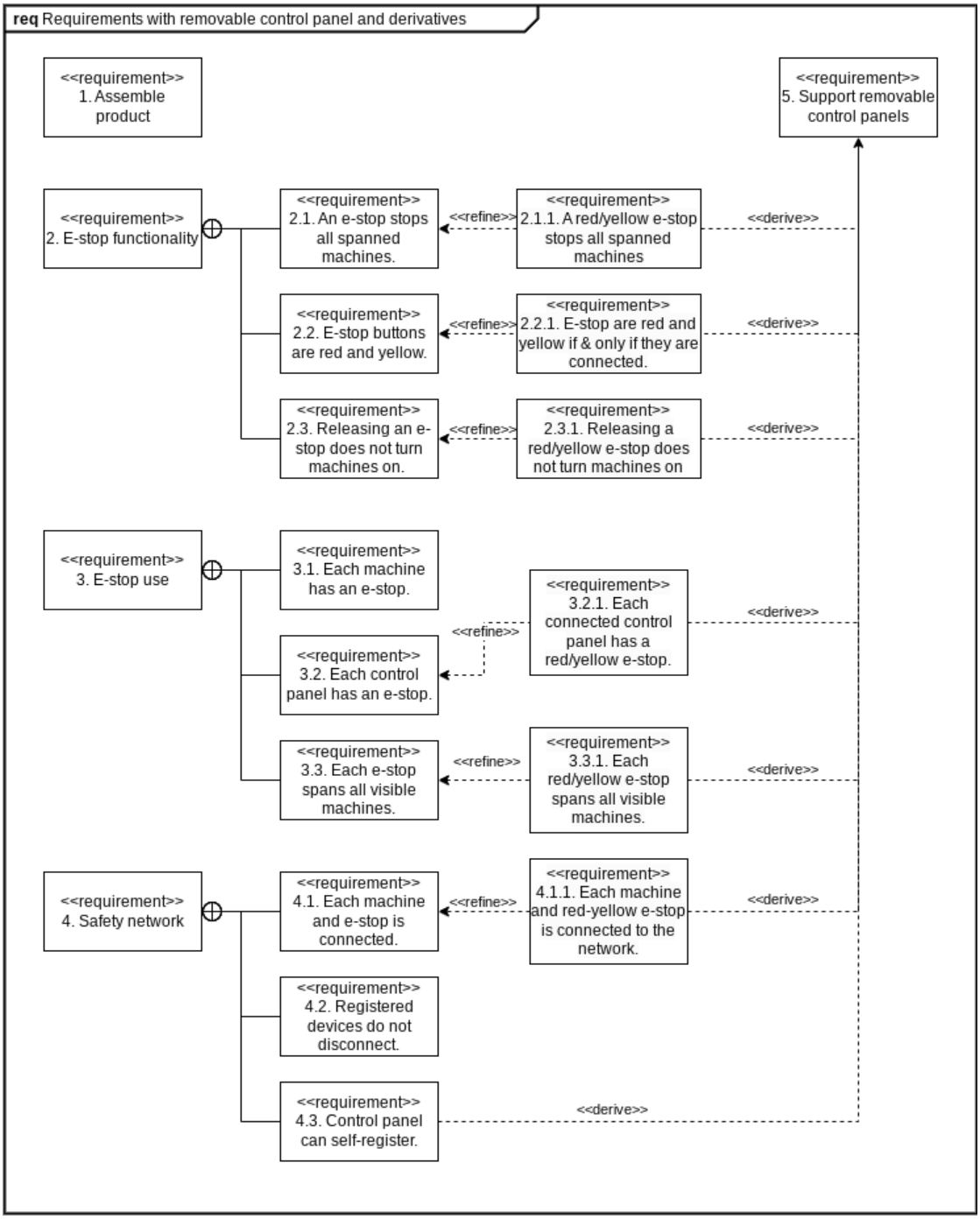


Figure 7.2: Requirements diagram showing «refine» and «derive» relations from new color-changing e-stop requirement

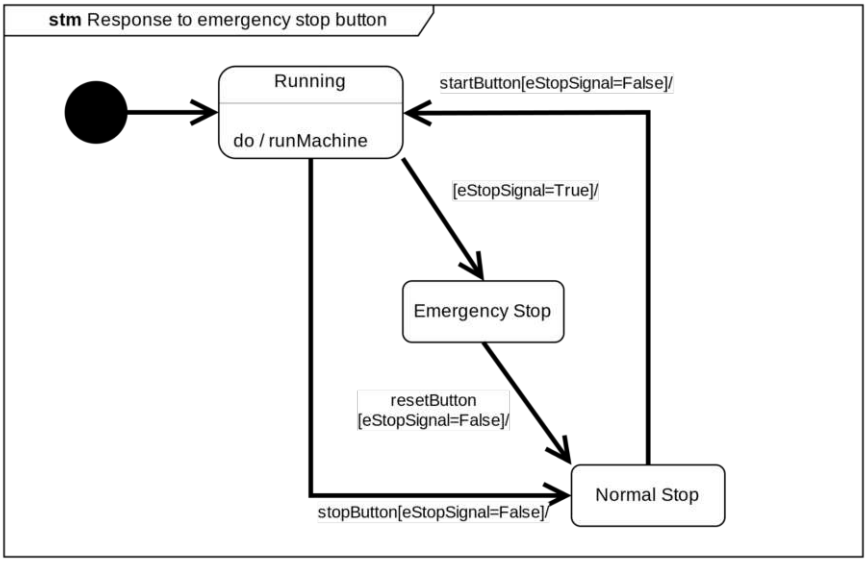


Figure 7.3: State Machine diagram illustrating the response of a machine to an e-stop button

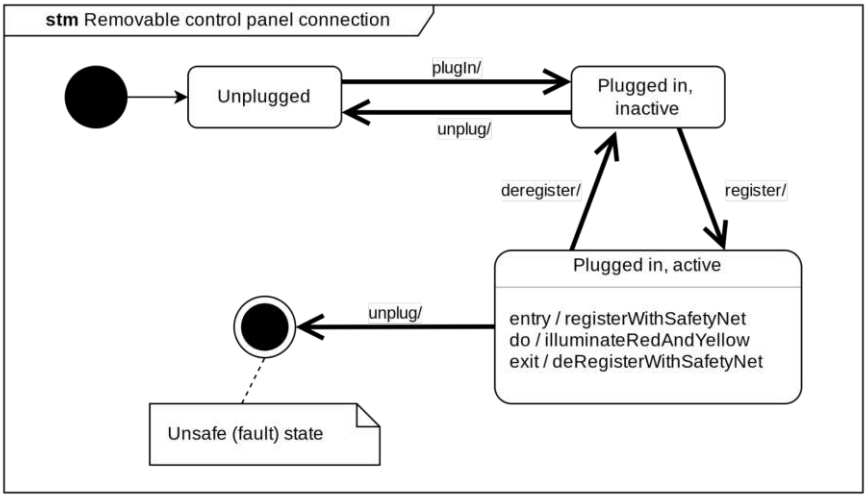


Figure 7.4: State Machine diagram illustrating the connection/disconnection behavior of a removable control panel

A note on SysML syntax: State transitions are denoted “trigger[guard]/”, where the guard is a condition which must be true for the transition to occur. In this way, the Emergency Stop Button Response state machine has relatively few states, as the transitions check the state of the e-stop signal itself.

Finally: one thing we have *not* shown is the set of trace relationships to external standards like ISO 13850: this is because we have only roughly paraphrased these in our scenario design. However, in a real SysML application, this kind of data would be valuable, as it would anchor the specific requirements to the more fundamental stakeholder need (here, standards compliance). This supports maintaining the link from detailed requirements introduced by engineers and the needs of the customer or other stakeholder – without this, there is a major risk of requirements proliferating without actually being valid [27].

### Component Modeling

We present here a detailed set of classes representing the components in the scenario. Note that this shows only the structure, not the detailed function of the system. It will be a helpful aid for us when we instantiate our system as a knowledge base in the next section. We organize our classes and their relations in a SysML Block Definition diagram: see Figure 7.5.

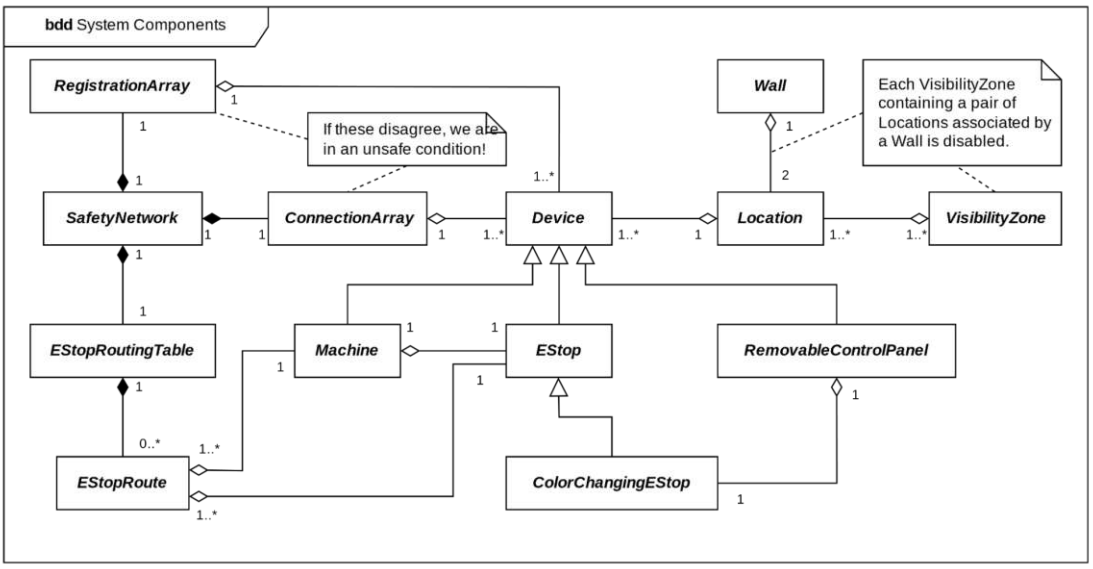


Figure 7.5: Block Definition diagram showing components and relations in our example system

Notes on SysML syntax:

- Open triangle arrows indicate subclass relationships: Machine extends Device, in Java syntax.
- Open diamonds indicate that one component is associated with another. The numbering indicates the number of elements, with "1..*" indicating one or more.

So, an EStopRoute object contains one Machine and one EStop object. Conversely, a Machine object is contained in *one or more* EStopRoute objects.

- Filled diamonds indicate association where the subpart has no independent existence: outside of an EStopRoutingTable object, there is no such thing as an EStopRoute.

- Boxes with folded corners are comments.

#### Component-Requirement Relations

Usefully, SysML provides a syntax with which we can connect requirements to the system elements which satisfy them. In practice, this means that when we change our requirements, we can trace the effects to our system elements. A table provides greater readability than a very tangled diagram: we collect our «satisfies» relations in Table 7.2.

| **Requirement** | **Component «satisfies» Requirement** |
|---|---|
| RQ 1. Assemble product | Machine |
| RQ 2.1.1. A red/yellow e-stop stops all spanned machines | EStop, Machine, EStopRoute, ConnectionArray |
| RQ 2.2.1. E-stops are red and yellow if and only if they are connected. | EStop, ColorChangingEStop, ConnectionArray |
| RQ 2.3.1. Releasing a red/yellow e-stop does not turn machines on | EStop, Machine, stm "Response to emergency stop button" |
| RQ 3.1. Each machine has an e-stop. | EStop, Machine |
| RQ 3.2.1. Each connected control panel has a red/yellow e-stop. | EStop, RemovableControlPanel, ConnectionArray |
| RQ 3.3.1. Each red/yellow e-stop spans all visible machines. | EStop, Machine, EStopRoute, ConnectionArray, VisibiltyZone |
| RQ 4.1.1. Each machine and red-yellow e-stop is connected to the network. | EStop, Machine, ConnectionArray, SafetyNetwork |
| RQ 4.2. Registered devices do not disconnect. | Device, ConnectionArrray, RegistrationArray |
| RQ 4.3. Control panel can self-register. | RemovableControlPanel, ConnectionArray, RegistrationArray, stm "Removable control panel connection" |

Table 7.2: Component «satisfies» requirement relations

### 7.2.3 Knowledge Model

With our SysML model complete, we move on to the next step: building a knowledge model using OWL which will allow us to instantiate our test cases and provide the context in which we formalize our requirements. We begin by creating a new ontology, under

the namespace IRI `http://www.semanticweb.org/donald/ontologies/2024/9/estoppoc`.

### Classes

We begin by introducing our component classes. While OWL has the ability to define classes according to DL rules and infer new class relationships, we have no need of this. Thus, this is a straightforward process of copying our class structure into Protégé. We can start with our Block Definition diagram in Figure 7.5.

As a precaution, we have gone ahead and specified at each level of hierarchy that sibling classes are disjoint. We will see later that this helps the reasoner catch data entry mistakes.

### Object Properties

Our System Components view indicates a number of associations which we will need to model. Modeling an association as a property will be considerably more complicated than modeling a class. Just to capture the information in the System Components view, we will need to specify domain, range, and cardinality restrictions. It is also helpful to use Protégé's support for inverse properties to create logical inverses. So, we will have both estoppoc:hasComponent and estoppoc:isComponentOf, for example.

Parsing our Block Definition diagram, we find the following properties, organized by related component class in Tables 7.3 and 7.4.

Protégé can automatically create inverse relationships, so we'll only fully configure the first of each pair. We will also configure the functionality of each property: a *functional* property takes only one object instance; an inverse-functional property takes only one subject instance. So, for people, `hasHeightInCm` is functional: two people may have the same height, but one person cannot have two heights. A one-to-one property is both of these things. Pellet, the reasoner we are using, does not automatically infer functional/inverse-functional relationships, so we encode them manually.

### Data Properties

Some of our requirements depend on the state of a machine (running or not, emergency-stopped or not) or an e-stop (engaged or not; red/yellow or not). We also use a data property to control visibility, as we describe below. We will introduce boolean-valued data properties to model these states: they are listed in Table 7.5.

| Class | Data Properties | Type | Default |
|---|---|---|---|
| Machine | isRunning | Boolean | True |
| | isInEStopCondition | Boolean | False |
| EStop | isEngaged | Boolean | False |
| | isRedYellow | Boolean | True |
| VisibilityZone | isObstructedVisibilityZone | Boolean | Special case |

Table 7.5: Data properties

Visibility is one of the more challenging concepts to model. We might initially think that we could define an `isVisibleFrom` relation between `Location` instances. This seems simple, and corresponds to our definition of `Wall` instances as adjoining two `Location` instances. Then we could invalidate visibility betweeb the two adjoined locations when a wall occurs.


Figure 7.6: Visibility example with no wall

In Figure 7.6, there is no wall: Location 1 `isVisibleFrom` Location 2 , Location 1 `isVisibleFrom` Location 3, Location 2 `isVisibleFrom` Location 3 (and the symmetric closure).

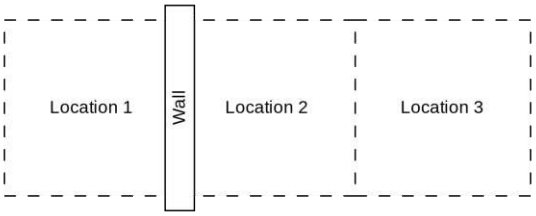


Figure 7.7: Visibility example with wall

Following this idea, in Figure 7.7, we introduce a wall between Location 1 and Location 2, and so we would invalidate Location 1 `isVisibleFrom` Location 2:

~~Location 1 isVisibleFrom Location 2~~ , Location 1 `isVisibleFrom` Location 3, Location 2 `isVisibleFrom` Location 3 (and symmetric closure of these)

This is incorrect: Location 1 `isVisibleFrom` Location 3 is wrong. The simple rule of removing visibility where a wall is creates a serious error. The solution is to introduce a layer of abstraction: a `VisibilityZone` is a set of locations which are all mutually visible. Then we will invalidate a zone if it contains both locations which a wall adjoins, using the `isObstructedVisibilityZone` property.

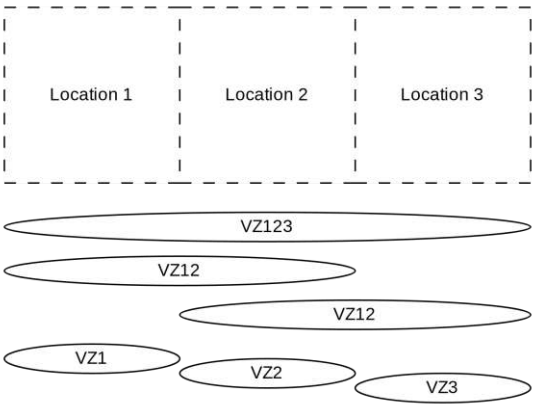


Figure 7.8: Visibility zones with no wall

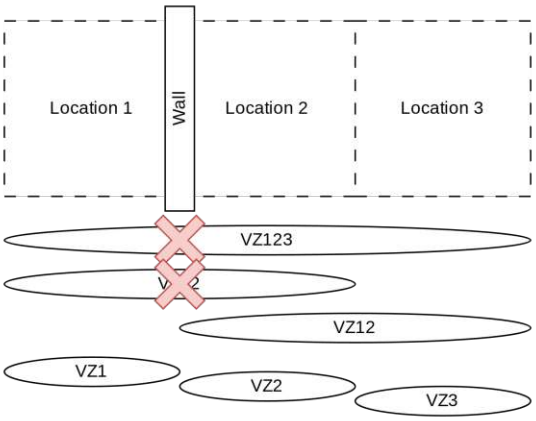


Figure 7.9: Visibility zones with wall

We can see looking at Figures 7.8 and 7.9 that this approach gives the correct result.

#### SHACL Shapes

While OWL's built-in DL features provide some level of model validation, we will use SHACL to apply closed-world constraints. This will in turn allow us to form queries in SPARQL that make sense.

First, we require that all boolean data properties be true or false, never both, never neither. We do this by the expedient of requiring precisely one value, as in Listing 7.1.

```
poc:MachineStateVariablesClosure
  a sh:NodeShape ;
  sh:targetClass poc:Machine ;
  sh:property [
    sh:path poc:isRunning ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:message "Must have exactly one value for isRunning."
  ] ;
  sh:property [
    sh:path poc:isInEStopCondition ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:message "Must have exactly one value for isInEStopCondition."
  ] .
```

Listing 7.1: Boolean closure in SHACL

We could apply more elaborate requirements, but for the purposes of our proof of concept, we stop here.

### 7.2.4 Inferring Facts

Examining the list of requirements above, we see that we need to discuss e-stop span and visibility – especially for

- EStop spans Machine
- Machine spannedBy EStop
- Device isVisibleFrom Device

The first two are straightforward; the last requires a special workaround due to its inherent incompatibility with open-world reasoning and non-monotonic reasoning.

#### Emergency stop span relation

For the first two of our required properties, we create the two object properties and set their domains, ranges, and inverse relationship as usual. We then implement the SWRL rule in Listing 7.2.

```
EStopRoute(?r)
^ routesToMachine(?r, ?m)
^  routesFromEStop(?r, ?e)
-> spans(?e, ?m)
```

Listing 7.2: SWRL rule for e-stop span

This gives us the desired set of relationships. The inverse relation is populated automatically by the DL reasoner.

### Visibility Relation

The visibility relation is dramatically more difficult to implement. From a reasoning point of view, the fundamental problem is that OWL and SWRL *really do not like* non-monotonic reasoning. Why is this a problem? Because we can't use negations in SWRL rules. We would like to trace through two locations which are mutually-visible, provided that there is *not* a wall adjoining the two locations. However, that negation is precisely what we can't do. So instead, we turn to SHACL. Since the purpose of SHACL is to enforce data integrity, it is by design well-suited to closed-world reasoning. However, it is bad at existential quantification: we need to invalidate VisibilityZones which contain both sides of *the same* wall, and there isn't a good way to do that. To limit our reliance on SHACL, we will use a three-step process:

1. Use SWRL to assign an `isObstructedVisibilityZone` 'true' boolean predicate to any blocked visibility zone, as in Listing 7.3. In a correctly configured visibility model, SWRL will do nothing.

```
VisibilityZone(?z)
^ Wall(?w)
^ Location(?loc1)
^ Location(?loc2)
^ differentFrom(?loc1, ?loc2)
^ adjoinsLocation(?w, ?loc1)
^ adjoinsLocation(?w, ?loc2)
^ hasVisibleLocation(?z, ?loc1)
^ hasVisibleLocation(?z, ?loc2)
-> isObstructedVisibilityZone(?z, true)
```

Listing 7.3: Setting `isObstructedVisibilityZone` to 'true'

2. Use SHACL to invalidate any zone with `isObstructedVisibilityZone` set, as in Listing 7.4. Again, in a correctly configured case, SHACL will find nothing.

```
poc:NoObstructedVisibilityZones
  a sh:NodeShape ;
  sh:targetClass poc:VisibilityZone ;
  sh:property [
    sh:path poc:isObstructedVisibilityZone ;
    sh:maxCount 0 ;
    sh:message "A VisibilityZone is blocked by a Wall." ;
  ] .
```

Listing 7.4: Invalidate obstructed `VisibilityZone`

3. Use SWRL to infer the `isVisibleFrom` property for devices, using only the remaining unobstructed zones, as in Listing 7.5.

```
VisibilityZone(?z)
^ isInVisibilityZone(?locA, ?z)
^ isInVisibilityZone(?locB, ?z)
^ isLocatedIn(?deviceA, ?locA)
^ isLocatedIn(?deviceB, ?locB)
-> isVisibleFrom(?deviceA, ?deviceB)
```

Listing 7.5: Infer the `isVisibleFrom` property

The clear downside of this approach is the level of manual intervention required: visibility zones must be manually configured, and must be manually deleted if SHACL detects a wall conflict.

### 7.2.5 Query Design

At the highest level, the purpose of our automated reasoning application is verification of our safety requirements. They are:

- RQ 2.1.1: A red/yellow e-stop stops all spanned machines
- RQ 2.2.1: E-stops are red and yellow if and only if they are connected
- RQ 2.3.1: Releasing a red/yellow e-stop does not turn machines on
- RQ 3.1: Each machine has an e-stop
- RQ 3.2.1: Each connected control panel has a red/yellow e-stop
- RQ 3.3.1: Each red/yellow e-stop spans all visible machines
- RQ 4.1.1: Each machine and red-yellow e-stop is connected to the network.
- RQ 4.2: Registered devices do not disconnect

We remove the functional requirements 1. and 4.3 from consideration, as they are not safety-related and are not detailed enough to be directly represented in our system model. Even among those remaining, a great deal of complexity is visible. They use ideas like "spans" and "visible" which are not directly modeled in our knowledge base; however, they can be inferred as consequences of the knowledge that is modeled.

We begin by translating each of our requirements into a query for its violation, shown in Table 7.6.

| **Requirement** | **Violation query (natural language)** |
|---|---|
| RQ 2.1.1. A red/yellow e-stop stops all spanned machines | a red/yellow e-stop which is engaged, and a machine which it spans which is running |
| RQ 2.2.1. E-stop are red and yellow if and only if they are connected. | a red/yellow e-stop which is not connected, or a non red/yellow e-stop which is connected |
| RQ 2.3.1. Releasing a red/yellow e-stop does not turn machines on | a red/yellow e-stop which is not engaged, and a spanned machine which is in emergency-stop condition and is running |
| RQ 3.1. Each machine has an e-stop. | a machine which does not have an attached e-stop |
| RQ 3.2.1. Each connected control panel has a red/yellow e-stop. | a removable control panel which is connected and has an e-stop which is not red/yellow |
| RQ 3.3.1. Each red/yellow e-stop spans all visible machines. | a red/yellow e-stop, and a visible machine which it does not span |
| RQ 4.1.1. Each machine and red-yellow e-stop is connected to the network. | a device which is a machine or is a red/yellow e-stop which is not connected to the network |
| RQ 4.2. Registered devices do not disconnect. | a device which is registered and is not connected |

Table 7.6: Queries for requirement violations

Note that many of our queries involve closed-world style atomic negations. We can handle these in SPARQL using the MINUS operator, as in Listing 7.6, which selects e-stops which are red-and-yellow but *not* connected to the safety network, as well as e-stops which are not red-and-yellow but are connected to the safety network. Note that logical "and" is implicit in the `SELECT` command, but "and not" requires the `MINUS` command.

```
SELECT ?estop
WHERE {
  {
    {
      ?estop a poc:EStop .
      ?estop poc:isRedYellow true .
    }
    MINUS
    {
      ?estop a poc:EStop .
      ?estop poc:isConnectedTo ?safetyNetwork .
    }
  }
  UNION
  {
    ?estop a poc:EStop .
    ?estop poc:isRedYellow false .
    ?estop poc:isConnectedTo ?safetyNetwork .
  }
}
```

Listing 7.6: Example SPARQL query

We formalize each requirement in this way as a SPARQL query for its violations. With this, we have essentially completed our technical process: it remains to organize this process so that we get to our verification result in an efficient way. While there is no really nice way to build human-readable output into the SPARQL query itself, it is fairly straightforward to provide a description of the requirement and of the violation as comments. Then we can pull those comments out of the query file as needed.

### 7.2.6 Instantiation

With our "T-box" classes and properties defined, we are ready to instantiate our base scenario. First we manually instantiate all of the entities in our base scenario. In our limited scenario, this is doable. In a more realistic application, some degree of automation and inference would be necessary. Next, we need to manually introduce all of our object properties. Fortunately for us, some are immediately inferred: in the example below, `eStopRoutingTable1` is already identified to have property `isEStopRoutingTableOf` `safetyNet1`, because we manually assigned `safetyNet1` `hasEStopRoutingTable` `eStopRoutingTable1`. In Protégé, inferred relations are always shown with a yellow background, as in Figure 7.10.

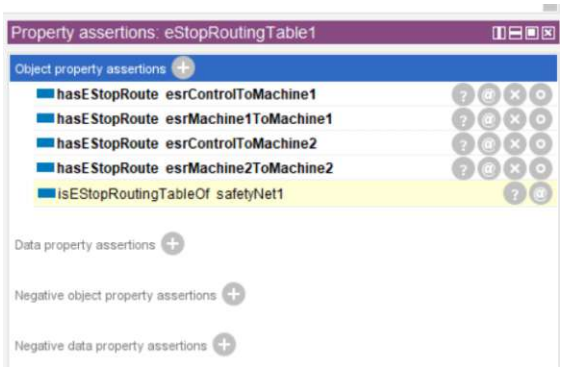


Figure 7.10: Inferred relations are yellow in Protégé.

In this kind of tedious data entry, the reasoning-based checking powers of a knowledge base become apparent. For example, when configuring `eStopRouteMachine1ToMachine1`, we accidentally set the route from `Machine1`, rather than from the e-stop connected to it. We can see in Figure 7.11 that the ontology immediately caught the error.


Figure 7.11: Protégé identifies an incorrect class relation.

It is worth pausing to explain how this error arises, because it tells us something about how typing works under the OWL's open world assumption. The `routesFromEStop` property has domain `EStopRoute` and range `EStop`. Thus, setting this object property, Protégé concludes that object `machine1` must have type `EStop`. However, `machine1` has explicit type `Machine`. At this point there is no problem. The problem arises because we have already specified, in our class structure, that `EStop` and `Machine` are disjoint classes: hence `machine1` both is and is not a member of `Machine`. The mistake is easy to fix.

In this spirit, we will also specify that all of our individuals are different from each other. Since OWL lacks the unique name assumption, this is something which we must do manually. Protégé has a one-click shortcut to do so. Our final step is to set the data properties to represent our default states. For our basic model, our machines are all running, our e-stops are all disengaged, and our disconnected color-changing e-stop is gray.

With this approach, we produce an instantiated OWL knowledge model for our scenario. We copy the model and make changes in order to produce a separate RDF file for each of our test cases.

### 7.2.7 Execution and output

We use a bash script to automate our Jena command line calls. The output for each test case shows us the expected result. For unsafe test cases, a readable message explains the failure. A sample of the output is shown in Figure 7.12.

In all our test cases, our script correctly detects the violated requirements and reports them in a human-readable way.

The reader may access the code in a public GitHub repository[i] to try this out personally. Our repo provides:

- `rdf/test_case[0-n].rdf` - test case models with exported inferred axioms. Test case 0 is the base case.
- `shacl/shacl_shapes.ttl` - our SHACL shapes for model validation in all cases.
- `sparql/safety_requirement[n-n-n].rq` - our SPARQL queries for our safety requirements, for batch processing in Jena ARQ.
- `check.sh` - our bash script which automatically checks everything.

The reader will need a Linux system with Apache Jena and its Java runtime requirements.

## 7.3 Summary

With these steps complete, we now have a knowledge-based system model, a compatible expression of our requirements, and an automated reasoning tool which we can use to verify system correctness. We have an unambiguous specification of each test case. Behind this, we have a richly documented design process which allows us to see that our knowledge base really does represent the scenario which we originally described in language and pictures. Applying our reasoner to our test cases, we are able to successfully detect violations of safety requirements in a range of test cases.

At this point, we have produced a proof-of-concept solution to our technical problem of automatically verifying safety requirements in a reconfigurable system. In the next chapter, we will evaluate the quality of our solution.

[i] `https://github.com/donaldacker/safety-verification`

| Class | Properties | Inverse | Functional? | Inverse-functional?' |
|---|---|---|---|---|
| Safety Network | hasRegistrationArray | isRegistrationArrayOf | Yes | Yes |
| | hasConnectionArray | isConnectionArrayOf | Yes | Yes |
| | hasEStopRoutingTable | isEStopRoutingTableOf | Yes | Yes |
| EStopRoutingTable | hasEStopRoute | isEStopRouteOf | No | Yes |
| | isEStopRoutingTableOf | hasEStopRoutingTable | Yes | Yes |
| EStopRoute | isEStopRouteOf | hasEStopRoute | Yes | No |
| | routesFromEStop | | Yes | No |
| | routesToMachine | | Yes | No |
| RegistrationArray | isRegistrationArrayOf | hasRegistrationArray | Yes | Yes |
| | hasRegisteredDevice | isRegisteredBy | No | Yes |
| ConnectionArray | isConnectionArrayOf | hasConnectionArray | Yes | Yes |
| | hasConnectedDevice | isConnectedTo | No | Yes |
| Device | isRegisteredBy | hasRegisteredDevice | Yes | No |
| | isConnectedTo | hasConnectedDevice | Yes | No |
| | isLocatedIn | containsDevice | Yes | No |
| Machine | hasEStop | isEStopForMachine | Yes | Yes |
| EStop | isEStopForMachine | hasEStop | Yes | Yes |
| ColorChangingEStop | isColorChangingEStopForRemovableControlPanel | hasColorChangingEStop | Yes | Yes |

Table 7.3: Properties of component relations, part 1

| Class | Properties | Inverse | Functional? | Inverse-functional?' |
|---|---|---|---|---|
| Removable-ControlPanel | hasColor-Changin-gEStop | isColor-Changin-gEStopForRe-movableCon-trolPanel | Yes | Yes |
| Location | containsDe-vice | isLocatedIn | No | Yes |
| | adjoinsWall | adjoinsLoca-tion | No | No |
| | isInVisibility-Zone | hasVisibleLo-cation | No | No |
| Wall | adjoinsLoca-tion | adjoinsWall | No | No |
| VisibilityZone | hasVisibleLo-cation | isInVisibility-Zone | No | No |

Table 7.4: Properties of component relations, part 2

```
Checking test case ./rdf/test_case_3.rdf for compliance with safety requirements:
# Requirement 2.1.1: A red/yellow e-stop stops all spanned machines
# Requirement met.

# Requirement 2.2.1: E-stops are red and yellow if and only if they are connected
# Requirement met.

# Requirement 2.3.1: Releasing a red/yellow e-stop does not turn a spanned machine on
# Requirement met.

# Requirement 3.1: Each machine has an e-stop
# Violation: a machine which does not have an attachedEStop.
----------------
| machine      |
================
| poc:machine3 |
----------------

# Requirement 3.2.1: Each connected control panel has a red/yellow e-stop
# Requirement met.

# Requirement 3.3.1: Each red/yellow e-stop spans all visible machines
# Violation: A r/y e-stop and a visible machine which it does not span.
---------------------------------------
| estop                | machine      |
=======================================
| poc:eStopMachine1    | poc:machine3 |
| poc:eStopControlRoom | poc:machine3 |
---------------------------------------

# Requirement 4.1.1: Each machine and red-yellow e-stop is connected to the network.
# Violation: A device which is a machine or a r/y e-stop which is not connected to the network.
----------------
| device       |
================
| poc:machine3 |
----------------

# Requirement 4.2: Registered devices do not disconnect
# Requirement met.

3 requirement(s) were not met for test case ./rdf/test_case_3.rdf

3 test case(s) failed to pass all requirements.
```

Figure 7.12: Command line output from Jena SPARQL, via our Bash script

# CHAPTER 8

# Evaluation

Our response to our research question is a verification process, itself consisting of two processes: A process for modeling an industrial system, including its safety requirements, and a process for automatically verifying the compliance of a system (model) with its requirements. Using the criteria for a successful design artifact which we presented in the introduction, we will now evaluate the two parts of our process. For each criterion, we score the artifact on a verbal rating scale of "very bad", "bad", "neutral", "good", and "very good".

## 8.1 Evaluation of Modeling Process

Our system modeling process can be summarized as:

1. Translate narrative descriptions and external standards into a SysML model, including Block Definition diagrams, State Machine diagrams, and Requirement diagrams.
2. Translate the Block Definition diagrams into classes in an OWL knowledge base.
3. Translate the State Machine diagrams into data properties in the OWL knowledge base.
4. Translate the structural features of the Block Definition diagrams into SHACL shapes.
5. Identify the concepts used in the Requirement diagram and create SWRL rules to infer necessary facts.
6. Translate the safety requirements into SPARQL queries. Include human-readable descriptions in the comments of the SPARQL files.

The product of this process is an OWL knowledge base in RDF format, a SHACL shapes file, and a series of SPARQL query files (one query per file).

We check our modeling process against the success criteria from Chapter 1.

1. *Allows the modeler to accurately reflect the target system.*
   Very good: Because we chose to create our own ontology, we have a very strong ability to reflect whatever we see in the target system.

2. *Unites physical and configuration elements as well as requirements.*
   Good: Our SysML model handles requirements explicitly; our knowledge model does not include requirements but allows them to be modeled separately as SPARQL queries.

3. *Allows for iterative change as the target system or the modeler's understanding of it changes.*
   Good: The SysML model makes tracking iterative changes easier. However, the modeling process still depends on manually updating values, which is labor-intensive.

4. *Makes use of existing technologies in order to reduce the modeler's effort.*
   Good: We use existing modeling languages. OWL provides really useful tools such as DL reasoning on classes and SWRL rule generation, meaning the modeler does not have to manually input every single element of the model. However, because we are creating our own ontology, we are not benefiting from much reuse.

5. *Makes use of existing modeling languages in order for different modelers' separate work to be mutually intelligible and functionally compatible.*
   Neutral: While we gain mutual intelligibility from the common languages of SysML and OWL, we do not have the benefit of a widely-established reference ontology to work from.

| **Criterion (Modeling)** | **Evaluation** |
|---|---|
| Allows the modeler to accurately reflect the target system. | Very good |
| Unites physical and configuration elements as well as requirements. | Good |
| Allows for iterative change as the target system or the modeler's understanding of it changes. | Good |
| Makes use of existing technologies in order to reduce the modeler's effort. | Good |
| Makes use of existing modeling languages in order for different modelers' separate work to be mutually intelligible and functionally compatible. | Neutral |
| *Overall* | ***Good*** |

Table 8.1: Evaluation of system modeling criteria

## 8.2 Evaluation of Verification Process

Our formal verification process can be summarized as:

1. Use Protégé to check the consistency of the RDF knowledge base using the Pellet reasoner.

2. Use Protégé to export the knowledge base including inferred axioms.

3. Validate the knowledge base using the SHACL files.

4. Run each SPARQL query, capturing failures and explanatory notes.

For this process, our input is an RDF file, and our output is a report in text as shown above. We have wrapped this in a bash script which applies this process to each RDF file in a folder, allowing us to check all of our test cases conveniently.

We check our verification process against the success criteria from Chapter 1:

1. *Uses automated reasoning tools to ensure correct results.*
   Very good: We are satisfied that the well-established tools we use perform correctly, and we see they give unambiguous results.

2. *Detects common human errors from the modeling stage.*
   Good: We were successfully able to use OWL class reasoning and, in particular, OWL 2 disjoint classes, as well as SHACL shapes, to validate our model. However, human errors are still possible: in particular, configuring visibility is tricky, and while we can invalidate incorrect wall configurations, we cannot ensure that every necessary visibility zone is configured.

3. *Produces a clear answer, either affirming the system's consistency or specifically identifying each fault.*
   Very good: our reasoning toolchain performs exactly as desired.

4. *Provides human-readable explanations for each fault.*
   Very good: while our SPARQL-comments-plus-grep workaround is a bit awkward, its output is highly readable. For larger-scale work, it would not be at all difficult to aggregate our output into more structured reports.

5. *Can be conveniently applied to multiple test cases or to entirely different system models.*
   Very good: our simple bash script makes running multiple tests very easy.

| Criterion (Reasoning) | Evaluation |
|---|---|
| Uses automated reasoning tools to ensure correct results. | Very good |
| Detects common human errors from the modeling stage. | Good |
| Produces a clear answer, either affirming the system's consistency or specifically identifying each fault. | Very good |
| Provides human-readable explanations for each fault | Very good |
| Can be conveniently applied to multiple test cases or to entirely different system models. | Very good |
| *Overall* | ***Very good*** |

Table 8.2: Evaluation of automated reasoning criteria

## 8.3 Results

Based on the criteria, we can say that our modeling process is **good** and our verification process is **very good**. We see that the shortcomings come from two major areas:

- Dependence on labor-intensive manual data entry
- Lack of an already-established ontology

The first is essentially unavoidable, though perhaps it would be possible to further automate reasoning on the class relationships to allow for *less* data entry. The second is a clear drawback, and relates to the first, as an established ontology would have the potential to greatly reduce manual entry. However, at this time, we do not believe that a sufficiently mature option exists.

Overall, we can consider our design process successful. Next, in our final chapter, we reflect on these results and consider directions for further research.

CHAPTER 9

# Conclusion

## 9.1 Summary

We have come a long way in our exploration of our research question, "How can we automatically verify that an industrial system is consistent with all of its safety requirements?" A quick review will remind us of our work and help make our conclusions make sense.

Seeing that our research question demanded that we first model our target system and then perform verification of the model, we recognized that our artifact would be a process divided into two major parts: a modeling process, and a verification process. Viewing this pair of processes as our artifact, we created two sets of success criteria: one for the modeling process, and one for the verification process.

With that in mind, we briefly summarized the state of the art on four key topics: functional safety, model-based system engineering, knowledge engineering, and automated reasoning. We then examined the current research on functional safety automation in these areas. We paid particular attention to Etz's Flexible Safety Systems project, to which this thesis is one of eight related thesis projects.

After the literature review, we began our own project by defining the industrial scenario with which we would work: in our case, a pair of production rooms with robots, separated by a removable wall. We then made a detailed examination of the technologies used for our two main tasks: system modeling and automatic verification. In each of these chapters, we ended by choosing the specific technologies we would use to create our process. With everything decided, we presented our design artifact: a detailed step-by-step process.

We then began our actual technical work, creating a proof of concept in which we applied our process to our example scenario. With positive results here, we evaluated the two halves of our process according to the criteria we devised in Chapter 1, finding that our

modeling process was "good" and our automated reasoning process "very good". These basic qualitative results now bring us here: what can we conclude?

First, we will review the identified short-comings of our solution, and hence identify open questions remaining on the topic. We will then recast these as potential directions for future work.

## 9.2 Open Questions

In our evaluations, we identified two overarching challenges: dependence on manual data entry, and the lack of a well-developed standard ontology. In addition to these, we can look back on our process and ask what steps introduced particular friction. We immediately see that modeling visibility clashes with open-world-assumption reasoning in a serious way: checking that there *isn't* a wall presents a really awkward challenge, which we can only solve with further dependence on correct data entry. State-based requirements also required an extra layer of complexity. But, even so, while we explicitly modeled the states, we do not have any explicit modeling of the transitions themselves: with more complex state-based requirements, this could be a major problem. Finally, we can review the high-level challenges and ask: is this practical? Does it scale? And, thinking back to Sternudd, is it even *possible* for safety requirements to be made unambiguous? With all of this is mind, we identify the following open questions:

1. Can we reduce or automate manual data entry?
2. What would have to happen for an ontology to reach de facto standard status?
3. How can spatial relationships and visibility be modeled in an ontology?
4. How can state machines be fully represented in an ontology?
5. Does our process scale to larger numbers of system components? To larger numbers of requirements?
6. How, ultimately, should safety requirements be represented?

## 9.3 Further Work

These open questions point the way to several directions for future work. We introduce each direction with reference to the open questions above.

### 9.3.1 Ontology-Based Digital Twin

Questions 1, 3, and 4 amount to: "how do we turn our industrial system into a knowledge-based representation, without excessive manual labor?" We might imagine a digital twin which both encompasses the full range of entities and relations in the knowledge model,

and allows an engineer to visually explore the model to ensure correctness of entered data. This would be almost a video game version of a factory, with a robust level editor.

### 9.3.2 Ontology-Based Safety Requirements

Question 6 is presented as a question, but is really quite loaded if we're honest: the advantages of representing safety requirements in a machine-readable way are obvious in the context of automated reasoning and, hence, of reconfigurable systems. Developing an ontology for safety requirements would make it possible for future safety requirements to be precisely articulated, with a single source of truth grounded in a shared ontology. This then essentially becomes the answer to Question 2: if safety standards are expressed in a certain standardized ontology, this will become the backbone of a broader standard for representing the industrial systems which must comply with them.

### 9.3.3 Real-Life Trial

Question 5 remains: does this approach to modeling and formal verification scale? The obvious way to find out would be to choose a real-world working industrial system to model and verify. This would answer the most important question of scale: does the process scale up enough to be useful for something real?

## 9.4 Remarks

The implications of even these three directions of work are, upon reflection, enormous. The history of human efforts to force the natural and human world to conform to "logic" are fraught. Ambiguity and uncertainty are, after all, inescapable parts of the human experience. On the other hand, technical systems can be made to operate with an impressive degree of precision and well-defined behavior. But the border between the technical and the human lies at the operator's workstation, at the steering wheel, at the edge of the conveyor belt. This is a space where the potential for rigor and the necessity of uncertainty are unclear: a boundary of ambiguous ambiguity. In the past, scientific efforts have focused on forcing the human to conform to the demands of the machine: from Frederick Taylor's time-motion studies to today's workplace safety trainings. The value the machine produces is recognized, while the values of the humans who use it are, in the best circumstances, a demand to be compromised with.

Seen in this context, functional safety itself amounts to a quiet revolution: the needs of the humans are first expressed, and then the technical system is designed. Moreover, with an ontological approach, the technical system would have to "speak the same language" as the human requirements. This reversal, if fully realized, would be a profound change in relations between human and industry. I can only hope to see it happen!

# Appendix: Overview of Generative AI Tools Used

I have used the Google Gemini 2.0 Flash large language model chatbot to help quickly produce bibtex markup for references [39]. The generated code appears in the .bib file, not in this paper. Typical examples of prompts:

```
Please provide a bibtex reference for this webpage:
https://www.w3.org/RDF/Metalog/docs/sw-easy
The Semantic Web Made Easy
W3C
```

and

```
Please provide a bibtex reference for yourself - that is, for
the Gemini 2.0 Flash tool
```

In each case, I have reviewed and often changed the Gemini output. The most common changes were entering access dates for webpages and fixing random syntax errors such as unclosed brackets.

I also used Gemini for an initial translation of my abstract into German, with the following prompt, followed by my original English text:

```
Please translate the following abstract into German.  Please use
a register appropriate for academic writing.  And please
translate words in a way that makes sense in a systems
engineering context:
```

I then reviewed the output myself and had my wife and my advisor (both native speakers) review it as well.

# List of Acronyms

**MBSE** Model-Based System Engineering

**OWL** the Web Ontology Language

**SHACL** the Shapes Constraint Language

**SPARQL** the SPARQL Protocol and RDF Query Language

**RDF** the Resource Description Framework

**RDFS** RDF Schema

**SWRL** the Semantic Web Rule Language

**SQWRL** the Semantic Query-Enhanced Web Rule Language

**DL** Description Logics

**SysML** the Systems Modeling Language

**SQL** Structured Query Language

**FOAF** Friend of a Friend

**MD** the Machinery Directive

**UML** the Unified Modeling Language

**W3C** the World Wide Web Consortium

**FOL** First-Order Logic

# List of Figures

# List of Tables

# Bibliography


[1] AutomationML. https://opcfoundation.org/markets-collaboration/automation-ml/. Accessed: 2023-11-13.

[2] Commission Implementing Decision (EU) 2023/1586 of 26 July 2023 on harmonised standards for machinery drafted in support of Directive 2006/42/EC of the European Parliament and of the Council. http://data.europa.eu/eli/dec_impl/2023/1586/oj. Accessed: 2023-11-13.

[3] Directive 2006/42/EC - machinery directive. https://osha.europa.eu/en/legislation/directives/directive-2006-42-ec-of-the-european-parliament-and-of-the-council. Accessed: 2023-11-13.

[4] MBSE Knowledge Graph. https://projekte.ffg.at/projekt/4154553. Accessed: 2023-11-13.

[5] MBSE Wiki. https://www.omgwiki.org/MBSE/doku.php. Accessed: 2023-11-13.

[6] Model Based Systems Engineering (MBSE) Workshop at INCOSE IW 2023. https://www.omgwiki.org/MBSE/doku.php?id=mbse:incose_mbse_iw_2023. Accessed: 2023-11-13.

[7] Safety and functional safety FAQ - IEC 61508 explained. https://www.iec.ch/functional-safety/faq#1542. Accessed: 2023-11-13.

[8] SysML FAQ: What is the relation between SysML and MBSE? https://sysml.org/sysml-faq/what-is-relation-between-sysml-and-mbse.html. Accessed: 2023-11-13.

[9] System Engineering Models meet Knowledge Graphs. https://nebula.esa.int/content/system-engineering-models-meet-knowledge-graphs. Accessed: 2023-11-13.

[10] What is Industrie 4.0? https://www.plattform-i40.de/IP/Navigation/EN/Industrie40/WhatIsIndustrie40/what-is-industrie40.html. Accessed: 2023-11-11.

[11] Functional safety essential to overall safety. International Electrotechnical Commission, Geneva, Switzerland, 2015.

[12] Safety of machinery — emergency stop function — principles for design, 2023.

[13] F.H. Abanda, J.H.M. Tah, and R. Keivani. Trends in built environment semantic web applications: Where are we today? Expert Systems with Applications, 40(14):5563–5577, 2013.

[14] Sunitha Abburu. A survey on ontology reasoners and comparison. International Journal of Computer Applications, 57(17):33, November 2012.

[15] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995.

[16] Ananda Adhikari, Aspak Saban, Shirish Bohora, and Viranchi Shastri. Functional safety for automatic emergency braking based on iso 26262: Paper no.: 2023-gi-04. ARAI Journal of Mobility Technology, 3(3):666–685, 2023.

[17] Emhimed Alatrish. Comparison of ontology-editors for semantic web. Management Information Systems, 8(2):018–024, 2013. Received 03 April 2012; Accepted 24 April 2013; UDC 004.4.

[18] Nazakat Ali, Kristina Lundqvist, and Kaj Hänninen. Mitigation ontology for analysis of safety-critical systems. In Magryta-Mut Kolowrocki, editor, Advances in Reliability, Safety and Security, Part 2, 2024.

[19] Renzo Angles and Claudio Gutierrez. The expressive power of sparql. In Amit Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan, editors, The Semantic Web - ISWC 2008, pages 114–129, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[20] Sunil Basnet, Ahmad Bahootoroody, Meriam Chaal, Osiris A. Valdez Banda, Janne Lahtinen, and Pentti Kujala. A decision-making framework for selecting an mbse language–a case study to ship pilotage. Expert Systems with Applications, 193:116451, 2022.

[21] Stephan Baumgart, Joakim Fröberg, and Sasikumar Punnekkat. Enhancing model-based engineering of product lines by adding functional safety. 09 2015.

[22] Mohamad Bdiwi, Marko Pfeifer, and Andreas Sterzing. A new strategy for ensuring human safety during various levels of interaction with industrial robots. CIRP Annals, 66(1):453–456, 2017.

[23] Alessandro Bernardini, Wolfgang Ecker, and Ulf Schlichtmann. Where formal verification can help in functional safety analysis. In 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–8, 2016.

[24] N. Bridwell. Clifford the Big Red Dog. Number bk. 1 in Be Big! Be a good friend. Scholastic, 2010.

[25] Jan vom Brocke, Alan Hevner, and Alexander Maedche. Introduction to Design Science Research, pages 1–13. 09 2020.

[26] M. Catelani, L. Ciani, and V. Luongo. A simplified procedure for the analysis of safety instrumented systems in the process industry application. Microelectronics Reliability, 51(9):1503–1507, 2011. Proceedings of the 22th European Symposium on the RELIABILITY OF ELECTRON DEVICES, FAILURE PHYSICS AND ANALYSIS.

[27] Maarten Dalmijn. Wtf is a requirement? the deceptive nature of requirements, September 2024.

[28] Pierre De Saqui-Sannes, Rob A. Vingerhoeds, Christophe Garion, and Xavier Thirioux. A taxonomy of mbse approaches by languages, tools and methods. IEEE Access, 10:120936–120950, 2022.

[29] Michael DeBellis. A practical guide to building owl ontologies using protégé 5.5 and plugins, October 2021.

[30] D. Etz. Flexible Safety Systems: Use Cases, Requirements, System Design, and Software Architecture. Dissertation, Technische Universität Wien, 2024.

[31] D. Etz, P. Denzler, T. Frühwirth, and W. Kastner. Functional Safety Use Cases in the Context of Reconfigurable Manufacturing Systems. In 27th International Conference on Emerging Technologies and Factory Automation (ETFA), pages 1–8, 2022.

[32] Dieter Etz, Patrick Denzler, Thomas Fruhwirth, and Wolfgang Kastner. Functional Safety Use Cases in the Context of Reconfigurable Manufacturing Systems. In 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), pages 1–8, 2022.

[33] Federal Aviation Administration. Summary of the faa's review of the boeing 737 max. Technical report, Federal Aviation Administration, November 2020.

[34] Christof Fetzer, Christoph Weidenbach, and Patrick Wischnewski. Compliance, functional safety and fault detection by formal methods. In Tiziana Margaria and Bernhard Steffen, editors, Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications, pages 626–632, Cham, 2016. Springer International Publishing.

[35] FOAF Project. Foaf vocabulary specification. Accessed: 2025-02-23.

[36] Gerald Fudge and Emily Reeves. A model-based reverse system engineering methodology for analyzing complex biological systems with a case study in glycolysis. IEEE Open Journal of Systems Engineering, PP:1–15, 01 2024.

[37] Jan Gačnik, Henning Jost, Frank Köster, and Martin Fränzle. The descas methodology and lessons learned on applying formal reasoning to safety domain knowledge. In Eckehard Schnieder and Geza Tarnai, editors, FORMS/FORMAT 2010, pages 207–215, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[38] Antonello De Galizia. Development of a knowledge base kb3 dedicated to the modeling of electrical systems for the realization of dynamic probabilistic safety assessment : application to the uk epr. Master's thesis, Politecnico di Milano, Italy, 10 2013.

[39] Google AI. Gemini 2.0 flash, 2024. Large Language Model, accessed: 2025-02-23.

[40] Henson Graves. Integrating sysml and owl. In Proceedings of [Conference Name - if known, add it here, otherwise omit], Fort Worth, Texas, USA, 2015. Lockheed Martin Aeronautics Company. henson.graves@lmco.com.

[41] Tom Gruber. Ontology. Springer-Verlag, 2009.

[42] Myron Hecht and Jaron Chen. Verification and validation of sysml models. INCOSE International Symposium, 31:599–613, 07 2021.

[43] Thomas Hofweber. Logic and Ontology. In Edward N. Zalta and Uri Nodelman, editors, The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[44] Erik Hollnagel. Is safety a subject for science? Safety Science, 67:21–24, 2014. The Foundations of Safety Science.

[45] J. Holt, S. Perry, Institution of Engineering, and Technology. SysML for Systems Engineering. Computing and Networks Series. Institution of Engineering and Technology, 2008.

[46] Savannah Hutchins, Niral Jhaveri, and Vincent G. Duffy. Integrating industry 4.0 technologies for enhanced safety engineering: A comprehensive review and analysis. In Vincent G. Duffy, Heidi Krömker, Norbert A. Streitz, and Shin'ichi Konomi, editors, HCI International 2023 – Late Breaking Papers, pages 43–58, Cham, 2023. Springer Nature Switzerland.

[47] Ivo Häring, Vivek Sudheendran, Roman Sankin, and Stefan Hiermaier. Joint functional safety ISO 26262 and cybersecurity STRIDE/HEAVENS assessment by developers within MBSE SPES framework using extended SysML diagrams and minor automations. https://www.iapsam.org/PSAM16/slides/IV42-SLIDES-PSAM16.pdf.

[48] INCOSE UK Ltd. Mbse languages. Model Based Systems Engineering Wiki, 1 2021. Accessed: 2025-03-23. Last modified on 28 January 2021, at 13:24.

[49] Muhammad Atif Javed, Faiz Ul Muram, Hans Hansson, Sasikumar Punnekkat, and Henrik Thane. Towards dynamic safety assurance for Industry 4.0. Journal of Systems Architecture, 114:101914, 2021.

[50] Lu Jinzhi. A Knowledge Management Approach Supporting Model-Based Systems Engineering. 03 2021.

[51] Marcio Lazai Junior, Álvaro dos Santos Justus, Eduardo de Freitas Rocha Loures, Eduardo Alves Portela Santos, and Anderson Luis Szejka. Industry 4.0 technologies for improving functional machinery safety management from the interoperability point of view in the automotive industry. In Antônio Márcio Tavares Thomé, Rafael Garcia Barbastefano, Luiz Felipe Scavarda, João Carlos Gonçalves dos Reis, and Marlene Paula Castro Amorim, editors, Industrial Engineering and Operations Management, pages 475–487, Cham, 2021. Springer International Publishing.

[52] Henning Kagermann and Wolfgang Wahlster. Ten years of industrie 4.0. Sci, 4(3):26, 2022.

[53] Hanumanthrao Kannan. Formal reasoning of knowledge in systems engineering through epistemic modal logic. Systems Engineering: The Journal of The International Council on Systems Engineering, 24(1):3–16, 2020.

[54] Andreas M. Kleinhans. Knowledge-Based Modelling. In Peter M. Milling and Erich O. K. Zahn, editors, Computer-Based Management of Complex Systems, pages 527–534, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[55] Markus Krötzsch. Knowledge graphs lecture 8: Expressive power and complexity of sparql, December 2018. Knowledge-Based Systems, TU Dresden, 4th Dec 2018.

[56] John Lee, Ian Cameron, and Maureen Hassall. Improving process safety: What roles for Digitalization and Industry 4.0? Process Safety and Environmental Protection, 132:325–339, 2019.

[57] Nancy Leveson. A Paradigm Change for Safety and for System Engineering. https://www.omgwiki.org/MBSE/lib/exe/fetch.php?media=mbse:incose_mbse_iw_2023:0.0.2023-01-28.iw2023_mbse_workshop_plenary_leveson_incose_keynote_talk.pdf.

[58] Chokri Mraidha Luis Palacios Medinacelli, Florian Noyrit. Augmenting model-based systems engineering with knowledge. In MODELS '22: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2022.

[59] Tom Meany. Functional safety and industrie 4.0. pages 1–7, 06 2017.

[60] Faïda Mhenni, Nga Nguyen, Hubert Kadima, and Jean-Yves Choley. Safety analysis integration in a SysML-based complex system design process. In 2013 IEEE International Systems Conference (SysCon), pages 70–75, 2013.

[61] Natalie Mouyal. Updated IEC Standard ensures the functional safety of machinery. https://etech.iec.ch/issue/2021-02/updated-iec-standard-ensures-the-functional-safety-of-machinery. Accessed: 2023-11-13.

[62] Mark A. Musen. The protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015.

[63] Martin O'Connor. The semantic web rule language, 2009. Stanford Center for Biomedical Informatics Research, Stanford University.

[64] Aleksandra Polak-Sopinska, Zbigniew Wisniewski, Anna Walaszczyk, Anna Maczewska, and Piotr Sopinski. Impact of industry 4.0 on occupational health and safety. In Waldemar Karwowski, Stefan Trzcielinski, and Beata Mrugalska, editors, Advances in Manufacturing, Production Management and Process Control, pages 40–52, Cham, 2020. Springer International Publishing.

[65] Frederic Portoraro. Automated Reasoning. In Edward N. Zalta and Uri Nodelman, editors, The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Summer 2023 edition, 2023.

[66] D Reitgruber. Knowledge base for reconfigurable safety systems. Master's thesis, Technische Universität Wien, 2022.

[67] Rhedogian. Change my view: Model based systems engineering in 2024 is at best overhyped, or is at worst actively dying, 2024. Reddit post, accessed: 2025-02-23.

[68] Catherine Roussey, François Pinet, Myoung-Ah Kang, and Oscar Corcho. An Introduction to Ontologies and Ontology Engineering, volume 1, pages 9–38. 07 2011.

[69] Alejandro Salado. Model-Based Requirements, pages 1–29. Springer International Publishing, Cham, 2020.

[70] Nataliya Shevchenko. An introduction to model-based systems engineering (mbse). Carnegie Mellon University, Software Engineering Institute's Insights (blog), Dec 2020. Accessed: 2025-Feb-21.

[71] SIGMATEK. Safety hot swap: Change machine network during active operation, 12 2019. Accessed: 2025-02-23.

[72] Jamie Smith. Automated reasoning for sysml v2 part 2. Medium, 2024. Accessed: 2025-02-23.

[73] Patrik Sternudd. Unambiguous requirements in functional safety and iso 26262: Dream or reality? 2011.

[74] J. A. Torrecilla-García, M. C. Pardo-Ferreira, M. Martínez-Rojas, and J. C. Rubio-Romero. A conceptual approximation toward occupational safety and health within the servitized industry 4.0. In David De la Fuente, Raúl Pino, Borja Ponte, and Rafael Rosillo, editors, Organizational Engineering in Industry 4.0, pages 37–48, Cham, 2021. Springer International Publishing.

[75] Anna Townshend. Standards guide the use of e-stops. Control Design, 1 2023.

[76] Leo van Ruijven. Ontology for Systems Engineering as a base for MBSE. INCOSE International Symposium, 25:250–265, 10 2015.

[77] Federico Vicentini, Mehrnoosh Askarpour, Matteo G. Rossi, and Dino Mandrioli. Safety assessment of collaborative robotics through automated formal verification. IEEE Transactions on Robotics, 36(1):42–61, 2020.

[78] W3C. The semantic web made easy, 2003. Accessed: [Insert Date Accessed].

[79] W3C OWL Working Group. Owl 2 web ontology language: Direct semantics (second edition). Technical report, World Wide Web Consortium (W3C), 2012. Accessed: 2025-02-23.

[80] W3C OWL Working Group. Owl 2 web ontology language document overview (second edition). W3C Recommendation REC-owl2-ref-20121211, World Wide Web Consortium, December 2012.

[81] W3C OWL Working Group. Owl 2 web ontology language mapping to rdf graphs (second edition). Technical report, World Wide Web Consortium (W3C), 12 2012. Accessed: 2025-02-23.

[82] W3C OWL Working Group. Owl 2 web ontology language: Overview (second edition). Technical report, World Wide Web Consortium (W3C), 2012. Accessed: 2025-02-23.

[83] W3C OWL Working Group. Owl 2 web ontology language profiles (second edition). W3C Recommendation REC-owl2-profiles-20121211, World Wide Web Consortium, December 2012.

[84] W3C Working Group. Rdf schema 1.1. Technical report, World Wide Web Consortium (W3C), 2014.

[85] W3C Working Group. Rdf 1.2 concepts and abstract syntax. Technical report, World Wide Web Consortium (W3C), 2024.

[86] Helna Wardhana, Ahmad Ashari, and Anny Kartika Sari. Transformation of sysml requirement diagram into owl ontologies. International Journal of Advanced Computer Science and Applications, 11(4), 2020.

[87] Main Commission Aircraft Accident Investigation Warsaw. Report on the Accident to Airbus A320-211 Aircraft in Warsaw. http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html, 1994.