Quantitative Resilience Assessment of Critical Infrastructures using High-Performance Simulations

by Oleksandr Netkachov.

The thesis is submitted for the degree of Doctor of Philosophy
to City, University of London.

The research was conducted at City, University of London,
School of Science & Technology,
Department of Computer Science.

July 2023

Contents

Acknowledgements

I would like to offer my appreciation and thanks to my supervisor, Dr Peter Popov, for his support during this project, and to my second supervisor, Prof Robin Bloomfield, for his inspirational and practical suggestions. I am grateful for the opportunity to explore this exciting area of research at the Centre for Software Reliability at City, University of London.

I would also like to thank Dr Kizito Salako for his expert guidance, support, and advice during our working group meetings.

I, Oleksandr Netkachov, grant powers of discretion to the Director of Library Services to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to the standard acknowledgement conditions.

CITY
UNIVERSITY OF LONDON
EST 1894

STATEMENT OF CO-AUTHORS of JOINT PUBLICATIONS

TO WHOM IT MAY CONCERN

Publications:

Netkachov, Oleksandr, Peter Popov, and Kizito Salako. 2014. "**Quantification of the impact of cyber attack in critical infrastructures.**" In Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8696 LNCS:316–27. Springer.

Netkachov, Oleksandr, Peter Popov, and Kizito Salako. 2016. "**Model-Based Evaluation of the Resilience of Critical Infrastructures under Cyber Attacks.**" In Critical Information Infrastructures Security, 231–43. Lecture Notes in Computer Science. Cham: Springer International Publishing.

Netkachov, Oleksandr, Peter Popov, and Kizito Salako. 2019. "**Quantitative Evaluation of the Efficacy of Defence-in-Depth in Critical Infrastructures.**" In Resilience of Cyber-Physical Systems, edited by Francesco Flammini, 89–121. Springer.

Name of candidate: Oleksandr Netkachov

Title of research thesis: Quantitative Resilience Assessment of Critical Infrastructures using High-Performance Simulations

Name of first supervisor: Dr. Peter Popov

We, the undersigned, co-authors of the above publications, confirm that the above publications have not been submitted as evidence for which a degree or other qualification has already been awarded.

We, the undersigned, further indicate the candidate's contribution to the publications in our joint statement below.

Signature: [redacted]
Name: Kizito Salako
Date: 18/07/2023

Signature: [redacted]
Name: Peter Popov
Date: 20 July 2023

These publications are the result of a collaborative endeavour of a working group at CSR, led by Dr. Peter Popov, which included Dr. Kizito Salako and Oleksandr Netkachov. Oleksandr's contributions to the group's results included leading the design and implementation of the HPS simulation engine and visual modelling tools, proposing and implementing a hierarchical composition modelling approach, which enabled modelling system behaviours more effectively. He also implemented the model and its behaviours, such as attackers, control, recovery, overloading, and control infrastructure. In addition, he supported the group with all aspects of designing, running, aggregating, interpreting, and optimising the performance of the model simulations. He developed and refined the modelling methodology based on incremental improvements, which resulted in a more robust and efficient approach.

CITY
UNIVERSITY OF LONDON
EST 1894

STATEMENT OF CO-AUTHORS of JOINT PUBLICATIONS

TO WHOM IT MAY CONCERN

Publications:

Netkachova, K., Netkachov, O., Bloomfield, R. (2015). **Tool Support for Assurance Case Building Blocks Providing a Helping Hand with CAE**. In: F. Koornneef and C. van Gulijk (Eds.): Computer Safety, Reliability, and Security, SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9338, pp. 62-71, 2015. Springer International Publishing Switzerland. doi:10.1007/978-3-319-24249-1_6

Name of candidate: Oleksandr Netkachov

Title of research thesis: Quantitative Resilience Assessment of Critical Infrastructures using High-Performance Simulations

Name of first supervisor: Dr. Peter Popov

We, the undersigned, co-authors of the above publication, confirm that the above publication has not been submitted as evidence for which a degree or other qualification has already been awarded.

We, the undersigned, further indicate the candidate's contribution to the publication in our joint statement below.

Signature: [redacted]

Name: Kateryna Netkachova
Date: 27/07/2023

Signature: [redacted]

Name: Robin Bloomfield
Date: 28/7/23

Under the leadership of Dr. Kateryna Netkachova, Oleksandr Netkachov gathered and organised the requirements, researched and implemented UI forms for editing assurance case building blocks, and integrated them with ASCE. The application constructed and the methodologies developed in this process facilitated the development of the reusable ASCAD templates for assuring security and reliability of critical infrastructures.

CITY
UNIVERSITY OF LONDON
EST 1894

STATEMENT OF CO-AUTHORS of JOINT PUBLICATIONS

TO WHOM IT MAY CONCERN

Publications:

Netkachova, K., Bloomfield, R., Popov, P., Netkachov, O. (2015). **Using Structured Assurance Case Approach to Analyse Security and Reliability of Critical Infrastructures**. In: Koornneef, F., van Gulijk, C. (Eds.): Computer Safety, Reliability, and Security, SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9338, pp. 345-354. Springer International Publishing Switzerland. doi:10.1007/978-3-319-24249-1_30

Name of candidate: Oleksandr Netkachov

Title of research thesis: Quantitative Resilience Assessment of Critical Infrastructures using High-Performance Simulations

Name of first supervisor: Dr. Peter Popov

We, the undersigned, co-authors of the above publication, confirm that the above publication has not been submitted as evidence for which a degree or other qualification has already been awarded.

We, the undersigned, further indicate the candidate's contribution to the publication in our joint statement below.

Signature: [redacted]

Name: Kateryna Netkachova
Date: 27/07/2023

Signature: [redacted]

Name: Robin Bloomfield
Date: 28/7/23

Signature: [redacted]

Name: Peter Popov
Date: 20 July 2023

Oleksandr Netkachov supported the group or authors working on this publication with the expertise in modelling of critical infrastructures, helped formulate reusable ASCAD templates for ensuring system properties through modelling, guided application of the developed templates to Nordic32 system, provided the team with the model simulation results.

Abstract

Accessing the resilience of large cyber-physical systems (LCPS) is essential for ensuring the continuity of operations and minimising the impact of disruptions caused by natural disasters, cyberattacks, and other stressful events. Recent empirical studies of LCPS have demonstrated the usefulness of modelling and simulation in assessing properties that emerge from component interactions, including resilience. However, the sheer complexity of CIs poses challenges for modellers:

1) Resilience assessment requires high-fidelity models that include a probabilistic model of the system and adverse events of interest, such as accidental failures or malicious activities, and a physics simulation model of LCPS processes, such as power/liquid/gas flows.

2) Assessing resilience with high statistical significance requires a systematic exploration of the space of possible adverse events and recovery from their effects. Exploring this space requires a significant amount of effort.

This work offers solutions intended to help modellers overcome these difficulties by using the recent advances in modelling LCPSs and high-performance computing:

i) It offers a new modelling methodology for building agent-based hybrid hierarchical stochastic models using a new domain-specific language. The new modelling approach allows easy integration of a) a variety of modelling formalisms used to model cyber-attacks on CI/LCPS; and b) a set of deterministic models, as needed by the chosen level of fidelity and specific for the modelled CI. However, the deterministic models are not the focus of this work. Such models are assumed to exist in software available from third-party vendors.

ii) It presents a set of tools to support this methodology: the visual modeller and an extensible Monte Carlo simulation engine designed to utilise high-performance and cloud computing capabilities. The engine and the editor utilise modern development practices and technologies to provide a state-of-the-art solution.

This thesis provides a survey of the relevant literature, summarises the progress with the modelling methodology, and presents the results published to date with case studies based on an extended Nordic32, a reference architecture of a power transmission network with the SCADA subsystem. The studies explore the effects caused by adversaries targeting IT infrastructure and demonstrate the application of a defence-in-depth approach to reduce the effects of these attacks.

1. Introduction

1.1 Objectives

Modern systems have made people safer and more secure by providing better access to emergency services, enhancing surveillance and security systems, and improving disaster response. However, as people increasingly rely on digital systems, it becomes crucial to ensure they are reliable, safe, and secure.

Establishing confidence in whether a system can reliably fulfil its purpose (i.e. validation) is an integral part of the engineering discipline. The applicability of different validation methods depends on the characteristics of a system. For dispensable mechanical systems, validation often involves testing the system under various conditions. However, the direct testing approach is unsuitable for larger systems, as it is either expensive, unsafe, or both. System modelling is the only practicably applicable methodology for analysing the behaviour of large systems under severe stress factors, such as natural disasters, massive cybersecurity attacks, overloading, or failures of critical components.

The main idea of system modelling is to create an abstraction or simplified representation of a complex system and then use this model to investigate the system's behaviour under different circumstances. The fundamental problem of modelling is ensuring that the model, despite being an abstraction, retains the essential properties of the modelled system.

The most commonly used approach for modelling large systems is representing the system as a composition of many interconnected components. This modelling paradigm is a core concept of the system dynamics, agent-based, and discrete event simulation modelling approaches. Applying other methodologies, although theoretically possible, may present challenges that are hard to overcome. For example, representing a complex multi-component system as a Markov state machine most likely results in a machine with an enormously large and unmanageable state space.

The central problem in model-based resilience assessment of critical infrastructures is model validation. The standard way to validate is to compare

the model's behaviour with the behaviour of the modelled system under equivalent circumstances. Validating the model of critical infrastructure is usually possible for normal working conditions. However, research often focuses on understanding how the system functions in exceptional circumstances. This work does not address the problem of model validation directly. Instead, it offers advances in creating and simulating models that give practitioners a powerful and flexible tool which can be tuned to operate at the chosen level of abstraction.

For critical infrastructures, exceptional events of significant impact are natural disasters, overloadings, or massive cyber-attacks. These are rare and unique events which may never be seen before assessment. The lack of observations for these events makes the model validation problematic. Therefore, the typical approach for assessing the reliability properties of critical infrastructures is to validate the model reaction against the observed events and extrapolate it. This approach, however, requires a lot of building-simulating-analysing iterations before reaching sufficient similarity between model-generated data and real-life datasets. It is difficult because it takes significant computation resources and requires modelling tools tailored to the studied problem domain.

These objective difficulties are unlikely to be solved in the foreseeable future. However, the right editing tools and performant simulation engines can help reduce the time required to build, adjust, and run the model.

These observations are based on the experience gained while working on the SESAMO project [1] at the Centre for Software Reliability in City, University of London. In this project, the research team applied the "Preliminary Interdependencies Analysis" methodology to assess the resilience of a large power transmission network.

"Preliminary Interdependencies Analysis" [2] is a methodology that helps to understand interdependencies between the elements through the development of a simulatable model, in which the studied part of the system is represented as a set of semi-Markov continuous-time state machines.

The method introduces stochastic associations as a generic mechanism of modelling dependencies between the components of a system with a large number of elements without explicitly generating the *entire system state space,*

which for realistically complex critical infrastructure is too large. The application of the method revealed the following problems: the modelling of large systems is complicated and error-prone; the performance of the simulation engine is not satisfactory; the editing tools do not provide enough support for the modeller.

To adequately address these issues, the questions directing this research have been established as follows:

- What is the most effective methodology for modelling large-scale network systems that accurately captures the intricacies of the system's behaviour, including both probabilistic elements and complex deterministic processes such as power distribution, hydrodynamics, and meteorological patterns?
- How can the duration required for obtaining, aggregating, and interpreting simulation results be minimised, potentially through the implementation of extensive high-performance computing (HPC) optimization strategies and cloud-based parallelisation techniques?
- Which editing features significantly enhance the efficacy of the modelling process? Could it be features such as visualisation and editing tools, access to component libraries, or facilities for running and reporting?
- How do the constructed models and their subsequent results substantiate existing methodologies for assuring system safety and security? Specifically, is it possible to construct reusable patterns that align with current structural approaches?
- What strategies can be implemented to achieve the reusability of the constructed models and, in doing so, expedite subsequent research projects?

The methodology presented in this thesis provides the answers to the above questions. It reduces the complexity of modelling large systems by introducing hierarchical composition and encapsulation. The new simulation engine and task distributor improve the simulation performance and horizontal scaling. The new editor increases efficiency and provides a consistent user interface. The developed methodology and solutions were

applied while researching the effects of cyber-security attacks on Nordic32, a power transmission network.

1.2 Thesis summary

The organisation of this thesis is as follows:

- Chapter 1, "Introduction", defines research objectives, summarises the results, and provides the list of publications.
- Chapter 2, "Background Review", reviews the fields of study relevant to the research: definitions, analysis, and modelling of critical infrastructures; system resilience and assessment; high-performance computing.
- Chapter 3, "Assurance Cases for Critical Infrastructures", provides research results supporting reliability assessment in assurance cases with stochastic models.
- Chapter 4, "Stochastic Modelling and Simulation", presents research results on developing modelling methodology and simulation engine;
- Chapter 5, "Applications", demonstrates the applicability of the developed modelling methodology and tools;
- And final Chapter 6, "Conclusion", concludes the dissertation with a summary and a discussion on the need for future research.
- Appendix 1, "Nordic 32", contains an overview of the Nordic32 power transmission network.
- Appendix 2, "Preliminary Interdependency Analysis", provides an overview of the Preliminary Interdependency Analysis methodology.

1.3 Publications

Netkachov O, Popov P, Salako K. **Quantification of the impact of cyber attack in critical infrastructures.** Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer; 2014. pp. 316–327.
doi:10.1007/978-3-319-10557-4_35

A study on the impact of cyber-attacks on complex industrial systems is reported in this paper. The approach involves building a hybrid model comprising the system under study and an adversary. The model is applied to a complex case study of a reference power transmission network (NORDIC 32), which is enhanced with a detailed model of the computer and communication system used for monitoring, protection, and control. The resilience of the modelled system is analysed under different scenarios, including a baseline scenario where the system operates in the presence of accidental failures without cyber-attacks and scenarios where cyber-attacks can occur. The study's findings are discussed, and future research directions are outlined.

The main research results presented in this article contribute to Chapter 4 and Chapter 5. The methodology, modelling language, and simulation approach are described in Chapter 4. Chapter 5 focuses on applying the methodology, and presents the model and research results on resilience assessment.

NNetkachova, K., Netkachov, O., Bloomfield, R. (2015). **Tool Support for Assurance Case Building Blocks Providing a Helping Hand with CAE**. In: F. Koornneef and C. van Gulijk (Eds.): Computer Safety, Reliability, and Security, SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9338, pp. 62-71, 2015. Springer International Publishing Switzerland. doi:10.1007/978-3-319-24249-1_6

The presented tool and methodology in this paper, which are designed to structure arguments in assurance cases, have potential applications for

reliability assessment and assurance cases. The methodology of Claims-Arguments-Evidence (CAE) Building Blocks provides a set of archetypal fragments to support the structuring of cases in a formal and systematic manner. The tool automates the creation of claim structures and manages CAE blocks, which facilitate the development and maintenance of structured assurance cases. Additionally, the paper proposes new visual guidelines named "Helping hand" to aid in the application of the building blocks. The tool has been implemented on the Adelard ASCE platform, and its intended audience includes assurance case developers and reviewers. The tool and methodology provide a valuable framework for building structured assurance cases and can potentially enhance the reliability assessment of critical systems.

The research results presented in the article are related to Chapter 3, which focuses on applying the introduced approach for assessing the reliability properties of a system.

Netkachova, K., Bloomfield, R., Popov, P., Netkachov, O. (2015). **Using Structured Assurance Case Approach to Analyse Security and Reliability of Critical Infrastructures.** In: Koornneef, F., van Gulijk, C. (Eds.): Computer Safety, Reliability, and Security, SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9338, pp. 345-354. Springer International Publishing Switzerland.
doi: 10.1007/978-3-319-24249-1_30

This paper describes an approach for justifying the use of models to ensure the security, reliability, and resilience of critical infrastructures (CI). Due to the challenges posed by complex and interdependent systems and the pace and scale of attacks, model-based approaches and probabilistic design are necessary to evaluate CI. However, it is essential to assess the trustworthiness of these models. To this end, the paper presents a structured assurance case framework based on Claims, Arguments, and Evidence (CAE). The Preliminary Interdependency Analysis (PIA) method and platform are utilised in a case study involving a reference power transmission network with an industrial

distributed system of monitoring, protection, and control. The paper discusses the benefits of the modelling and assurance case structuring approaches, highlights findings from the case study, and outlines future work directions. In conclusion, this approach provides a valuable framework for evaluating the trustworthiness of models used in ensuring critical infrastructure security, reliability, and resilience.

A core contribution to the content of Chapter 3 is the approach described in this article, which involves using stochastic models for assessing system properties in assurance cases.

Netkachov O, Popov P, Salako K. **Model-based evaluation of the resilience of critical infrastructures under cyber attacks**. Critical Information Infrastructures Security. Cham: Springer International Publishing; 2016. pp. 231–243. doi:10.1007/978-3-319-31664-2_24

This paper reports on the results of improved models and simulation engines, which build on the work presented in a previous article titled "Quantification of the impact of cyber attacks in critical infrastructures." The models and simulation engines presented in this paper are applied to a complex case study, specifically a reference power transmission network enhanced with a detailed model of the computer and communication network used for monitoring, protection, and control, compliant with the international standard IEC 61850. The improved models utilise a hybrid approach, where accidental failures and malicious behaviour are modelled stochastically, while the consequences of these failures and attacks are modelled deterministically. The results of the simulations, which include various scenarios of cyber attacks, are discussed and analysed in the context of the resilience of the modelled system. The contributions of the work are mainly related to the content of Chapters 4 and 5, which describe the methodology, modelling language, simulation approach, and applications of the methodology to the case study.

Netkachov O, Popov P, Salako K. **Quantitative Evaluation of the Efficacy of Defence-in-Depth in Critical Infrastructures.** In: Flammini F, editor. Resilience of Cyber-Physical Systems. Springer; 2019. pp. 89–121. doi:10.1007/978-3-319-95597-1_5

The feasibility of quantitative cyber-risk assessment in cyber-physical systems (CPS), such as power-transmission systems, is discussed in this book chapter. Experimental evidence, using Monte-Carlo simulation, is presented to demonstrate that the losses from a specific cyber-attack type can be accurately established using an abstract model of cyber-attacks. The benefits of deploying defence-in-depth (DiD) against failures and cyber-attacks for two types of attackers are established. This study provides insight into the benefits of combining design diversity with periodic "proactive recovery" of protection devices to harden some of the protection devices in a CPS. The results are discussed in the context of making evidence-based decisions about maximising the benefits of DiD in a particular CPS.

The approach employed in this study represents an evolution of the model and simulation engine used in “Model-Based Evaluation of the Resilience of Critical Infrastructures under Cyber Attacks” and is a significant contribution to Chapter 4. The primary contribution to the content of Chapter 5 is the results obtained from the DiD study.

2. Background Review

2.1 Modelling Critical Infrastructures

In the context of a company, the infrastructure is defined by the ISO standard 9001 as a system of facilities, services, equipment, and other assets that support the organisation in delivering a service or product to its customers or clients. This list of assets includes (but is not limited to) premises, supplies, equipment, and information.

Critical Infrastructure (CI) differs from infrastructure by the effect its disturbance causes on the system. As defined in the US President's Commission on Critical Infrastructure Protection Report [3], the US's critical infrastructures "are so vital that their incapacitation or destruction would have a debilitating impact on defence or economic security." The infrastructures in the scope of the commission are information and communications, electrical power systems, gas and oil production, storage and transportation, banking and finance, transportation, water supply systems, emergency services, and government services. In the EU, a critical Infrastructure is defined by Council Directive 2008/114/EC as "an asset, system or part thereof located in Member States which is essential for the maintenance of vital societal functions, health, safety, security, economic or social well-being of people, and the disruption or destruction of which would have a significant impact in a Member State as a result of the failure to maintain those functions" [4]. Specifically, the directive defines "European critical infrastructure" as "critical infrastructure located in Member States the disruption or destruction of which would have a significant impact on at least two Member States.". The UK government defines critical national infrastructure as "Those infrastructure assets (physical or electronic) that are vital to the continued delivery and integrity of the essential services upon which the UK relies, the loss or compromise of which would lead to severe economic or social consequences or to loss of life" [5]. The UK government authority for protective security advice to the UK national infrastructure, National Protective Security Authority (NPSA), recognises 13 national infrastructure sectors: Chemicals, Civil Nuclear Communications, Defence,

Emergency Services, Energy, Finance, Food, Government, Health, Space, Transport, and Water [6].

The "critical infrastructure" term is also used in application to a company or country, defining, in a broader sense, the system's connectivity and distribution assets and processes crucial for the system's existence - significant damage of these assets or disruption of the processes may cause the system to become extinct.

The critical infrastructure consists of highly interdependent systems. For example, financial services highly depend on information and communication services, while the latter highly depend on electricity. Electrical production and distribution, in turn, require transport and financial services. Although the unprecedented level of integration of infrastructural systems nowadays increases efficiency, it also can lead to increased damage as failures can propagate in many directions through the network of the system components.

The dominant modelling approach in the research community is to represent CI as a graph in which nodes represent infrastructure components, either physical or virtual, and edges correspond to the dependencies between the components [7] [8]. Specific components, their properties, types of dependencies, and modelling algorithms vary significantly between models.

Within computer systems, the specific physical and logical components (including human participants) are modelled by creating corresponding software agents, e.g., processes or objects. It is impossible to create a perfect digital copy of a system, “all models are wrong” [9]. Instead, the common practical approach is to model the relevant parts of the system at some level of abstraction, which includes validating whether the created model satisfactorily represents the system for known scenarios. A sufficient similarity between the model and the actual system in known scenarios provides a foundation for trusting the model and deploying it under major stresses.

The important property of the modelling framework is how general it is. A general modelling framework is applicable in many domains. It, however, may require time and effort to apply it for a particular domain as some unique processes, relationships, and entities should be implemented. In contrast, domain-specific frameworks usually fit very well for the specific problem, but the reusability of the constructed models is usually very limited.

The frameworks are very different regarding a community of users, available training materials, and commercial support. These properties, of course, correlate with the generality of the framework. Unsurprisingly, the more general frameworks have a more extensive community, but specialised frameworks may contain quite sophisticated modelling artefacts.

According to Pederson et al. [10] and Eusgeld et al. [11] comparative reviews, the main approaches to modelling CIs are agent-based, system dynamics, input-output model, physics models, Petri Nets, and Markov Chains.

Agent-Based Modelling (ABM) represents the system as a stateful environment that hosts agents - individual entities which can observe the environment's state and other agents through sensors and act according to their perceptions by modifying the state or interacting with other agents. The agent's internal decision-making behaviour can be implemented in various ways, including machine learning (neural networks, etc), state machines (either with Markov property or without, finite or infinite, deterministic or probabilistic, etc), decision trees, and others.

System Dynamics is a method to analyse the system's behaviour over time. The system is modelled as a set of processes and state variables. The processes update the state variables either by decreasing (negative feedback) or increasing them (positive feedback).

Petri Net represents a system as a set of places, transitions, and arcs [12]. The dynamic aspect of the system is represented via tokens (or marks) that sojourn in places and move through the arcs when transitions are enabled. This model was extended by adding non-deterministic behaviour (Stochastic Petri Nets [13]) and later with timed transitions (Generalised Stochastic Petri Nets, GSPNs [14] [15]).

A system can be represented as a Markov Chain (or corresponding Markov State Machine) when its state is a combination of discrete variables and transitions from one state to another are Markovian (memoryless, not dependent on the previous states).

ABM is the most frequently used method for modelling CIs. 14 methodologies out of 31 reviewed by Pederson et al. [10] represent and model

CIs with ABM. In 33 methodologies reviewed by Eusgeld et al. [11] ABM is used in 13, and the use of other methods is shown in the table 2.1.

Underlying method	No. of tools
Agent-Based Method	13
Geographic Information System	6
System Dynamics	4
Statistical Data Analysis	3
Monte Carlo	3
Input-Output Methods	2
Graph Theory	2
Control Theory	1
Miscellaneous	1

Table 2.1: Underlying methods used in tools for modelling and simulation of CIs in review by Eusgeld et al.

Bonabeau [16] stated three main benefits of ABM: i) it captures emergent phenomena, which result from the interactions of individual entities and cannot be reduced to the system's parts; ii) it provides a natural description of a system in a sense that it is more natural to describe how shoppers move in a supermarket than to come up with the equations that govern the dynamics of the density of shoppers; iii) it is flexible as new behaviours or agent types can be added in the straightforward model.

Object-Oriented Programming (OOP) has a lot in common with ABM. Luna and Stefansson [17] noted the similarity between OOP concepts and ABM: encapsulation, inheritance and polymorphism correspond to the self-contained nature of agents of some types participating in communication based on agent type's features. The basic OOP mechanics, however, are too limited to support a variety of behaviours like asynchronous communications, different internal behaviours, control and observing states. As a result, many extensions, frameworks, libraries and tools emerged aiming to circumvent these limitations in modern programming languages for particular environments and problems.

Macal and North [18] in their tutorial on ABM and simulation defined the following activities:

- thinking through an agent model - identify structural elements of the problem domain, their level of autonomy, structure, relationships with other elements and environment, behaviours, motivation and goals, emergence;
- model agents - improve general understanding developed while thinking through the agent model by defining formal types (classes) and their attributes;
- describe agent-based models in some kind of formal notation, especially consider Overview, Design Concepts, and Details (ODD);
- design model element - identify reusable elements, construct agents from templates, apply design patterns;
- advance model - enrich it with distributed computing, machine learning, GIS data or layout, fetch relevant data from relational databases, consider version control system and development environment;
- use software and tools - bring the designed and advanced model to life (perform computation) either by using spreadsheet editors, computational systems, dedicated ABM modellers and simulators, or programming language.

Due to the highly complex nature of particular CIs, many studies have resulted in the construction of hybrid models, which are created by combining multiple models of different kinds, e.g., Markov chains, agent-based, and physics-based. Application of PIA methodology [2] to modelling of the power production and transmission CI to analyse risks in the context of the IRRIIS project is an example of a hybrid model, where the state is captured by probabilistic state machines with the CI dependencies modelled by triggers changing the probabilistic characteristics of the machines' transitions and more complex relationships (relations between power grid substations) are handled by the power flow solver (physics model).

The findings of the studies surveyed in this chapter indicate an ongoing necessity for an enhanced modelling framework that fulfils several key criteria. Specifically, the framework must possess generality, allowing for broad

application across various domains and disciplines. Furthermore, the framework should leverage the latest advancements in computing and modelling technologies, ensuring accuracy and relevance. Additionally, it should be user-friendly, facilitating ease of use by researchers and practitioners. Not least, the framework should make use of high-performance computing techniques as studies with CI typically require extremely high computational intensity. These criteria are fundamental in achieving an effective, efficient, and contemporary modelling framework essential for addressing the challenges of the modern world.

2.2 Assessing Resilience

The work "Resilience and Stability of Ecological Systems" [19] is frequently cited as influential work in which Holling C. S. demonstrated the application of the principle of resilience to ecological systems [20], [21]. In this work, resilience was defined as a measure of the ability of the system to absorb changes in state variables and persist. Since then, "resilience" emerged as a transdisciplinary concept that applies to systems in a range of disciplines: social science, construction, engineering, economics, medicine, environmental studies and many others [22].

The meaning of the term "resilience" experienced a significant shift from the Holling's definition, and in modern publications "resilience" is described or defined as:

- "Cyber resilience refers to the ability of digital systems to prepare for, withstand, rapidly recover and learn from deliberate attacks or accidental events. It encompasses people-centred aspects of resilience such as reporting, crisis management and business continuity." cyber safety and security [23];
- "an ability to deliver, maintain, improve service when facing threats and evolutionary changes", information and communication technology [22];
- "the ability of an organisation to absorb and adapt in a changing environment to enable it to deliver its objectives and to survive and prosper", organisational management [24];

- “by the functionality of an infrastructure system after a disaster and also by the time it takes for a system to return to pre-disaster levels of performance”, transportation [25];
- “a tendency has been to use it [resilience], in each specific community, to indicate a more flexible, more dynamic and/or less prescriptive approach to achieving dependability, compared to common practices in that community” [26];
- “the ability to adapt to changing conditions and withstand and rapidly recover from disruption due to emergencies”, US Department of Homeland Security [27];
- “Infrastructure resilience is the ability of assets and networks to anticipate, absorb, adapt to and recover from disruption.” UK Cabinet Office [28].

Pimm's definition of the resilience, “the rate at which population density returns to equilibrium after a disturbance away from equilibrium”, [29] although attributed as “engineering resilience” to be distinguishable from “ecological resilience” [30], was used as in many research project and publications.

Bruneau et al. [31] proposed a measure of resilience as a total loss in quality occurred as a result of an event:

$$R = \int_{t_0}^{t_1} (100 - Q(t))dt$$

where t_0 is the time of the event, t_1 is the time of full recovery, $Q(t)$ is the quality of the analysed system, ranging from 0 (no service) to 100 (no degradation).

Tierney and Bruneau [25] introduced a visual concept of the “resilience triangle”, shown in figure 2.1. This concept was used by Adams et al. [32] to analyse the effect of extreme weather conditions on transportation. Their quantitative approach for calculating resilience resulted in categorising weather conditions by the effect they cause. Their work also shows two stages of the system's response to the disruption: reduction and recovery.

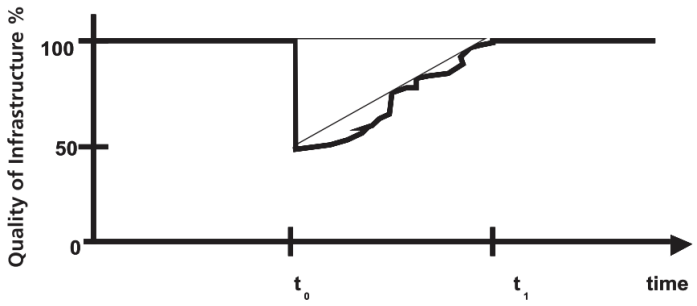


Figure 2.1: Resilience triangle, introduced by Tierney and Bruneau.

Gluchshenko and Foerster [33] proposed a quantitative measure of resilience based on comparing time of deviation T_d with time of recovery T_r, where the time of deviation is a time between leaving the normal state and reaching an extremum and the time of recovery is a time between an extremum and returning to the normal state. They introduced three levels of resilience:

- high resilience - the time of deviation is considerably longer than the time of recovery, i.e. $T_d \gg T_r$;
- medium resilience - the time of deviation and the time of recovery are approximately equivalent, i.e. $T_d \approx T_r$;
- low resilience - the time of deviation is considerably shorter than the time of recovery, i.e. $T_d \ll T_r$.

In addition, they proposed measurable robustness defined as the amount of stress the system can accumulate without leaving the normal state, which can be alternatively defined as a time the system withstands a deviation.

The figure 2.2 demonstrates the generalised resilience triangle for a highly resilient system, system's robustness R, time of deviation T_d, time of recovery T_r, as defined in Gluchshenko and Foerster. As the load on the system increases, the system continues to operate at its normal level until its robustness is depleted. Then, the system gradually decreases its level of operation until the accumulated stress is dissipated. Then the system quickly returns to normal operation.

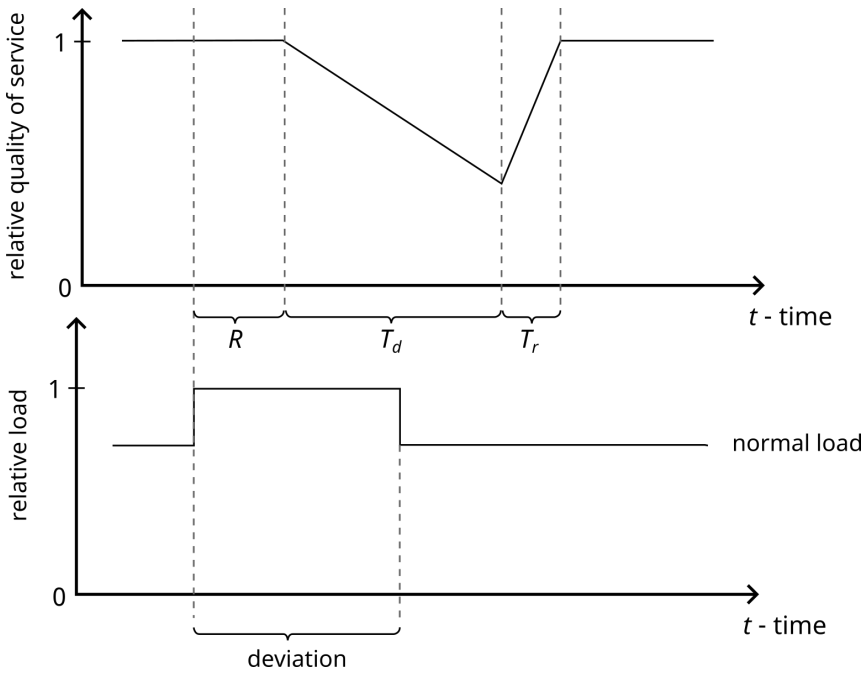


Figure 2.2: Conceptual view of resilience triangle for a highly resilient system.

The research studies and engineering techniques based on the definition of resilience as the rate of recovery recently formed a recognisable engineering specialisation known as "resilience engineering" [34]. The recovery from disruption is part of the official definition of the resilience of the US Department of Homeland Security [27] and the UK Cabinet Office [28].

The framework for assessing resilience proposed by Bruneau et al. (2003) [31] conceptualises resilience of socio-technical systems and infrastructures as encompassing the technical, organisational, social, and economic dimensions and consisting of the robustness, redundancy, resourcefulness, and rapidity properties. It defines resilience as an integral loss in quality over time and suggests finding different performance measures for systems under assessment.

Bloomfield and Gashi (2008) proposed a risk-based framework for assessing resilience based on distinguishing two types of resilience: to design basic threats and their effect on availability, robustness, confidentiality, integrity and resilience beyond basic design threats - to threats that are unknown when the system is designed or assessed. The proposed framework was built by combining existing risk assessment techniques such as HAZOP, the definition of resilience and research trends in assurance cases, the discovery of interdependencies, formal methods, static analysis,

fault-tolerance assessment, benchmarking, modelling with a view on the assessed system as a system of systems.

Devanandham and Ramirez-Marquez [20] provided an illustrative example of assessing change in system resilience when comparing recovery strategies. Starting from defining the loss function (figure-of-merit) and deriving quantitative resilience metrics from this function, they set system boundaries, identify risks, build models, perform calculations and analyse results.

The brief and concise algorithm for assessing the resilience of a system was published by Gluchshenko and Foerster [33]:

- define and describe the system and its boundary to the environment;
- specify the scale and/or the level of hierarchy to observe;
- define the performance indicators;
- specify the reference state of the system;
- indicate and classify disturbances by type, frequency, intensity and duration;
- set the time horizon and investigate resilience or robustness of the system.

Based on the observed studies on assessing resilience and experience in modelling critical infrastructures, it can be concluded that evaluating the resilience of large cyber-physical systems to significant events requires the creation of a hybrid model that incorporates both the system and the adversary. This model should be constructed at an appropriate level of abstraction and verified by comparing its predictions with data collected from observed events. Once the model is verified, the system's resilience can be assessed by observing the service degradation of the model when the disturbing factor is introduced and monitoring the recovery process as the disturbing factor is lifted. This approach provides valuable insights into the system's resilience, facilitating the identification of vulnerabilities and the development of mitigation strategies to enhance the system's overall resilience.

2.3 High-Performance Computing

High-performance computing (HPC) refers to using computer clusters, supercomputers, and parallel processing techniques to perform computationally intensive tasks. It requires specialised hardware devices to achieve high processing power:

- CPUs: Central Processing Units (CPUs) are the main processing units in HPC systems. CPUs are responsible for executing instructions and performing arithmetic and logical operations. HPC systems often use multiple CPUs, which can be arranged in clusters to provide increased processing power.
- GPUs: Graphics Processing Units (GPUs) are specialised processors designed to handle complex graphical computations. However, GPUs are also useful for HPC, particularly for parallel processing tasks.
- FPGAs: Field-Programmable Gate Arrays (FPGAs) are specialised hardware devices that can be reconfigured to perform specific tasks. FPGAs are particularly useful for processing large datasets and can be customised to handle specific data types and processing needs.
- ASICs: Application-Specific Integrated Circuits (ASICs) are specialised chips designed for a specific purpose. In HPC, ASICs are used for specialised applications requiring high processing power, such as encryption and decryption.
- Memory: Memory is a crucial component in HPC systems, as it determines the amount of data that can be processed at once. HPC systems often use high-speed memory devices, such as solid-state drives (SSDs) or dynamic random-access memory (DRAM), to achieve faster processing speeds.
- Interconnects: Interconnects are specialised networking devices used to connect multiple CPUs, GPUs, and other devices in a high-performance computing cluster. High-speed interconnects are essential for achieving the low latency and high bandwidth required for HPC applications.

Cloud-based HPC has become increasingly popular recently as businesses and individuals seek fast, efficient, and cost-effective ways to access specialised HPC hardware devices and configurations without investing in their

infrastructure. Cloud companies have recognised the demand for HPC capabilities, and many are now offering high-performance computing devices on demand. These devices typically include powerful processors, large amounts of memory, and specialised software and tools for running complex applications. Cloud-based HPC solutions also offer flexibility in terms of scalability, as users can easily adjust their computing resources as their needs change. Businesses and organisations can easily ramp up their computing power during peak periods or reduce their usage during lower demand.

As a recognisable field of study, HPC focuses not only on hardware but also on theoretical and practical aspects of algorithm implementations. HPC is generally about achieving maximum performance for a family of algorithms by utilising available computational resources or engineering a custom solution [35].

The software frameworks and libraries for HPC address the challenges raised by the problem of orchestrating a computation on a large number of processors, considering network bandwidth, correctness and ease of programming. OpenMP is considered an industry standard for implementing HPC algorithms using shared memory [36].

The computer program's performance depends on how effectively the program manages the CPU and communication devices. On the micro-level, it may include minimising cache misses, organising the code in a way that keeps the CPU conveyor busy, reducing wrong branch prediction, and optimising memory reads. The macro-level computational performance depends on the task scheduler, managing shared resources and utilising network bandwidth.

Machine code instructions are executed by CPU directly, thus implementing algorithms directly in machine codes (or Assembler) and applying the platform-specific optimisations gives the maximum performance. However, the result might not be transferable to other architectures. The Lack of high-level abstractions for data structures in Assembler makes algorithm implementations more verbose and more challenging to maintain than implementations in other languages. Progress in compilers capable of generating efficient code with applied architecture-specific optimisations from high-level languages made

programming in Assembly completely ignored while considering language for high-performance algorithm implementation.

According to many studies and benchmarks, the top most popular and performant languages for HPC are C/C++ and Fortran. However, there are also languages that demonstrate performance similar to C/C++ on a range of tasks, also providing additional benefits to developers: simplified memory management, abstraction for data structures, advanced syntax, and portability.

Performance comparison of 29 modern programming languages [37] shows that the most performant implementations of the benchmarked algorithms are in C. Other languages with comparable performance: C++, Rust, Julia, Fortran, C#, Chapel, Ada, Haskel, FreePascal, Go, F#, Swift, Java, Lisp, OCaml.

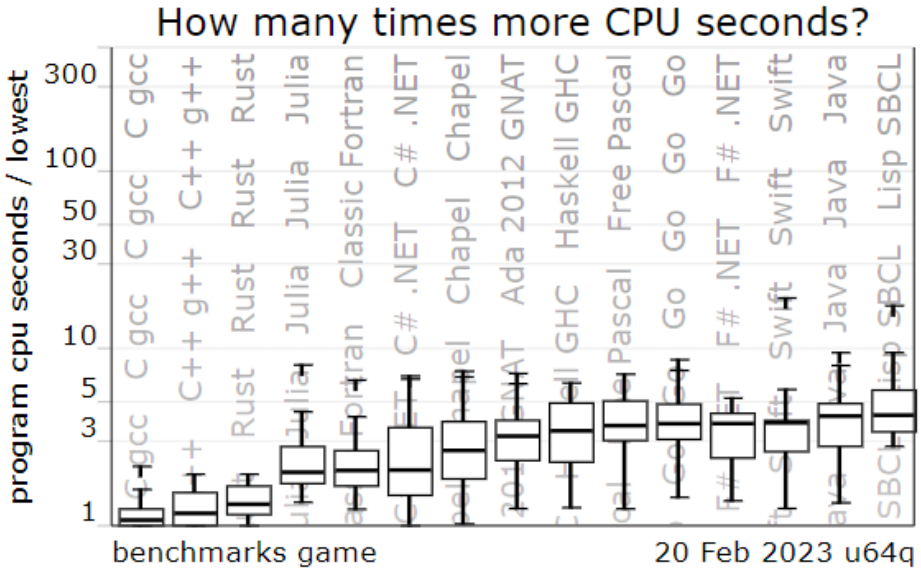


Figure 2.3: Performance of top 15 implementations of tested algorithms in Benchmarks Game.

The comparative review of the RosettaCode algorithm implementations in the most popular programming languages [38] led to the following observations:

- most popular languages by category are: procedural - C and Go, object-oriented - Java and C#, functional - F# and Haskel, scripting - Python and Ruby;
- functional and scripting languages provide significantly more concise code than procedural and object-oriented languages;

- C is the best on computing-intensive workloads and Go is close to C in performance, other compared languages are slower;
- procedural languages use significantly less memory than other languages;
- compiled strongly-typed languages are significantly less prone to runtime failures than interpreted or weakly-typed languages; Go is the least failure-prone performant language in the study.

The performance of the C programming language is unbeatable. By design, the statements and data types are mapped to typical machine instructions very efficiently. It is also one of the most popular languages, so the hardware vendors optimise their compilers and tools to produce the most efficient machine code from C code [39].

Thus suitability of other languages for HPC applications highly depends on whether it can call high-performance libraries (usually written in C or Fortran) efficiently. For example, Python, which is increasingly popular in the research community, has adapters (bindings) to the efficient data processing libraries: NumPy, SciPy, and pandas [40]. By combining calls to efficient libraries with high-level management code, Python programs can successfully compete with C in terms of performance. However, there are a few limitations of this approach: i) performance degrades quite quickly when the amount of computations performed outside of high-performance libraries grows; ii) interoperability with other libraries depends on whether efficient bindings exist for them; if two previous limitations are successfully addressed by extending the host language with C/C++ functions, maintaining the result requires advanced programming skill.

The limitations of the C language inevitably encouraged programmers to extend the language with domain-specific features or syntax constructions supporting other programming paradigms. The most successful, of course, has been the C++ extension that added constructions to facilitate object-oriented programming. Another direction in which the C language was extended is parallel programming. It was supported in the language by either additional libraries and frameworks, like OpenMP, or with language extensions, like Unified Parallel C.

In recent years several alternatives to the C language emerged. Swift, Rust, Go are comparable with the C programming language in terms of computational performance while providing developers with constructions that simplify designing parallel algorithms and make the application less error-prone.

Swift [41] is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. Its features and patterns include compulsory initialisation of variables, checking array boundaries, checking overflows for integers, ensuring that nil values are handled explicitly, automatic memory management via reference counting, and error handling. Its performance for tasks where memory management is not involved is close to C/C++.

Rust [42] is a systems programming language sponsored by Mozilla Research. Its main design goal is to be fast, concurrent, and safe. It supports functional and imperative-procedural paradigms. Its strategy to make code safer is to avoid null pointers, dangling pointers, or data races. It manages memory by "resource acquisition is initialisation" (RAII) with optional reference counting. On most of the benchmarks, its performance is very close to C/C++ code.

Go [43] is a general-purpose, procedural (like C but with limited structural typing), compiled programming language developed by Google. One of the design goals of the Go programming language was "The efficiency of a statically-typed compiled language with the ease of programming of a dynamic language". The garbage collector manages the memory within the Go process. The Go language syntax of the language is one of the simplest, albeit it supports advanced language elements like first-class functions and closures. The Go language contains an integrated concurrent programming model similar to CSP. This model simplifies parallel algorithms and utilises the available multi-core CPU resources efficiently. The performance of the Go programs is quite close to C/C++, and there is no strong evidence that algorithm implementation in Go is always slower than the implementation in C/C++.

In summary, multiple factors should be taken into account to achieve the best performance while developing a hybrid model of a large complex system and running simulations. These factors include the network topology, how

different scenarios change the computation hardware requirements, how third-party physics models are implemented and integrated with the model, how frequently and easily the model needs to be adapted for new requirements, whether the data is sensitive and can be transmitted to the cloud, whether the focus of the simulations is a small number of complex simulations or a large number of simple simulations, and whether a single simulation benefits from running on specialised hardware such as GPUs, FPGAs, or ASICs. These factors may affect the selection of technology, algorithms, hardware, and premises for the computation. Achieving the peak possible performance requires making the right decisions for these trade-off questions. Therefore, a comprehensive and systematic analysis of these factors is essential for developing and implementing a hybrid model of a large complex system.

3. Assurance Cases for Critical Infrastructures

3.1 Overview

Effective reasoning about safety, security, reliability, and assurance requires a structured approach. Many years of research resulted in a number of approaches being developed. They were applied while constructing and maintaining assurance cases for a wide range of devices, constructions, and networks. Structured assurance cases were developed for miniature medical sensors, nuclear power plants, and multinational power networks.

Decades of research in structured assurance cases showed that building sound assurance cases is difficult. The basic model of argumentation developed by Toulmin [44] was one of the first attempts to address this difficulty by introducing structured graphics notation. Studies that followed contributed to increased confidence in the fact that structuring an assurance case increases clarity and understanding, contributes to better decision-making, reduces risks, and improves accountability.

Among the factors that lead to choosing the best approach for a particular assurance case, the most important one is which processes the assurance case should support within an organisation. Characteristics of these processes lead to choosing between a less structured approach, e.g., a list of facts in presentation, and a more sophisticated one, such as using visual notation (e.g., ASCAD [45] or Goal Structuring Notation, GSN [46]) and special editing tools.

Assessing safety, security, reliability, resilience, and other system properties of a large cyber-physical system is only possible in one of two ways: constructing argumentation considering the observed operation history or substituting the system with an appropriate model (a “digital twin”) to study how good the real system.

Substituting the system with a model, together with the introduction of blocks [47], and Assurance 2.0 [48], helps create more rigorous assurance cases. With the credible simulatable substitution of the system, the natural language claims about the system’s reliability properties can be replaced, by

applying the Substitution and Conretion CAE blocks, with the appropriate quantitative evidence such as distributions against failures and intrusions obtained through simulation.

For this approach to be successful, the credibility of the model needs to be confirmed by a separate justification that the model and the tool can be trusted.

Confidence in the results obtained with the model (e.g. via solving the model using Monte Carlo simulation) depends on whether the model represents the real system accurately, i.e. with a sufficient level of detail and at the right level of abstraction. In other words, whether the model of the system behaves close enough to the real system in situations that are relevant to the assessed properties. There are different strategies for addressing this challenge, e.g. comparing the model behaviour and observed behaviour of the real system under the same environmental circumstances.

Several aspects contribute to the trustworthiness of the simulation engine when it is applied to a particular scenario. Among them is whether there are successful applications of the simulation engine for similar use cases, whether the simulation engine produces results similar to observations under the equivalent modelled circumstances, and whether the quality of the simulation engine software is adequate. For any particular application of the engine, applicability analysis needs to be performed, in which the above and other relevant factors should be considered. Although the following chapters provide input for this analysis, such as the architecture of the software, comparison of implementation, and use cases, exploring the applicability of the engine for different scenarios is not a part of this thesis.

A number of studies and research projects, including IRRIS and SESAMO [49,50], demonstrated that in practical applications substituting a large and complex system with a formal model, which can be solved analytically (e.g. state machine), is either very hard or practically impossible. As a result, most of the substituting models are hybrid agent-based models, i.e. the individual parts of the system are modelled independently with technologies most appropriate for a particular component or process. For example, the individual independent components can be modelled as state machines, and they participate in or are affected by a computational approximation of a physical or chemical process.

Processes happening in large systems are frequently probabilistic - the events of interest such as disturbances, failures or malicious interventions can be captured by a suitably chosen stochastic process. Grasping the stochastic aspects of the component behaviour is one of the challenges the model developers need to take into account when substituting the real system with its model. It is a common practice to use probabilistic state machines in continuous time with distributions of the time spent in a particular state assigned to state transitions. Some of the probabilistic parameters often can be estimated from available operational data (e.g. for accidental failures of the components). Some other parameters, however, may be difficult to estimate due to lack of sufficient operational data (e.g. on malicious interventions). In such cases one can deploy "sensitivity analysis" and study the model behaviour under different values of the parameters, which are difficult to estimate. As a result of sensitivity analysis, one gains an insight as to how different parameters affect the system properties of interest and concentrate on those parameters which affect significantly the properties of interest. The properties of such models are calculated by observing and aggregating the outcomes of repetitive simulations, i.e. using the Monte-Carlo method.

The results of model simulations, either deterministic or stochastic, e.g. in the form of the probabilistic distribution of a variable of interest (often referred to as a "utility function" or "reward"), can be used to support the argumentation in the assurance case. In the Claim-Argument-Evidence approach, as will be shown in the next section, simulation results can be incorporated in the assurance case as evidence.

3.2 CAE Assurance Cases

Claim-Argument-Evidence (CAE) approach [45,51] provides an effective methodology for developing, maintaining, and communicating cases. Its graphic notation, ASCAD, is used to visualise and organise relations between the claims, argument, and evidence.

Using stochastic models in CAE assurance cases requires developing a systematic and practical approach. It should provide guidance on how to perform decomposition of the *top claim* and identify system properties that can

be calculated, justify the substitution of the system with its model, build the credible model, obtain and interpret model simulation results, and incorporate them into the assurance case.

The key elements of a CAE approach are the following:

- Claim - a statement about the system, its parts, or operation context.
- Argument - a structured and systematic way of arguing that upholds the claim through more detailed sub-claims or by providing evidence either supporting or refuting the claim and subclaims.
- Evidence - support of the claim (i.e. that the statement captured by the claim is true), e.g. formal analysis, design, verification. The evidence may be supportive of the claim or otherwise (i.e. can contradict the claim).

The graphical notation ASCAD provides a framework for visualising relations between claims, arguments, and evidence. The Adelard ASCE tool provides a visual editor for creating and maintaining structured safety cases using the CAE approach and the ASCAD graphical notation.

The normal form of a CAE assurance case requires it to begin with a top-level claim, which is justified by argument and supporting sub-claims and evidence nodes. The top claim formulates the general assertion about the system, e.g. "the system is safe for a given application in a given environment". The practical approach to justification may begin with expressing initial thoughts as a diagram, showing the factors that influence the claim, followed by iterative application of CAE building blocks, improving understanding, and using sophisticated engineering models.

CAE building blocks are archetypal CAE fragments. Bloomfield and Netkachova defined the following five CAE blocks [47]:

- Decomposition - deducing conclusion about the claim through claims or facts about constituent parts.
- Substitution - replacing the claim about the system with a similar claim about an equivalent system.
- Evidence incorporation - justifying the claim with evidence.
- Concretion - providing a more precise definition or interpretation of the claim.

- Calculation - justifying the claim through computing a numerical value, e.g. evaluating a formula or gathering data while simulating a model.

Architecting an assurance case is a creative process. Creating an assurance case for simple systems might be possible by starting with a top claim and just expanding the claim's tree up to evidence leaves. However, in the most practical applications, this process is different. It begins with expressing the initial thoughts informally, in lists, notes, and simple diagrams. While the understanding of the system improves, the structure of the case becomes more formal. With this approach, whether or not part of the system can be modelled and used to justify claims can be decided at the beginning. This decision influences the development of the case, e.g. the Decomposition block may emerge to narrow down the part of the system that is modelled, and the Concretion block may be used to transition from human-readable claims to claims that can be calculated. On the path from the top claim to the Calculation block should be an instance of the Substitution block. At this instance, it is crucial to justify that the model is indeed an equivalent of the real system and suitable for providing evidence through calculating.

3.3 Using Stochastic Models in CAE Assurance Cases

While assessing the properties of interest of a system, the results from stochastic model simulation can provide evidence for claims in CAE assurance cases [52]. However, these results can provide sufficient evidence only when the credibility of the model is assured.

In CAE methodology, the purpose of the Substitution block is to replace a claim about the system with an equivalent claim about the substituted system, provided that the second system is an *adequate* substitution of the first system. For example, if the second system is a successfully validated model of the real system, then the claim about the real system can be substituted with a similar claim about the model.

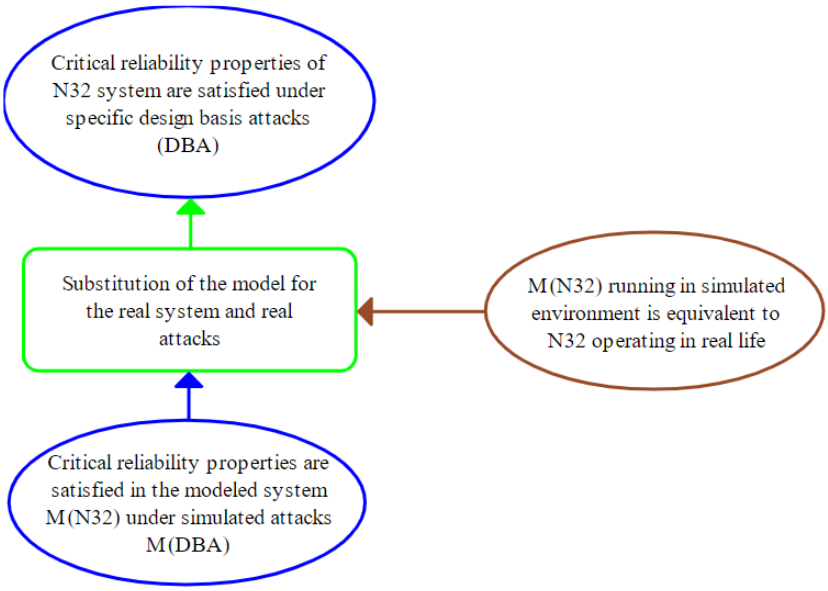


Figure 3.1. Substitution of the model for the real system.

Figure 3.1 demonstrates an example of ASCAD assurance case in which the real system, the Nordic32 power transmission network, is substituted with its stochastic model. The side-claim supports the justification that the model adequately represents the real system for this specific purpose. The justification is based on confirming that the simulation platform is trustworthy and relevant simulation models are validated, i.e. adequately representing reality.

The assertion concerning the reliability properties requires decomposition into several claims about particular properties. The properties of the power distribution system are quantitative characteristics important to the consumers, e.g., total losses. Figure 3.3 demonstrates the decomposition of the general claim about the system's reliability into concrete claims about specific properties.

The aggregated statistical results from the Monte-Carlo simulations of the Nortic32 stochastic model support the claims about the properties of the real system once the subclaim that the model of the system is credible..

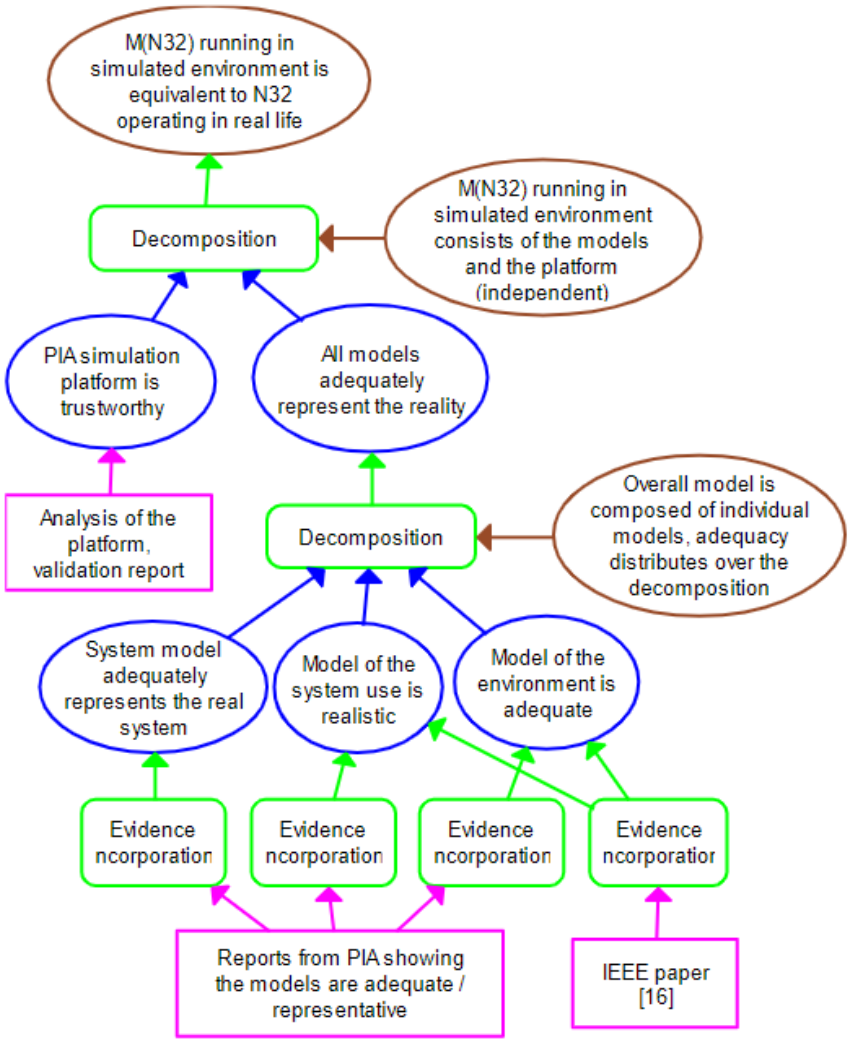


Figure 3.2. Expanded side-claim, justification of the model substitution [52].

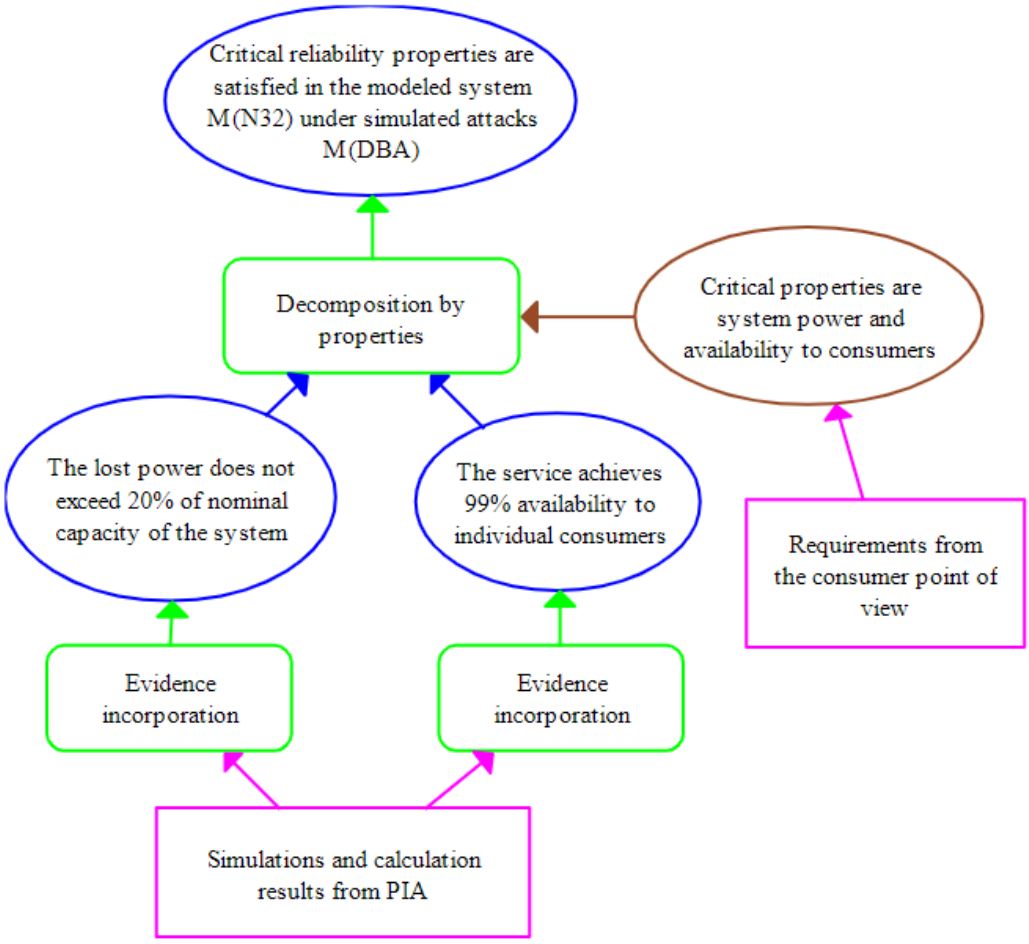


Figure 3.3. Demonstration of the decomposition of the general claim into claims about specific properties.

4. Stochastic Modelling and Simulation

4.1 Overview

My research on substituting real systems with models, as shown in the previous chapter, resulted in developing the following key requirements for a modelling and simulation framework:

- It should support *hybrid models*, incorporating different approaches, such as agent-based models, probabilistic (stochastic) models, and deterministic physics models (e.g. power-flows in power systems).
- It should be capable enough to provide the methods for modelling the system's components and the stress factors of interest, among them the state of the *operational environment* (e.g. the weather conditions), accidental component failures and repairs, the actions of malicious agents (adversaries), etc.
- It should be supported by software tools which allow for fast model creation and for efficient model validation, which must be user-friendly, providing researchers and practitioners with a consistent user interface.
- Furthermore, it should leverage the latest advancements in high-performance computing, including cloud-on-demand HPC capabilities, so that the complexity of "solving" the models and obtaining useful estimates of system resilience can be achieved in a timely manner.
- The implementation technology must be carefully selected, too, considering the trade-offs between performance, ease of extending and modifying the system models, and integration with existing third-party solutions (e.g., libraries, executables, services, and hardware components).

The modelling framework presented in this section attempts to fulfil these requirements by: i) providing a *new domain-specific language* for defining hierarchical models, ii) an *extensible simulation engine* that supports local and cloud-based simulation agents, and iii) a *Web-based editor* that provides a

visual aid to model developers. The engine and the editor utilise modern development practices and technologies to provide a state-of-the-art suite.

4.2. Modelling

This section provides an introduction to the HPS modelling methodology. Within this introductory section, the subsequent terms are defined with their corresponding meanings (in alphabetical order):

- ***Component*** is a constituent part of a system. In the model, a *component* is represented as a *machine*.
- ***Definition*** is a formal and generalised representation of a particular type of *component*. It is a template that defines the properties and behaviours of the *component*.
- ***Instance*** is a concrete representation of a *definition*. It is created by instantiating the *definition* and assigning specific values to its *properties*. It is unique and distinct from other *instances* of the same *definition*.
- ***Machine*** is a formal representation of an acting entity within the *system*. It serves as a means to represent a *component* of the *system* within the *model*. A *machine* is both defined and instantiated within the *model*, allowing for it to be utilised as a building block.
- ***Model*** is a formal representation of a *system*. It is constructed by representing the *components* as *machines*, where *component* types are defined as *definitions* and individual *components* and the *system* as a whole are represented as *instances*.
- ***Plugin*** is a programmatic extension to a *simulation engine* that provides additional functionality beyond what is available through the base functionality of the engine. *Plugins* are designed to address specific needs or requirements that are not possible to capture using *definitions*, such as models of physical processes or other complex phenomena. A plugin typically consists of one or more modules or libraries that are loaded into the *simulation engine* at runtime, and which provide additional functionality or services to the simulation.
- ***Reward function*** is a computation that provides a quantitative measure of the performance of a *system*. The *reward function* is typically defined

in terms of a set of objectives or criteria that the *system* is intended to achieve in the *simulation*. These objectives might include maximising efficiency, minimising response time, or optimising the use of resources. The *reward function* itself is a function that takes as input the state of the *model* at a given time, and returns a numerical value that reflects the desirability of that state.

- ***Simulation*** is a process of executing a *model* and generating output data that reflects the behaviour and performance. It involves interpreting a *model* within a *simulation engine* to simulate the interactions between the *components* of the *system* and producing results that can be used to analyse and understand the *system*'s behaviour under different conditions.
- ***Simulation engine*** is a software program that interprets a *model* and executes a *simulation* of the model. The *engine* uses algorithms and mathematical models for simulating the behaviour of the *components* of the system, as represented in the *model*, and producing output data that reflects the performance and behaviour of the system under different conditions.
- ***System*** is a set of interconnected and interdependent *components* that are orchestrated together to perform a specific set of functions. *Components* of the *system* can be hardware, software, data, people, and social and physical processes.

The aim of the HPS modelling approach is to create a sound, justifiable representation of a *system* amenable to an efficient solution via Monte Carlo *simulation*. The core of the approach is a continuous and incremental process of refinement, in which assumptions about identified components and processes, their relations, interactions, and stochastic characteristics are documented and captured by the developed model. The method is an evolution of the Preliminary Interdependency Analysis (PIA) [2].

Similarly to the PIA approach (spelled out in more detail in Appendix 2), most of the activities in the HPS modelling are either related to a *qualitative* or *quantitative analysis*.

Qualitative analysis is a process of identifying *components* and activities within the system, patterns in component organisation and interaction. This

analysis can be used to gain insights into the emergent behaviour and dynamics of the *system*, as well as the interactions and decision-making processes of individual *components* within the *system*.

Quantitative analysis, in turn, involves the use of statistical and mathematical methods to analyse and interpret the numerical data generated by Monte Carlo *simulations* of the *model* constructed through *qualitative analysis*. The goal of this type of analysis is to quantify the measure of interest (e.g. the loss of supplied energy in power systems due to accidental failures and/or due to successful cyber-attacks), the effect of strength of relationships among interdependent components (e.g. failure propagation likelihood). Identifying "interesting/surprising" patterns and trends within the simulated data is of great interest, too, as they provide the operators of simulated critical infrastructures with insight about how to make the infrastructure more resilient.

The PIA approach recognises two levels of abstraction while modelling the *system* and two models simultaneously developed while modelling the system: the interacting services model (service-level model) and the detailed service behaviour model. The HPS approach extends this to support *multiple levels of abstractions within a single hierarchical model.*

The PIA approach is iterative and based on revisiting earlier stages after progressing on the latter stages. The HPS approach is incremental and focuses on identifying and implementing changes that incrementally improve the model and its artefacts, e.g., documents, definitions, components, libraries, and plugins.

The HPS approach begins with establishing the following major artefact groups:

- **System documentation.** This includes all the available information on the modelled *system*, including requirements, architecture, design, operational environments, adversaries, and telemetry data.
- **Scenarios.** Research objectives, boundaries, selected adversaries, environmental circumstances, performance metrics, and *reward functions.*
- **Modelling artefacts.** This includes *models*, *plugins*, documented assumptions, and *simulation results.*

- **Results.** This includes aggregated *simulation results* and their interpretation.

There are several processes that result in incremental improvements in the artefacts:

- **Collecting information.** This includes collecting initially available system documentation and filling gaps in understanding and coverage.
- **Setting and updating research goals and scenarios.** This includes collecting, defining, and elaborating on research goals and scenarios, defining metrics, determining boundaries and limitations, and assessing feasibility and risks.
- **Modelling.** This includes identifying *components*, their dependencies and interactions and increasing the level of abstraction to a more detailed level by adding implementation.
- **Simulation.** This includes allocating resources, performing *simulations*, and aggregating *results*.
- **Interpreting simulation results.** This involves analysing and interpreting *simulation results* to draw conclusions and make recommendations based on the research goals and scenarios.

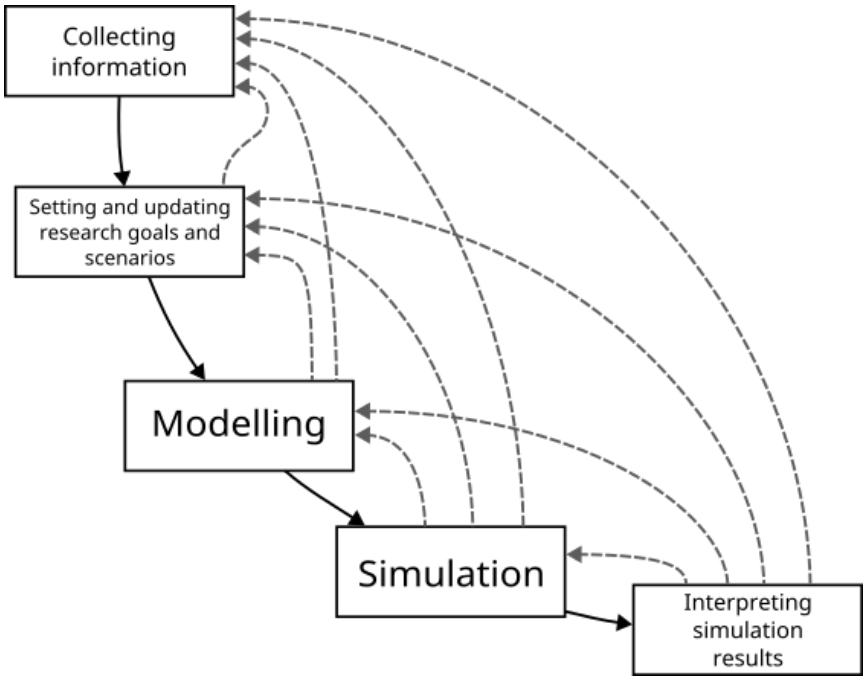


Fig. 4.1. Data flow between HPS processes (⟶) and backpropagation of changes (⇢).

Any of the processes can result in modifying any artefacts. For example, results obtained through *simulation* may reveal a deficiency in the system documentation, justify the investigation of new scenarios or need for performance improvements in the *model*, *engine*, or *plugin*.

The HPS modelling starts by representing the entire *system* as a single *machine*. This component is added to *the library* (HPS term) of identified components. New *components* are identified and organised through a systematic review of the collected *system* documentation, accompanied by setting initial research goals and identifying scenarios.

When a new *component* is identified, it is categorised either as a completely new component and added to the library or as an *instance* of an already identified component. Correctly adding the instance to the model requires identification of the context where this instance operates. When the context is an already identified component, incorporating the instance requires this component to be of a *network* type. If the identified component is already implemented as a *state machine*, the implementation can be shifted down into the new component and the component can be converted to the network of two components, a newly identified instance of an existing component and the instance of the state machine. If the identified component is implemented in any other way, apart from a state machine and network, e.g., as a model of physical process or hardware component, the method of encapsulation of an instance should be found in the documentation for this implementation.

The *state machine* component defines a state machine with a finite number of states and transitions between them. One of the states is an initial one, i.e. when a state machine is created by the simulation engine, it starts with this state. While simulating, the engine transitions the state machine from one state to another by choosing the state that is incidental to activated transitions of the current state. Transitions are activated or deactivated by their associated *triggers*.

The *trigger* is a function that enables and disables *transition*. The trigger can be as simple as the *deterministic trigger*, which activates a *transition* at a specific simulation time, or as complex as a real device, e.g. the *trigger* can be a program that reads measurements from a sensor, which is connected to a general purpose input/output (GPIO).

The *network* is a set of component instances. The instance is defined by specifying the component and providing values for its properties.

Similarly to the *state machine component*, the *network component* also has *properties* which can be mapped to one or many *instances' properties*. It is possible to create complex hierarchical structures by including the *network* instances in the *networks*.

The *component*'s behaviour can be parameterised by defining a *property* and mapping its value to a *trigger*'s or action's property. Consider the following example: the power line *component* of the power transmission *system* is modelled as a *state machine* with *states* ***OK*** and ***Fail***. There are many lines in the *system* and for each of the lines its probability of failure depends on the line's length - longer lines fail more often. Therefore, the *transition* between *OK* and *Fail* is controlled by the *Probabilistic trigger* with the *Exponential distribution*. The straightforward approach to represent the lines in the model would be to define a *state machine* for each line and set the λ *property* of the *distribution* to the corresponding value. However, it creates a lot of duplicates of the same state machine, which greatly complicates maintainability. Much more concise approach is to create a *definition of* the Link *machine* as a *state machine* with one *property* of the *ITrigger* type and use the *Property trigger* to evaluate the *trigger* from this *property*. Then while creating a *network component* that represents the *system*, the links can be represented by instances of the Link *definition* with corresponding property values.

4.3 Logical Model

A deterministic finite automaton (DFA) is defined by a quintet:

$$(Q = \{q_i\}_{i=0}^{n}, \Sigma = \{\sigma_i\}_{i=0}^{k}, \sigma : Q \times \Sigma \to Q, q_0 \in Q, F \subseteq Q)$$

Here Q is a non-empty finite set of states, Σ a non-empty finite set of inputs, the "alphabet", σ maps state and input pairs to new states, q_0 from Q is an initial state, and F is a possibly empty set of final (without outgoing transitions) states from Q.

A DFA can be visualised as a directed multigraph with labelled edges, representing transitions, nodes, representing states, and one node marked as an initial node.

Stochastic State Machines are the result of merging together concepts of probabilistic Markovian transitions and state machines. For the stochastic state machine definition, sets Q and Σ, initial state q_0 and F have the same meaning. The σ function is different. Instead of returning a next state for (state, input) pairs it returns a probability distribution for states

$$\sigma : Q \times \Sigma \to P(Q)$$

where

$$P(Q) = \{p_q | p_q : Q \to \mathbb{R}_{\geq 0}, \sum_{s \in Q} p_q(s) = 1\}$$

When the transition represents an event in continuous time and the probability of these events is defined as a probability distribution function, then the transition function can be defined as follows:

$$\sigma : Q \times \Sigma \to P_t(Q)$$

where

$$P_t(Q) = \{p_q | p_q : Q \to p(t), t : p(t) \geq 0, \int_0^\infty p(t)\, dt = 1\}$$

A hybrid state machine is a state machine with deterministic, discrete stochastic, or continuous-time stochastic transitions. It is defined as follows:

$$(Q = \{q_i\}_{i=0}^n, \Sigma = \{\sigma_i\}_{i=0}^k, \sigma : Q \times \Sigma \to Q \vee P(Q) \vee P_t(Q), q_0 \in Q, F \subseteq Q)$$

When a state machine has continuous-time transitions, the deterministic and discrete stochastic transitions are executed instantly, i.e. the system spends no time in a state prior to the triggered transitions.

Let's define a factory function as a function as

$$f : P_1 \times P_2 \times ... P_n \to M_H$$

where P_i is a set of all possible values for ith property and M_H is a set of all possible hybrid state machines.

A state machine, defined as a quintet of state-space set, input alphabet set, transitions, initial state, and final states set, can be equivalently described as a factory and a vector of property values for this factory.

Let's define a Simulated Stochastic Network recursively as a tuple of

- State machines, defined as quintets.
- State machines, defined as factory functions and property values.
- Simulated Stochastic Network
- Properties, bound to the properties of state machines and networks
- Current simulation state of state machines and networks.

During initialization, the simulation engine recursively constructs state machines from factories.

While simulating, if the state machine factory's range comprises machines with identical state spaces, changing the definition's property substitutes the machine or network. The new machine is in the same state and its continuous-time transitions are recalculated.

The following logical model provides a conceptual view of the domain entities, their attributes and relationships. It can be different in some areas from the implementation model. It is created with abstract data types such as lists, maps, and numbers instead of technology-specific data types. It does not include infrastructural classes. The language of the logical model diagrams is UML2.

4.3.1 Definitions Model

Definitions Model is a model for machine factories. It describes networks, machines, properties, and instances.

The main interface of the HPS definitions model is *IMachine*. It represents an entity within the system, e.g. a device, a person, or a process. The abstract class *MachineBase* provides a default implementation for the properties defined in the interface *IMachine*.

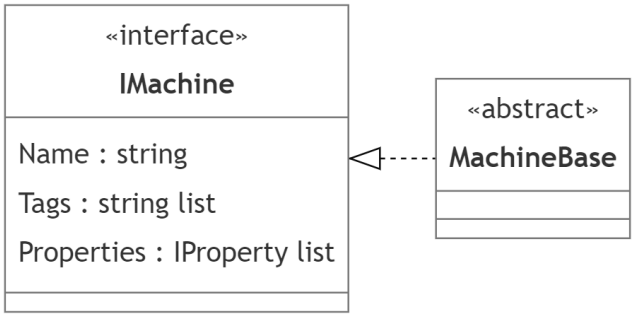


Fig. 4.2. IMachine interface from the definitions package.

The machine's properties serve two purposes:

- It is an observable part of the machine's state. Other machines can subscribe to the property notifications and change their behaviour when the property changes.
- They modify the behaviour of the machine. E.g., the property may be mapped to the parameter of the probabilistic trigger.

The composition pattern of simulated stochastic networks is supported by the structure of classes presented in Fig. 4.3.

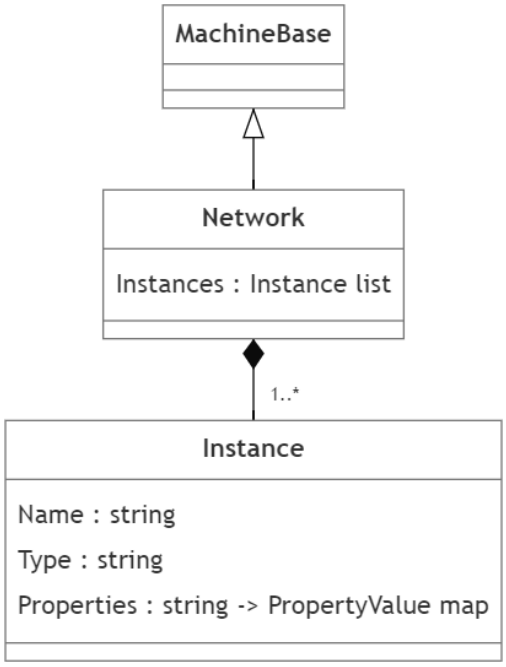


Fig. 4.3. Network and Instance classes from the definitions package.

The implementation of the factory creates the actual simulated state machine by finding the definition specified in *Type*, and then constructing the machine that corresponds to the given property values. PropertyValue has a special semantic in this logical model. It represents a container that can store any value. In the runtime, the exact value will be deserialized and cast to the property's value.

The logical model includes classes for defining state machines, as presented in Fig. 4.4.

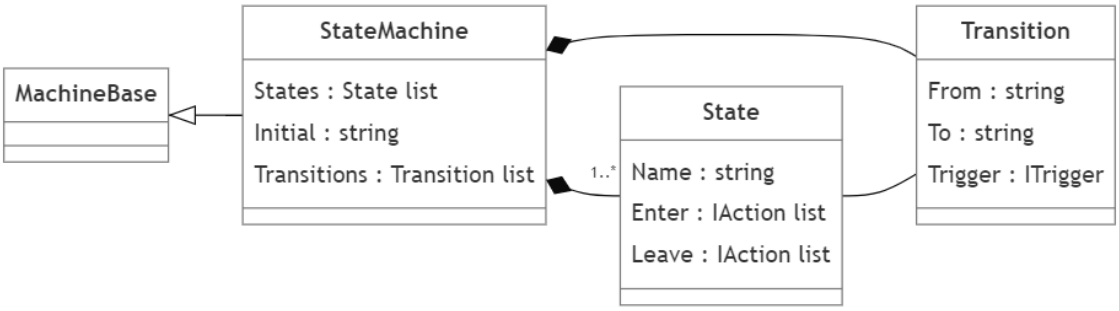


Fig. 4.4. State machine classes from the definitions package.

The logical model defines implementations of the interfaces *ITrigger*, *IAction*, and *IProperty* that were found to be useful while working on the use cases.

The abstract class *TriggerBase* provides the default implementation of the interface *ITrigger*.

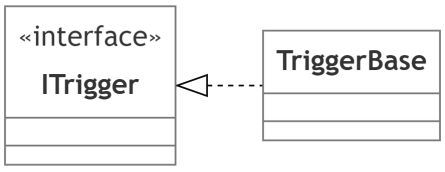


Fig. 4.5. *ITrigger* interface and the abstract base class.

The *Deterministic*, *Probabilistic*, *Instant*, and *Idle* triggers represent different transition types:

- **Idle** - this trigger never invokes the transition and can be used as the initial value of the property.
- **Instant** - this trigger invokes the transition immediately and can be used as the initial value of the property. This kind of transition is known as "an automatic transition" or "an eventless transition" [53].
- **Deterministic** - triggers transition in the specified time.
- **Probabilistic** - the time of triggering is probabilistic, distributed according to the value of the attribute "distribution": *normal* - normal distribution parameterised by the attributes "mu" (parameter μ) and "sigma" (parameter σ); *exponential* - exponential distribution,

parameterised by the attribute “lambda”, which corresponds to the parameter λ.

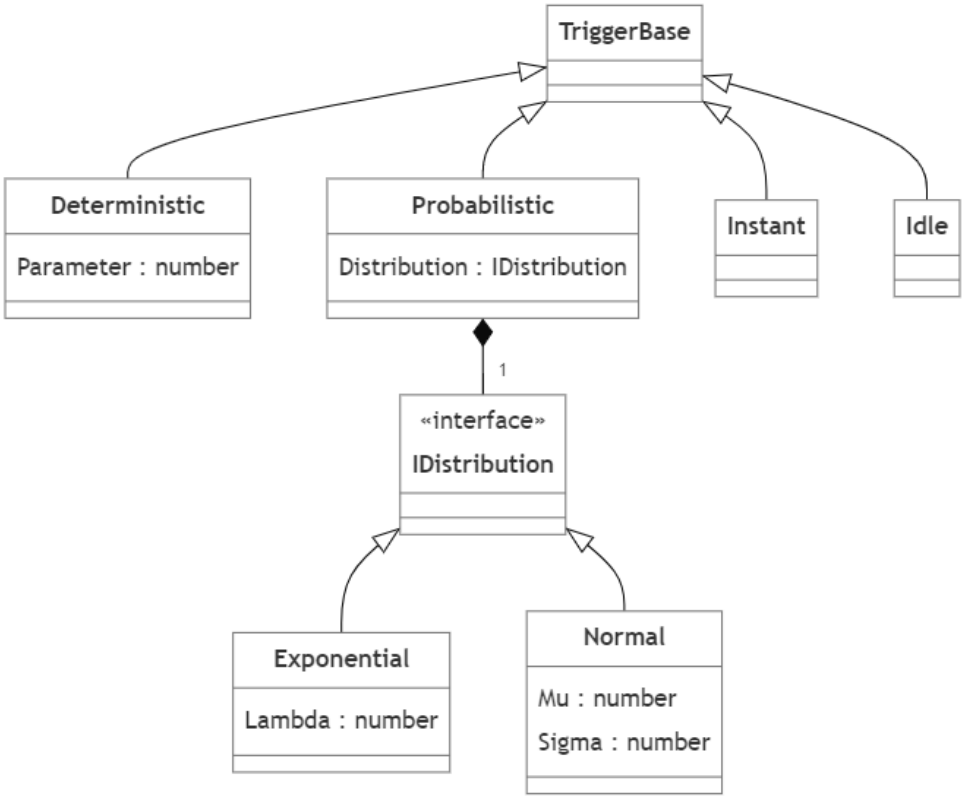


Fig. 4.6. Triggers, representing different transition types.

The *Gate* trigger enables or disables the transition depending on the other machine’s property or state.

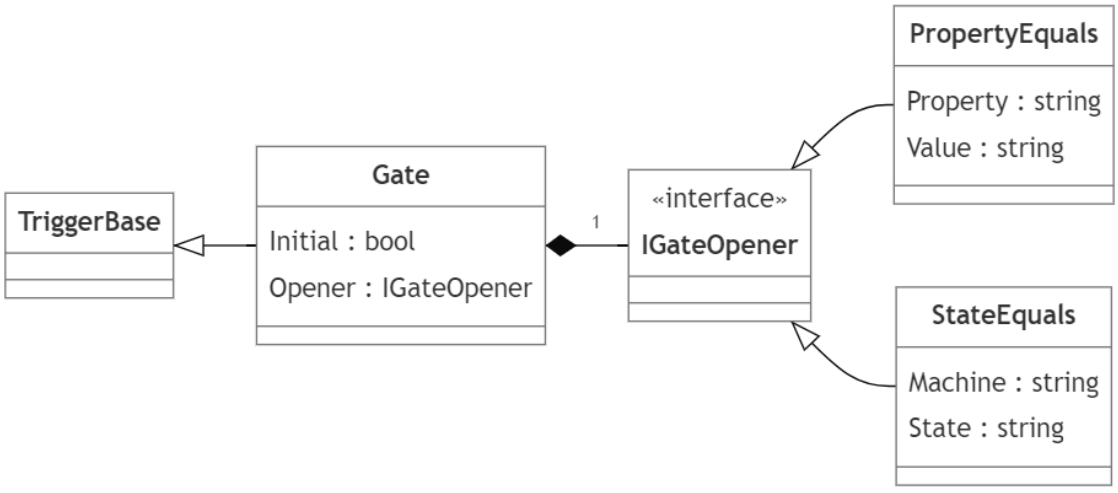


Fig. 4.7. The *Gate* trigger and different gate openers.

The special *Property* trigger maps the trigger of the transaction to the machine's property.

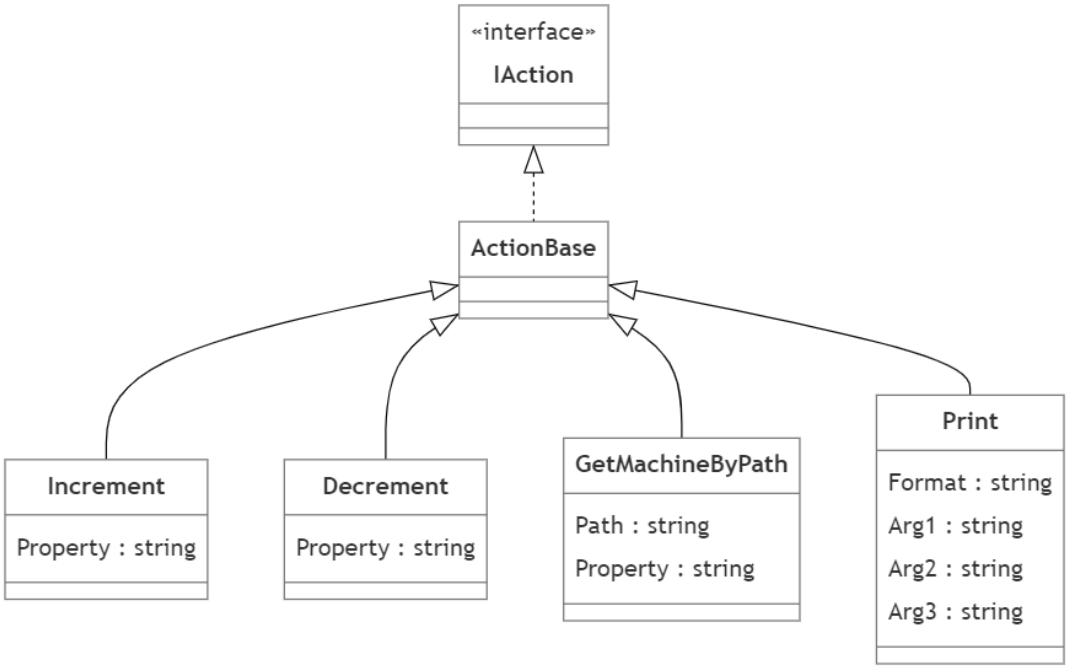


Fig. 4.8. Implementations of *IAction* in the definitions package.

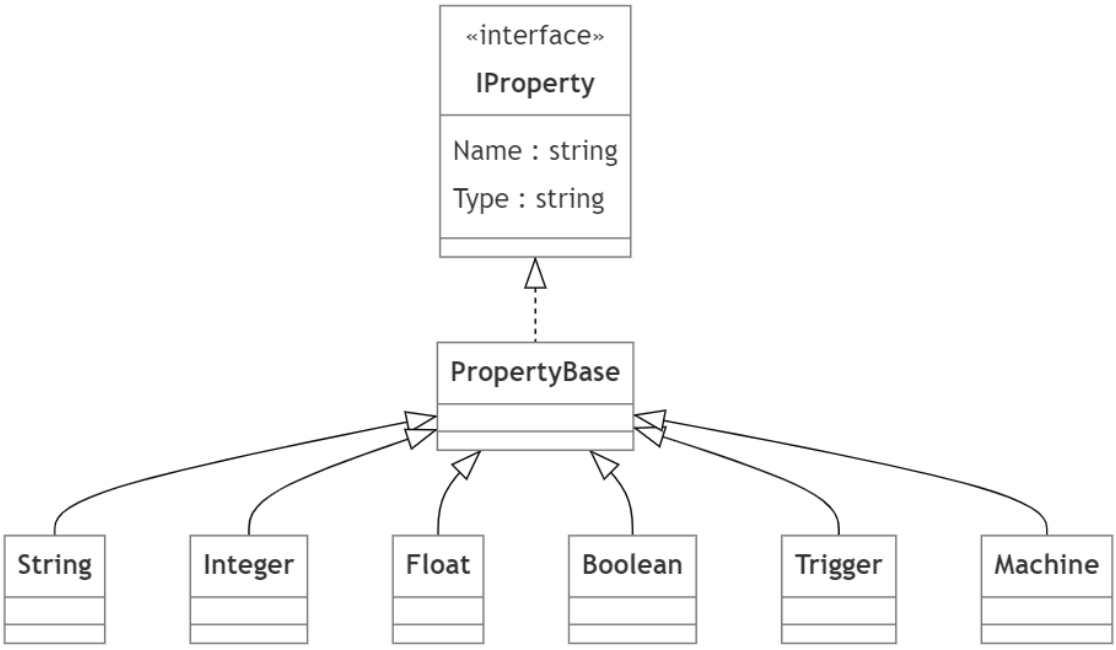


Fig. 4.9. Implementations of *IProperty* in the definitions package

How exactly the implementations of the logical model classes are instantiated depends on the chosen deserialisation approach. E.g., the entire model could be read from a single project file, there could be a single file per component, or the definitions could be read from the database.

4.3.2 Simulation Model

Similarly to the main interface in the definitions model, the main interface of the HPS simulation model is *IMachine*. The *machine* has *properties* that represent an observable part of its state.

An object, implementing *IMachine*, should be owned by the object, implementing *IContainer*. This relation is represented by the association between *IMachine* and *IContainer*.

Machines are hosted and simulated within the Environment. Machines backreference the environment in their Environment property, which is read-only. Therefore the machine can only belong to one environment.

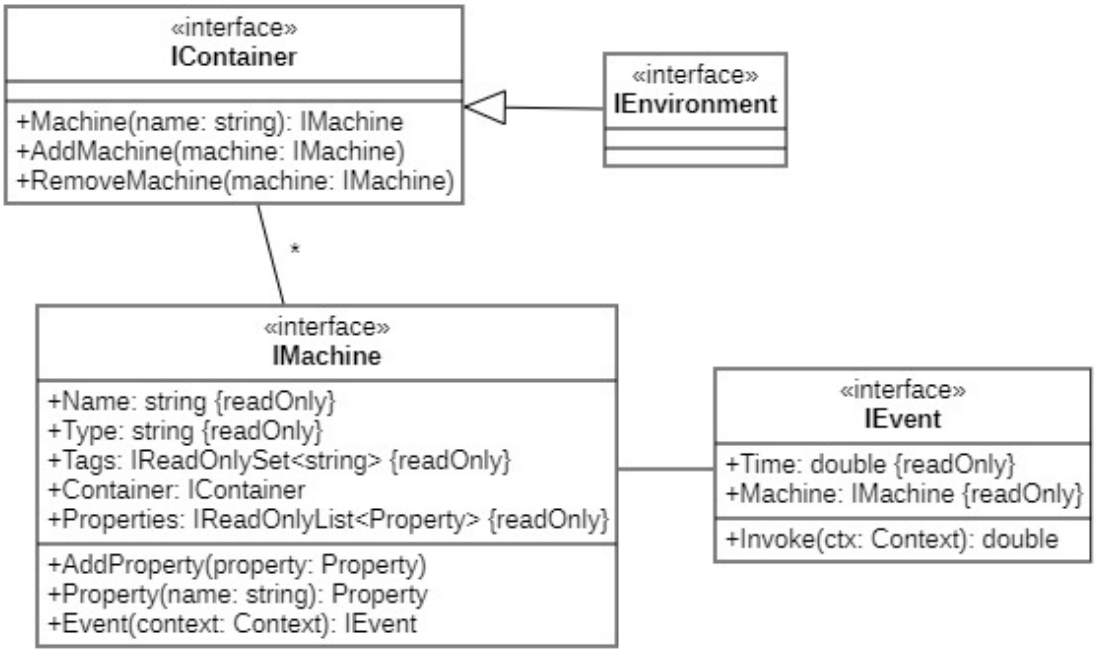


Figure 4.10: Main HPS interfaces.

HPS provides two implementations of the IMachine interface: State Machine and Network.

State Machine is a modelling abstraction that represents a component of the system as a set of states and transitions between them. When the probability of transitioning from one state to another depends only on the current state, the probabilistic state machine is a Markov state machine.

The state machine can be specified in the "structure" section of the machine definition using the following HPS DML structure:

```
"structure": {
  "states": [ "state1", "state2", ... ],
  "initial": "state1",
  "transitions": {
    "state1": {
      "state2": [ {
        "type": "probabilistic",
        "distribution": "exponential",
        "parameter": 0.1
      } ]
    },
    ...
  }
}
```

The states of the machine should be specified in the “states” property of the definition, with the initial state defined in the “initial” property. Transitions between states are defined in the “transitions” property, and the transition has an associated triggering function.

The triggering function defines the rule for invoking the transition from one state to another. Transition can be invoked by several triggers. The next transition is identified by enumerating all the transitions from the current state in the order in which they are defined, selecting the transitions that have the event time closest to the current moment and then selecting the first of them.

The required attributes for the trigger are “type” and “comment”. “type” specifies the type of the trigger and can be one of the supported trigger types. “comment” contains text associated with the trigger. Other trigger attributes depend on the type of trigger.

Network is a container for the machines. It is constructed by defining the machine instances. In agent-based modelling terms, it is the environment where the agents operate. The special type of machine, the network machine, is constructed by wrapping the network into a machine, forming a high-level agent, consisting of low-level agents. The top-level network in the simulation session is called Environment.

The internal structure of the machine is a concern of the HPS Engine. The language defines two types of machines: network - to support hierarchical composition, and state machine - to implement probabilistic and deterministic aspects of the systems.

Project

Project is a root element of the model. It contains project information properties, e.g. Title, and collection of the model components.

JSON Example:

```
{ "title": "Power Grid",
  "description" : "The model for analysing resilience of the power grid.",
  "components" : { ... } }
```

Machine

The machine represents the actor of the scenario. It encapsulates the observable state and behaviour associated with the acting element. The observable state is defined as machine properties. The behaviour is specified by the type of machine. The machine is identified by its unique name.

The property definitions are shared by all the machine instances, but the values of these properties are specific to the instance.

The machine's implementation can either be implemented in a plugin and injected into the engine or one of the supported machine kinds can be used. While creating the instance of the machine, the engine gets the initialisation parameters from the "structure" property of the machine. The format of this property should be supported by the engine. Data format of this property can inherit the data format of the document.

```
"machines": [
  {
    "name": "Substation",
    "type": "state-machine",
    "properties": { ... },
    "structure": { ... }
  },
  ...
]
```

Network

The network is defined by specifying its name, machine instances and their properties. All required properties without default values should be set while defining the instance of the machine.

```
"networks": [
  {
    "name": "Substations",
    "machines": [
      {
        "name": "G1",
        "machine": "Substation G1",
        "properties" : { "power": 1000 }
      },
      ...
    ]
  }, ... ]
```

Properties

The properties of the machine represent a part of the machine state that is visible to engine components. During simulation, the components can read the properties and react to changes in their values.

Property is identified by its name. The name should be unique among other property names, which are associated with the machine. The set of property values can be restricted by specifying the property type in the definition. The Required property specifies if the property value should be set.

When the machine is instantiated, the property is created with its default value or empty value. If the property is required, then the value should be specified while defining the network instance.

```
"machines": [
  {
    "name": "Substation",
    "properties": {
      "load": { "type": "Number", "required": true }
    },
    ...

"networks": [
  {
    "name": "Substations",
    "machines": [
      {
        "name": "G1",
        "machine": "Substation",
        "properties" : { "load": 1000 }
      },
      ...
```

Property values are parsed by the HPS engine and then passed to the machine properties in the run time. The property value data format can extend the data format of the document.

The simulation begins with the environment starting its machines. While running, the machines enqueue their events into the timeline, and then the environment dequeues and runs them. If there are several events scheduled for the same time, they are dequeued in order of enqueueing, i.e. FIFO.

4.4 Implementation

The implementation of the methodology provides the following tools and applications: 1) the simulation engine and its infrastructure (e.g., managing server and scheduler, computational agents, and command line tools) and 2) the Web-based model editor for developing models and orchestrating simulations.

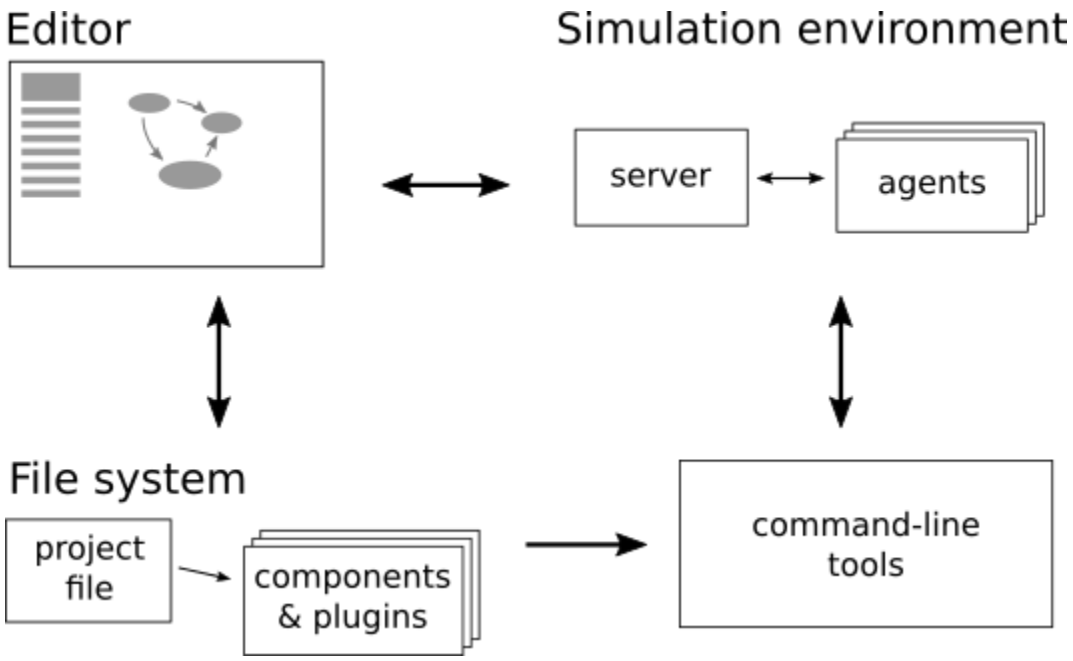


Figure 4.11: Conceptual diagram of the main HPS components.

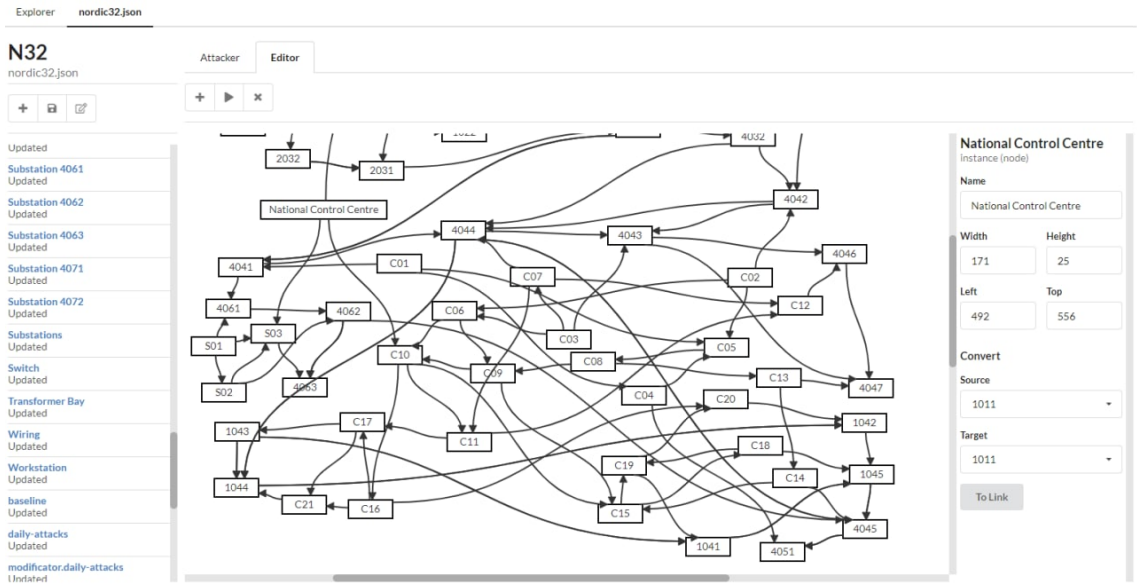


Figure 4.12: Part of the Nordic32 study project in the HPS editor.

The previously defined modelling approach introduced the model elements: machine, network, trigger, and others. The implementation of the model can be different, depending on usage. In the current implementation, it is different in the simulation engine and the editor. The simulation engine has two models: the design model, which contains definitions of the components, and the simulation time model, which contains instances of these components. The simulation engine models are designed to be efficient and consistent. The editor's model is based on the abstract model definition, but it has different requirements. It is designed to be easily modifiable. Differently from the simulation engine model, the editor's model can be inconsistent, e.g. contains cross-references and partially configured components.

The plugin API of the simulation engine provides interfaces for including programmatic behaviours into the model. It may be necessary when some particular aspect of the system can not be captured by using existing triggers, actions, and machines or when it is impractical to do so. For example, the technological process of the oil refinery can be implemented as an application that calculates critical state by simulating physical processes.

While simulating the model, the distribution server reads tasks and distributes them to agents. While the simulation runs, the agent collects log messages and reported measurements and sends them to the server. Server aggregates simulations from agents and builds a report.

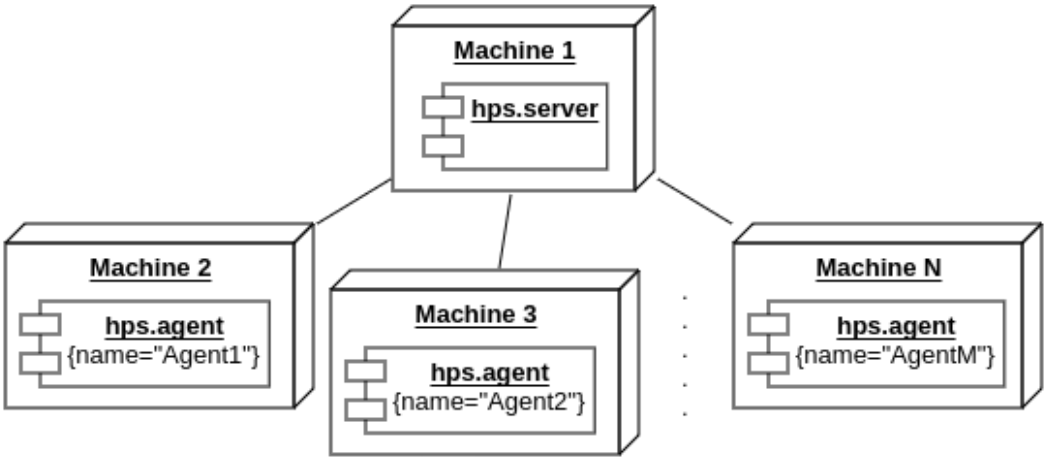
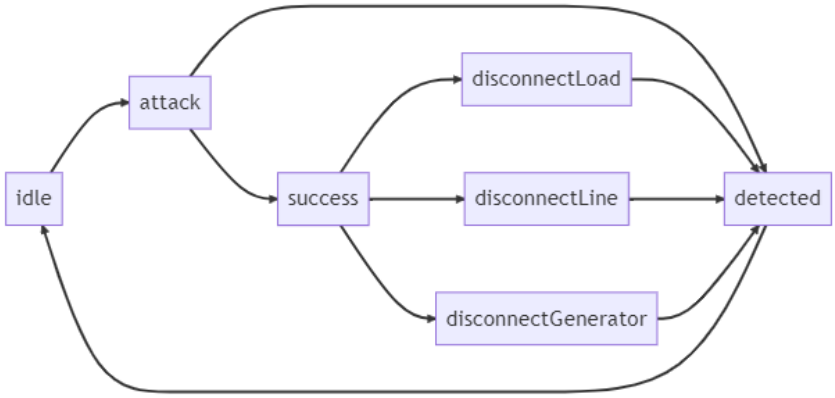


Figure 4.13: Deployment diagram of HPS server and agents.

Where serialisation is necessary, e.g. persisting the editor's model or sending the simulation model from the HPS server to the agent, the JSON file format is used. The structure of the serialised models is designed to be human-readable. Thus it is possible to create, maintain, and run models by editing model files directly and running command line tools.




```
{ "name": "Attacker",
  "type": "state-machine",
  "properties": [
    { "name": "frequency", "type": "Trigger" },
    { "name": "component", "type": "Machine" } ],
  "states": [
    { "name": "idle" },
    { "name": "attack", "enter": [ ... ] },
    { "name": "success", "enter": [ ... ] },
    { "name": "disconnectLoad", "enter": [
        { "action": "get machine by tag",
          "tag": "has-load", "property": "machine" } ] },
    { "name": "disconnectLine", "enter": [ ... ] },
    { "name": "disconnectGenerator", "enter": [ ... ] },
    { "name": "detected" } ],
  "initial": "idle",
  "transitions": {
    "idle": { "attack": [ { "type": "property", "property": "frequency" } ] },
    "attack": {
      "success": [ { "type": "gate" } ],
      "detected": [ { "type": "gate" } ] },
    "success": {
      "disconnectLoad": [
        { "type": "probabilistic",
          "distribution": { "type": "exponential", "lambda": 52560 } } ],
      "disconnectLine": [ ... ],
      "disconnectGenerator": [ ... ] },
    "disconnectLoad": { "detected": [ ... ] },
    "disconnectLine": { "detected": [ ... ] },
    "disconnectGenerator": { "detected": [ ... ] },
    "detected": { "idle": [ ... ] } } }
```

Figure 4.14: Example of state machine definition in JSON. The sample state machine is the Attacker from the Nordic32 case study.

The model editor provides a visual and editable representation of the model components, an interface for running simulations, and basic dataset

visualisation reports for data collected during simulation runs. The simulation results can be exported for more advanced data analysis or charting.

The editor extends the HPS model with diagram-specific data, e.g. instance size and position on the diagram, whether the instance represents a link or node on the network graph.

The selection of an optimal technology for the HPS engine requires a multifaceted assessment. This includes, but is not limited to, performance, error-proneness, interoperability, maintainability, potential for optimisations.

While researching the best technology for HPS engine implementation, it was implemented in several programming languages: Go, C++, JavaScript, and .NET. Based on the analysis of these implementations, it has been observed that each of these technologies possesses certain strengths and weaknesses.

Go is not an OOP language per see, it does not contain hierarchical types. However, its other features, such as interfaces and references allow writing programs in OOP style, which is more than enough to implement the HPS engine. The concept of property implemented as a pair of getter and setter is not available in Go. Instead, methods should be used directly to mimic property-like getters and setters. The concept of events is not implemented in the language either. Instead, first-class functions and arrays can be used to implement the concept of multicast delegates.

Go's approach to error handling is different from other modern languages. While many languages use exceptions and exception-capturing approaches, Go's approach to handling errors is to return the status while calling the operation.

```
value, err := action()
// when action() completes, either value or err is set, not both
```

While Go offers a favourable blend of code readability and efficient runtime, its lack of robust language constructs such as generics, modularity, properties, inheritance, and exceptions can present challenges when it comes to modifying and extending models. The absence of generics, for example, may require developers to write more boilerplate code to achieve similar functionality, which can increase the likelihood of errors and make maintenance more challenging. The limited modularity and absence of

inheritance can also make it harder to write reusable code that can work with different types and extend functionality as requirements change.

As anticipated, the C++ implementation of the language exhibits the highest speed performance. Nevertheless, due to the language's complexity and challenges in achieving cross-platform compatibility, modellers may encounter significant difficulties when working with C++ code. The intricacies of C++ make it most challenging for developers to write and maintain code. Consequently, while C++ may offer strong performance benefits, its use may be less practical for applications that prioritise ease of development and cross-platform compatibility.

Modern JavaScript stands out as a high-performance scripting language in comparison to other languages in the same category. Its ability to interact with libraries enables developers to delegate intensive computations to more performant languages such as C++. However, these interactions have been found to be inefficient, as a significant number of calls to native libraries can diminish the performance benefits of such delegation. While JavaScript is renowned for its rapid prototyping and ease of small modifications, ensuring the stability of JavaScript code necessitates a significantly larger automated testing coverage due to the language's dynamic nature. Overall, while JavaScript's performance and library interactions offer many benefits, developers must carefully consider these factors when deciding whether to use the language for their particular application.

According to the analysis, .NET technology exhibits the most advantageous combination of strength-contributing factors. Its broad range of programming language features and paradigms, combined with the ability to execute compiled code on various platforms, including WebAssembly, make it a highly versatile and adaptable platform. Additionally, .NET provides robust integration and extensibility technologies that further enhance its flexibility and usefulness. Moreover, its UI libraries simplify the process of creating editors and integrating the engine with them.

While C++ is a lower-level language that can offer superior performance, it is possible to achieve performance levels comparable to C++ in C#. For this, developers can leverage various optimization techniques, such as unsafe code blocks and pointer manipulations, using value types, ref parameters, memory

pools, and structures to minimise unnecessary memory allocations and reduce garbage collections.

From a cloud computing perspective, utilising one of the technologies supported by the service providers offers the most streamlined integration. Currently, the market leaders in cloud computing are Amazon, Microsoft, and Google. These providers, namely AWS, Azure, and Google Cloud, offer similar autoscaling solutions for running functions in a fully-managed serverless environment, which is the optimal approach for running HPS simulations.

Among the reviewed technologies, C# and JavaScript are natively supported on all of these platforms. This allows for more efficient development and deployment of applications utilising these programming languages. Overall, the choice of cloud provider and technology will depend on the specific requirements and needs of the project, but opting for a supported technology from a leading cloud provider can offer significant benefits in terms of integration and scalability.

Based on the findings of this comparison analysis, the technology that has been selected for the project is .NET 7. The editor is a browser application created with Blazor, a modern web application development framework that allows using C# for web development. The editor's interface is built with Semantic UI. The state machines and network editors are built with jointJS, a versatile and advanced diagramming framework.

Some studies demonstrated that it is possible to achieve similar performance by heavily optimising the code. .NET is used in many performance-critical applications, including machine learning, finance, and game development. .NET's extensibility allows integration with other languages and technologies (e.g. CUDA). Where necessary, such libraries can be integrated into the HPS simulation engine through its plugin API.

The HPS simulation engine communicates with the agents and the editor through gRPC, a modern high-performance Remote Procedure Call (RPC) framework. It connects the editor with the server and the server with the HPS agents.

HPS Engine (HPSE) runs simulations of the models defined in HPS Modelling Language (HPSML). It extends the HPSML with the supported property and machine types. HPSE defines a runtime simulation environment,

how the simulation is performed, how the state machine is implemented, and rules for hierarchical instantiation of the state and network machines.

The runtime model is defined in terms of OOP, e.g. interfaces, classes, attributes, and objects. Where appropriate, property means HPSML machine property and attribute means the attribute of the class/object of the runtime implementation.

Types

IMachine is an interface that corresponds to the machine instance, defined in HPSML. There are two types of machines defined in HPSML that are implemented in HPSE: network machine and state machine. Machine has the attribute Name and method Event().

Event defines action that machine is intended to perform at some moment in time (possibly, the current moment in case of instant event). During simulation, the environment requests all the machines for their planned events, executes the most recent one, and then repeats until simulation stops.

Environment is a container for the machines. The main loop works over the environment.

Property is a data field associated with the machine. It is defined in the machine's definition and its value can be defined in machine instance definition in the network.

Events

The events mechanism is a way for one runtime entity to attach a callback, which is executed by another runtime entity in response to some action. For example, CPU interrupts is an event system. Some high-level programming languages provide language-level support for events. Most of the modern languages have features that enable event-driven programming. The essential features are an ability for a variable to store function pointers and a way to call the function by its pointer. These features are well supported by the C programming language. Modern languages have features that greatly simplify the interoperability of the code fragments: closures and anonymous functions.

There are the following events defined in the HPSE:

- Simulation events: starting iteration, ending iteration.

- Properties: property changed.
- State machine: entering a state, leaving a state, changing.

Simulation Loop

Simulation loop iterates over the model events and invokes them. Its prerequisite is the constructed environment with all machines instantiated.

The conceptual steps are the following:

1. Get the most recent event by querying all the machines.
2. If there is no event, exit from the loop with the status “completed by idle”
3. Update current time.
4. If one of the limits has been reached (number of events, simulation time, clock time) or continue simulation predicate returns false, end the simulation with corresponding status.
5. Notify subscribers of the starting iteration event.
6. Invoke the event' action
7. Notify subscribers of the ending iteration event.
8. Jump to 1.

Observing Model Properties

For the model designer the main point of interest is how the model behaves when a series of interesting events occurs. For that the model state is needed to be observed while simulation runs. The model state means simulation time, property values, instances of the machines, their current state and other machine-specific attributes.

The engine provides two approaches for observing the model state:

- **Tracing model changes** can be used while analysing model dependencies and chains of cascading events. In this case all model's changes are tracked and logged and then these logs either analysed manually or processed by the analytic tools. This method is very useful while validating and debugging.
- **Subscribing to events** is a preferred way to observe the model state that can be used for already validated models, when the simulation time is reasonably small and when the number of model changes is large, so

tracing model changes is impractical. This method is much faster and produces much less data about the model.

Running HPS

Running the compiled HPS executable without arguments logs error and suggests to use `hpscmd -h' to get the more information on how to run the executable:

```
$ hpscmd
... Model file is not specified.
    Run 'hpscmd -h' for the details.
```

Running the executable with -h prints the supported command line arguments with descriptions.

The HPS repository contains a few test models in the folder “models”. To run any of them add the “-file” command line argument followed by the relative or absolute path to the model.

```
$ hpscmd -file models/test.json
```

The test model creates a simple state machine with two states “ok” and “fail” and probabilistic transitions between them. Without the specified limits for duration, simulation environment time and number of events the simulation runs for 10 events or for 1 second, whatever comes first.

The random number generator seed is initialised by default from the current datetime. This option can be overwritten by specifying the “-seed” argument. So calling the simulation with the same file, seed and limit produces exactly the same results.

Output Interpretation

A normal run of the simulation executable “hpscmd” with command line arguments uniquely identifying the network generates simulation events. These events occur as a result of state machine transition activation, property change, or custom message produced by engine plugin(s).

With the default settings these events are logged to the stdout as JSON messages, one message per line. The message includes simulation time and payload, which can be either change happening in the model, or custom message. In case of multiple changes all of them are recorded at the same time and they appear in log in the order of recording.

Sample log (long lines are truncated for readability):

```
{"time":0,"message":
 "starting default with seed 1487207223497346...
{"time":0.03557550243171388,
 "delta":{"machines":{"test1":{"state":"fail"}}}}
{"time":0.736920916774142,
 "delta":{"machines":{"test1":{"state":"ok"}}}}
...
{"time":3.0751483156879686,
 "message":"10 events in 64.183...
```

Only model changes are recorded, as the original state of the model is set by the model file and network.

The output can be saved to file or redirected to another process using the standard operating system commands, such as redirection or pipe.

Examples:

```
$ bin/hpscmd -file models/test.json > log.jslog
$ bin/hpscmd -file models/test.json | grep test1
```

4.5 Extensibility

Modelling a large system as a set of interconnected semi-Markov state machines may not be enough to capture all the relevant factors affecting the system's behaviour. For example, accurate simulation of the power transmission network requires calculating load flow through the power lines, a physical simulation technique involving solving a set of non-linear algebraic equations (or even a system of differential equations).

The HPS Engine model provides several extension points where plugins can be integrated and thus extend the model capabilities without modifying the core code. The extension points are implemented as observables (also known as events) [54]. In order to receive notifications, a consumer subscribes to the producer. At a time of an event, the producer distributes the notification to the consumers. In order to stop receiving notifications, the consumer unsubscribes from a producer. The order and concurrency of notifications received by multiple subscribers to the event are indeterminable and not expected to remain consistent across multiple notifications. Notification processing is synchronous and completes only when all subscribers complete invocations of their methods.

The HPS Engine provides the following events:

- **Initialisation**. This event occurs while the engine is initialised on the agent. The model is not initialised at this moment. Typical tasks for this event include adding new types of machines, triggers, distributions, and actions and initialising the plugin's resources, e.g. caches, repositories, and data connections.
- **Environment.OnSimulationBegins**. Occurs before starting the simulation, the environment is populated with the necessary instances. At this event, the plugin can acquire resources specific to the single simulation and the particular model. This is a suitable event to subscribe to the events of the instances.
- **Environment.OnIterationBegins**. Occurs when the next environmental event is about to be processed. All calculations related to the previous event are done at this moment. Typical tasks for this event are recalculating reward functions, physical models, or logging. Any changes in the event queue will not affect the currently processed event.
- **Environment.OnIterationEnds**. It occurs after processing the environmental event. Typical tasks for this event are recalculating reward functions, physical models, or logging. This is a suitable event to change the model and event queue. The following environmental event will be picked up from the queue after this event.
- **StateMachine.Changed.** Subscribers to this event are notified after the state machine enters a new state. Typical tasks for this event are modifying the simulation model, recalculating physical models (e.g. load flows in power network), and logging.
- **StateMachine.State.Entering.** Subscribers to this event are notified when the state machine enters a new state but before the state machine's Changed event. Typical tasks for this event are modifying the simulation model, recalculating physical models, and logging.
- **StateMachine.State.Leaving**. Subscribers to this event are notified when the state machine leaves a state. Typical tasks for this event are modifying the simulation model, recalculating physical models, and logging.

- **Property.Changed**. Subscribers to this event are notified immediately after the machine's property changes its value. Typical tasks for this event are modifying the simulation model, recalculating physical models, and logging.

The Stochastic Associations approach, introduced in PIA, is a way to capture and model dependencies between components. More specifically, a stochastic association defines how a probabilistic change in one component's state changes the probabilistic behaviour of another. For example, a stochastic association between components **A** and **B** can increase the failure rate of **B** 10-fold when **A** is in a failed state. Stochastic associations can be defined as a table, with columns "State Machine", "State", "Affected State Machine", "Transition", "Parameter", and "Value".

In HPS support for stochastic associations is implemented as a plugin. The Stochastic Associations plugin reads the table of stochastic associations in CSV format, initialises the network, and attaches actions to the state machine's state events. These actions monitor the change in the machines' state and modify the parameter of the transition trigger in the affected state machines.

5. Applications

5.1 Overview

The application of a methodology is crucial in verifying its usefulness, gathering feedback, and establishing a foundation for further improvements. This chapter presents two such applications of the HPS modelling methodology.

The first application demonstrates how the HPS modelling methodology supports the model's construction, simulation, collecting and aggregating of the simulation's results [52]. The second case study demonstrates how an existing HPS model can be used to investigate the benefits of enhancing the system's reliability using the defence-in-depth approach [50].

An improved Nordic32, the power production and retransmission network with controlling infrastructure and detailed design of the electrical substations, was used as a foundation for the first case study. This model was developed by the FP7 AFTER project research team (grant agreement number 261788).

Although the model of the network represents the physical system only to some extent, it allows a much wider range of experiments and observations than the real physical system tolerates. Also, analysing the results of the simulations with various counter-agents (attackers and protectors) gives a base for further advances by providing instruments, scenarios and data.

The focus of the first study is to apply the modelling methodology to build a set of reusable components, such as generators, loads, attackers, and others, build networks by combining them and investigate effects caused by the different types of attackers on the network.

The focus of the second study is to demonstrate the reusability of the model in other research studies.

5.2 Nordic32 Case Study

The problem of identifying the effect of cyber-attacks is important and complex. It is important because the information infrastructure becomes crucial for the successful operation of the power networks. It is complex because there are many types of attacks, from simple worm-like distortions to sophisticated and targeted attacks that are very hard to generalise.

Substation elements are low-level construction elements that represent substation devices identified by the standard IEC 61850.

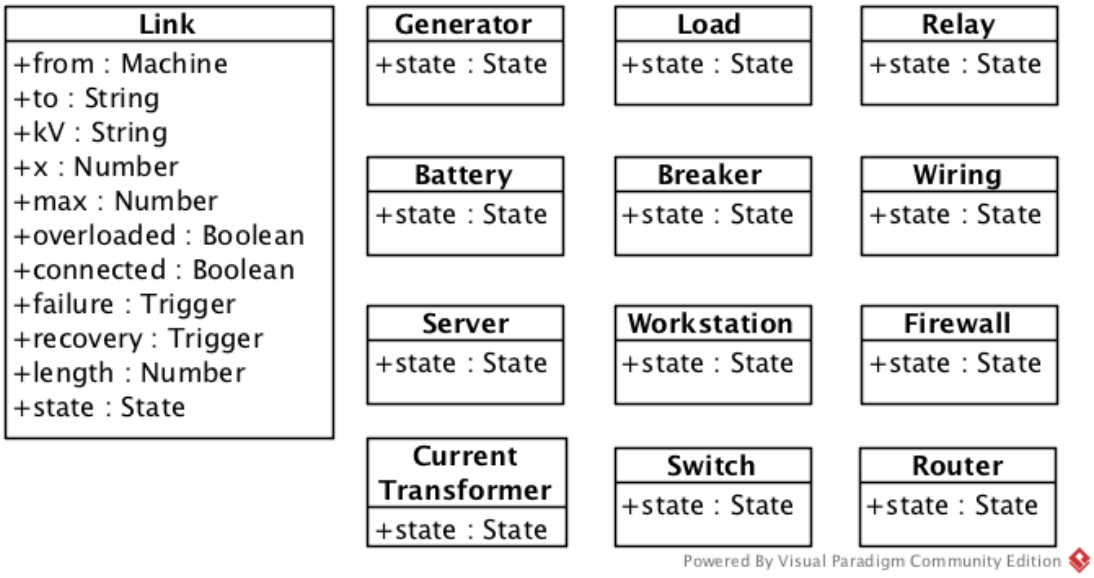


Figure 5.1: Substation Elements.

Link

The Link machine represents physical wiring between two substations. Its state represents the physical availability of the wire, so "ok" means that two substations can be connected, "fail" represents physical damage (ice, disaster, stolen cable, etc). Failure rate and recovery time are different for the links and depend on the link's length and reachability.

Link's properties:

from - incident substation.

to - another incident substation.

kV - voltage level

x - impedance

cf - maximum power allowed through the line

overloaded - flag showing whether the power going through the line is exceeded the "max" value

connected - flag specifying whether the line is connected to the substation's busbar

failure - activator of the "ok" to "fail" transition

recovery - activator of the "fail" to "ok" transition

length - physical length of the line

Generator

The Generator represents the power generator, connected to the substation. It is implemented as a state machine with two states "ok" and "fail". The failure rate and recovery time are equal for all generator instances. The Generator is a part of the Generator Bay.

Load

The Load represents the power consumers, served by the substation. It is implemented as a state machine with two states "ok" and "fail". The failure rate and recovery time are equal for all Load machines. The Load is a part of the Load Bay.

Breaker

The Breaker represents the line tripping device that disconnects the link when it becomes overloaded. The device disconnects the line instantly, if it is in the operational state. There are two such devices, connected to every line from both its ends.

It is implemented as a state machine with two states "ok" and "fail". The failure rate and recovery time are equal for all breakers. The Breaker is a part of the Line Bay and Transformer Bay.

Relay

The Relay represents the line disconnection device, differently from the Breaker, which disconnects the line only when the line is overloaded, the Relay reacts on the commands from the operator or control centre.

It is implemented as a state machine with two states "ok" and "fail". The failure rate and recovery time are equal for all relays.

There are two relays in the Line Bays on both sides of the line.

Current Transformer

The Current Transformer represents a device required to connect power networks with different currents. It has its own characteristics of time to failure and recovery.

It is implemented as a state machine with two states "ok" and "fail". The failure rate and recovery time are equal for all transformers. The Current Transformer is a part of the Transformer Bay.

Battery

The Battery represents an emergency power provider to the substation infrastructure.

It is implemented as a state machine with two states "ok" and "fail". It is a part of all bays.

Wiring

The Wiring represents connectivity between the substation components.

It is implemented as a state machine with two states "ok" and "fail". It is a part of all bays.

Switch

The Switch represents a device providing communication service to the bay components. It is responsible for routing packets between them.

It is implemented as a state machine with two states "ok" and "fail". It is a part of all bays.

Workstation

The Workstation represents an operator console. When the console is working, the operator can configure substation components.

It is implemented as a state machine with two states "ok" and "fail". It is a part of the Control Bay.

Server

The Server represents a substation telemetry data storage and processor.

It is implemented as a state machine with two states "ok" and "fail". It is a part of the Control Bay.

Router

The Router represents a substation connectivity device that connects it with the data centres. Its traffic is protected by the Firewall.

It is implemented as a state machine with two states "ok" and "fail". It is a part of the Control Bay.

Firewall

The Firewall represents a traffic filtering device that resides between the incoming connection to the substation and the Router. All the incoming requests to the substation devices should bypass the Firewall.

It is implemented as a state machine with two states "ok" and "fail". It is a part of the Control Bay.

IT Infrastructure Elements

Information Technology elements represent the construction blocks for building a connectivity network between the control centres and substations..

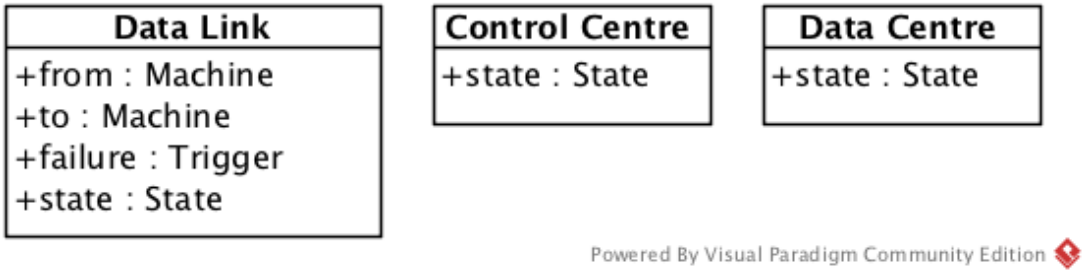


Figure 5.2: IT Infrastructure Elements.

Data Centre

The Data Centre is an intermediate data storage and transfer facility that participates in commands and data transfer between the substation and control centre. The Data Centre can be connected to any number of other Data Centres, Substations, or Control Centres. All network's Data Centres form a fault-tolerant transmission network. The Data Centre should not be able to tamper, re-send or interfere in any other way with the power network data channels.

It is implemented as a state machine with two states "ok" and "fail".

Control Centre

The Control Centre is a network management facility. It receives data from the substations and sends commands back to them. The Control Centres are connected with the Substations through the network of the Data Centres.

It is implemented as a state machine with two states "ok" and "fail".

Data Link

The Data Link represents connectivity between the Data Centres and other components of the network: the Substations, the Data Centres, and the Control Centres.

It is implemented as a state machine with two states "ok" and "fail".

Substation Bays

Bays are constructed by combining several substation elements into a network. They represent the substation parts as defined in standard [ISO 61850].

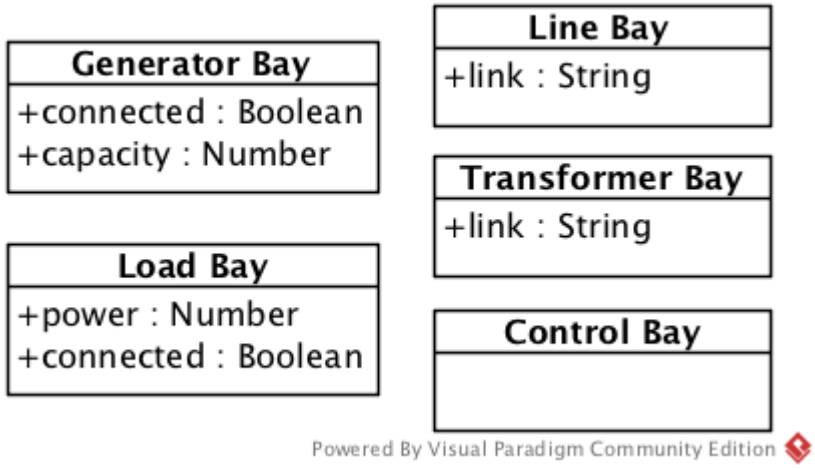


Figure 5.3: Substation Bays.

Generator Bay

The Generator Bay represents the connectivity between generator and substation bus bar.

The generator bay's properties:

capacity - capacity of the connected generator;

connected - specifies whether the generator is connected.

Line Bay

The Line Bay represents the connectivity between the link and substation bus bar.

The line bay's properties:

line - name of the link connected to the line bay.

Transformer Bay

The Transformer bay represents the connectivity between the link and substation bus bar through the current transformer.

The line bay's properties:

line - name of the link connected to the transformer.

Load Bay

The Load Bay represents the connectivity between the consumers supplied through the substation and the substation's busbar.

The load bay's properties:

power - total power required by all the consumers;

connected - specifies whether consumers are connected.

Control Bay

The Control Bay contains all the IT devices of the substation: workstations, servers, router, firewall, switch.

The object diagram below demonstrates how the substation state machines are placed on the "Substations" network. Each instance corresponds to the network defined in the model. The links on the diagram correspond to the Link state machines added on the network.

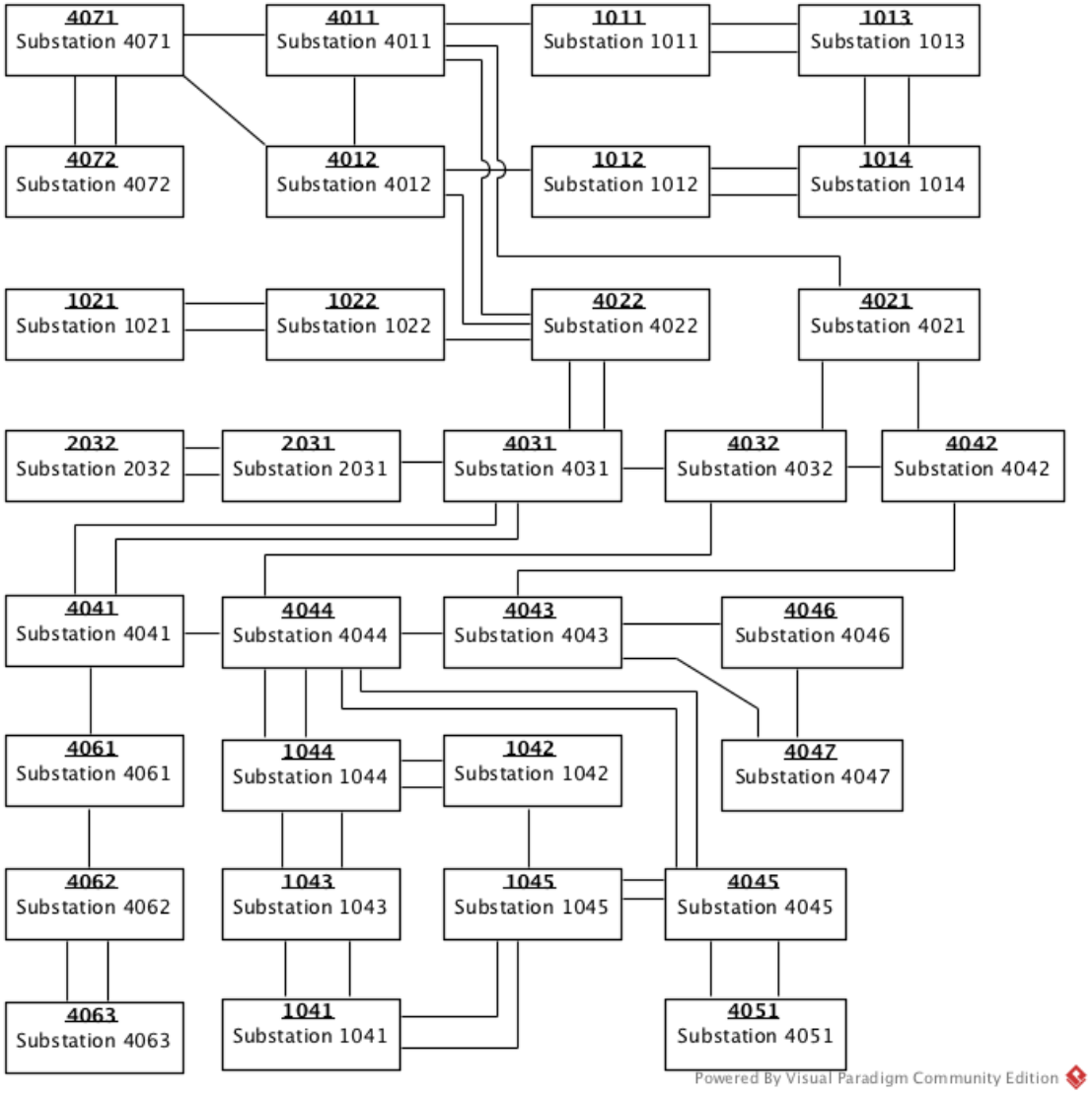


Figure 5.4: Substations.

5.3 Performance of Power Flow Calculation

Solving the Power Flow problem is an important part of the Nordic32 simulation model. The flow through the lines should be recalculated every time when the system's topology, generation, or load is changed. Performance of many other agents (control, attacker, statistics) depend on how efficiently the power flow solver works.

As DC power flow calculation is mainly matrix multiplications, using the existing optimised linear algebra library is an obvious choice. The standard software library interface for numerical linear algebra is BLAS [55]. Eigen is a

highly optimised C/C++ implementation of BLAS [56]. GONUM library set contains BLAS-compatible linear algebra library [57]. Although comparing C/C++ and Go solutions might be enough for the research, this group of technologies compared the combination of high-level scripting language with power flow solver implemented a) entirely in scripting language, and b) as a C/C++ extension using Eigen library. The node.js JavaScript was chosen for that purpose as it is one of the most popular modern scripting languages.

The target power network is defined as an undirected graph. Programmatically, the network data structures are similar to an incident list.

IDL definition of the network data structure

```
enum NodeType { Producer, Consumer }

interface INode {
  attribute NodeType Type;
  attribute float Power;
  attribute float Capacity;
}

interface IEdge {
  attribute int From;
  attribute int To;
  attribute float Reactance;
}

interface INetwork {
  attribute List<INode> Nodes;
  attribute List<IEdge> Edges;
}
```

Pseudocode of obtaining branch flows for balanced network:

Calculating branch flows, F

```
      n, M, N = getNetworkData()

      // construct matrices and vectors from network data
      Ba = matrix M x N
      Bm = matrix N x N
      Bl = vector N
```

```
for i in 1..M do
  from, to, x = n.links(i)
  Ba(i, from) = 1/x
  Ba(i, to) = -1/x
  Bm(from, to) += -1/x
  Bm(to, from) += -1/x
  Bl(from) += 1/x
  Bl(to) += 1/x

for i in 1..N do
  Bm(i, i) = Bl(i)

P = n.nodes(2..N).map(x -> x.Power)

// calculate flows
F = Ba * ([0] + Bm(2..N, 2..N)^-1 * P)
```

To compare the load flow performance the following networks where selected:

- Nordic32 network [58] - network of 32 substations representing the Swedish electrical network;
- IRRIIS network [59] - network of 52 electrical substations in central Italy;
- idealised tree networks [60] - symmetrical tree-like networks with 94, 190, and 384 nodes.

Table 5.1: Characteristics of the networks

Name	**Nodes**	**Links**	**Generators**	**Loads**
Nordic32	32	60	17	11
Rome IRRIIS	52	67	3	46
Grid 94	94	93	12	82
Grid 190	190	189	12	178
Grid 382	382	381	12	370

The following implementations of the load flow solver were compared:

- Go with GONUM library
- C++ with Eigen library
- node.js plugin with C++/GONUM solver

- JavaScript-only implementation

The correctness of the solvers was validated by doing cross-checks (also referred often as "back-to-back" testing) between the implementations and comparing them with examples [61].

The test networks are generated once and used during benchmarking by the particular implementation. The benchmarking network set contains 1000 Nordic32 networks, 1000 Rome IRRIIS, 1000 Grid 94, 300 Grid 190 and 100 Grid 382 networks. The networks are created by randomly changing network loads by up to 5% from their original values and rebalancing.

In order to assure that the average problem solving time is not affected by the random fluctuations, the total number of test cases for every network is divided into ten groups and an average is calculated for each of these groups. The execution time for the single batch is selected to be much larger than timer resolution.

The hardware, compiler and compilation options greatly affect the performance of the generated binaries. It is important to mention that the tests were performed on Debian Linux 64-bit, C++ version is compiled with GCC 4.7.2-5 on second level of optimization (-O2) and the architectural target is x64. The tests were run on the virtual machine, using a single core of Xeon E5-2680 CPU.

The table below contains average load flow solving time calculated for every combination of technology and network. The standard deviation (σ) is calculated across ten batches of the networks processed by the solver.

The results of the benchmarking shows that the GONUM implementation works slower on smaller networks (less than 94 nodes and less), but faster on larger networks (382 nodes and more) with median about Grid 190 (network with 190 nodes). Unsurprisingly, JavaScript implementation is slowest.

Table 5.2: Benchmark results

Name	C++/Eigen (ms;σ)	Go/GONUM (ms;σ)	JS/C++/Eigen (ms;σ)	JS/numeric (ms;σ)
Nordic32	0.044;0.005	0.087;0.165	0.193;0.083	0.321;0.460
Rome IRRIIS	0.136;0.129	0.180;0.086	0.232;0.125	0.552;0.037
Grid 94	0.473;0.177	0.533;0.081	0.646;0.015	2.671;0.265
Grid 190	2.895;0.843	2.874;2.801	3.240;0.286	19.897;0.755
Grid 382	19.259;2.935	13.471;8.203	19.520;2.973	156.150;3.806

The Fig.5.5 visualises results from the table above, grouping averages by network and technology. The y-axis is scaled logarithmically as the difference between extreme cases is 5 orders of magnitude.

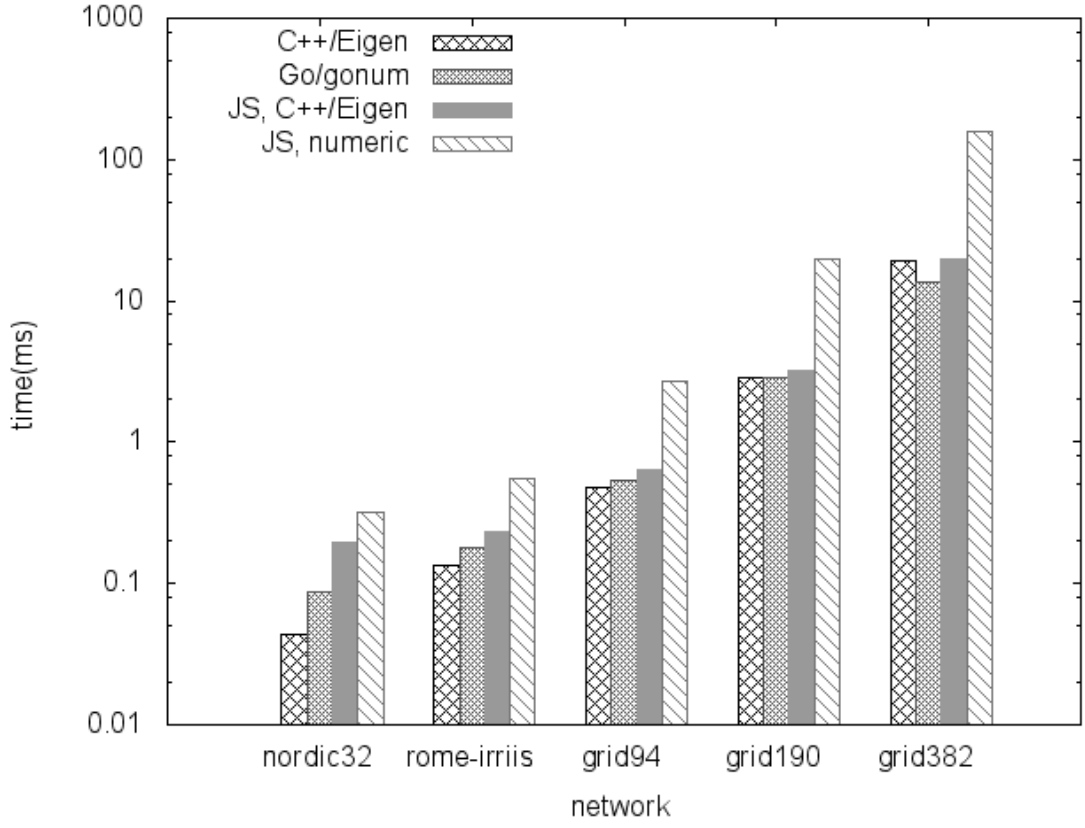


Figure 5.5: Time in milliseconds required to solve the load flow problem for the networks. Scale of the y-axis is logarithmic.

Many benchmarks show that C++ version of the algorithm is one of the fastest, having Go implementation of the load flow solver outperforming C++

version may be related to differences in algorithms used in Eigen and GONUM libraries.

However, the results show that the performance of the algorithm implementation in Go is comparable to the implementation in C++. Considering the other benefits it gives, such as simple integration with existing C/C++ libraries, garbage collector, closures, and parallelisation, it is a good choice to become a language for simulation models and data processing problems.

5.4 Assessing Resilience to Cyber-attacks

This section provides a synopsis and outcomes of the case studies conducted by the working group on the critical infrastructure modelling at CSR and published in the papers "Quantification of the Impact of Cyber Attack in Critical Infrastructures" [49] and in the chapter "Quantitative Evaluation of the Efficacy of Defence-in-Depth in Critical Infrastructures" of the book "Resilience of Cyber-Physical Systems" [50]. The references offer more information on the methods and techniques used in the research.

Assessing resilience to a threat is done by measuring the metrics of interest for the system under normal circumstances, then for the system under stress and then observing how the metrics are changing when the stress is removed. As the extended Nordic 32 model contains ICT elements, it is possible to investigate how the network is affected by the cyber-attacks.

The effect caused by the cyber-attacks can be observed by comparing the random variables, such as duration of load shedding or total delivered power, calculated when the system operates normally and under stress. The function that calculates the value of the random variable selected for the comparison is called "reward function". For this study the total delivered power for 10 years of operation was selected as the reward function.

The "normal" operation of the critical infrastructure implies operation with periodic accidental failures of the components. Such failures and recovery from them are captured by the transitioning from the "Ok" state to the "Fail" state and back in the state machines. This is a *base-line* model, i.e. it is without the adversaries (attackers).

The introduced attacker model corresponds to widely known cyber-attack concepts:

- Attacks are periodic, with exponentially distributed time between attacks.
- Attacks are performed through the elements of the IT infrastructure, in particular - firewalls.
- Success of an attack is probabilistic - most of the time it fails, but sometimes it succeeds.
- Attacks are detectable and the system eventually recovers from them, either by an automatic control function or manual intervention.

In addition to that, the following alternative behaviours of the attacking agents were studied:

- Selective versus random targeting on the substation firewalls.
- Immediately disconnecting the substation component or changing the configuration of the components.

The attacking agent is implemented as state machine of the following structure:

- States: idle, attack, firewallRule1, firewallRule2, firewallRule3, firewallRule4, firewallRule5, success, disconnectLoad, disconnectLine, disconnectGenerator, detected
- Initial state: idle.
- With some probability the attacker goes from "idle" to "attack".
- When attacking, the attacker randomly selects the firewall rule to try.
- The firewall rule either succeeds in stopping the attacker (it goes to "idle" then) or fails (attack succeeds).
- When it is in, the attacker performs the action, gets detected and goes to "idle". The action is either:
    - disconnecting the bay (generator, load, or link), or
    - changing the line overloading threshold.

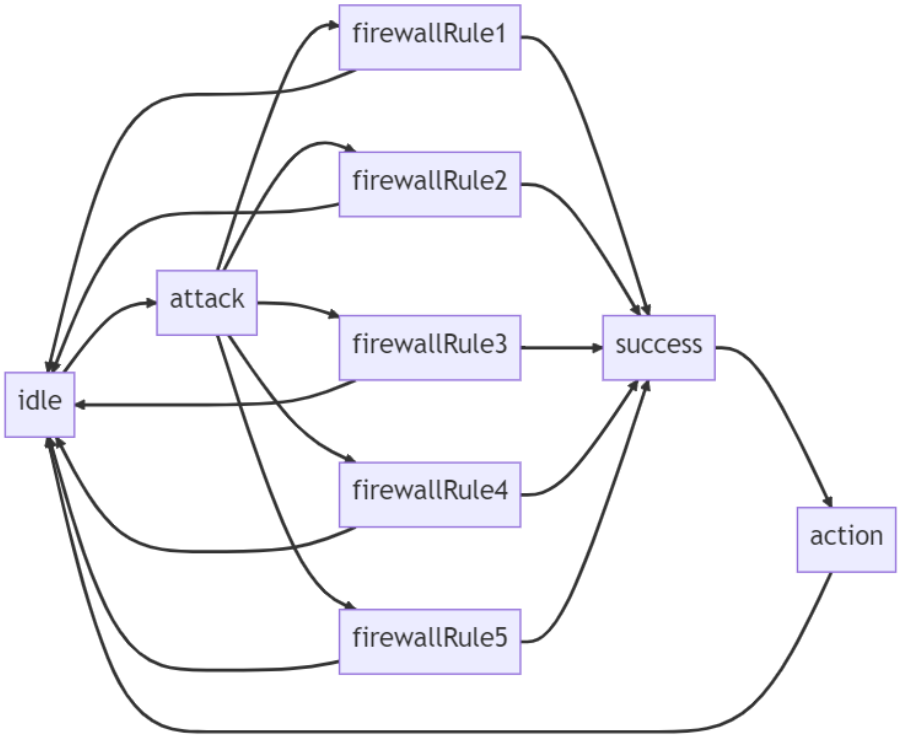


Fig. 5.6. State machine representing the attacker.

Disconnecting the bay changes the topology of the network, which causes recalculation of the load flow to links, shedding load if necessary, and, eventually, recovering from the disconnection by the "control" plugin.

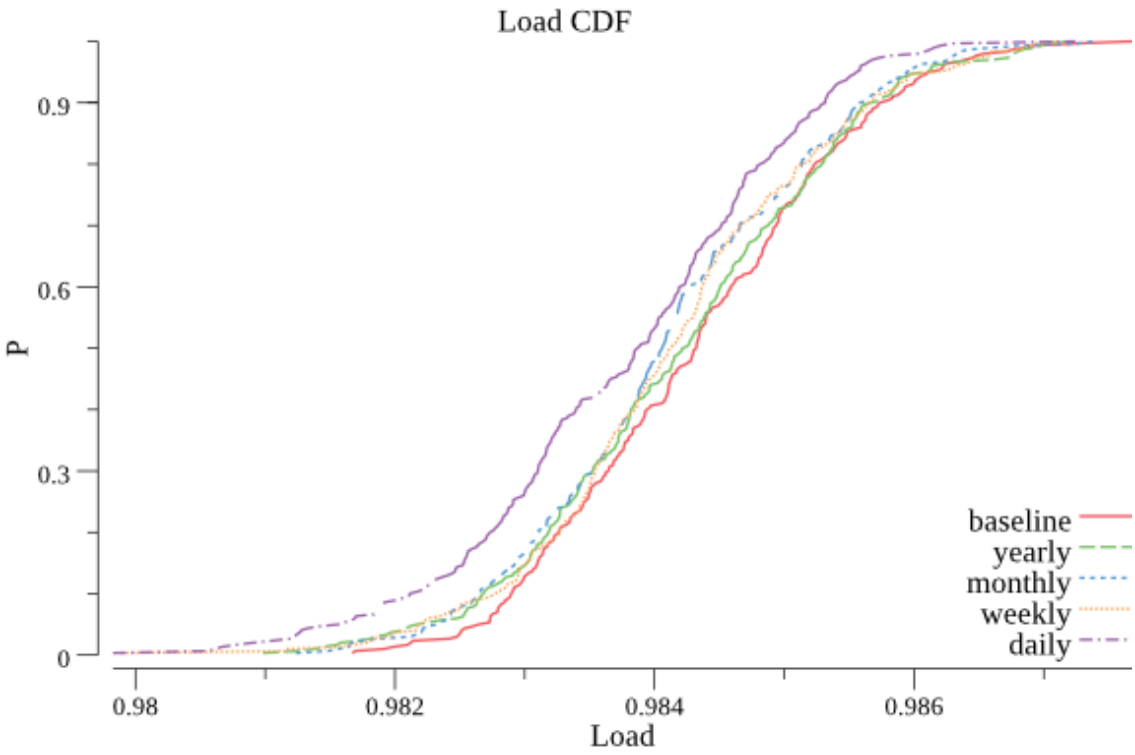
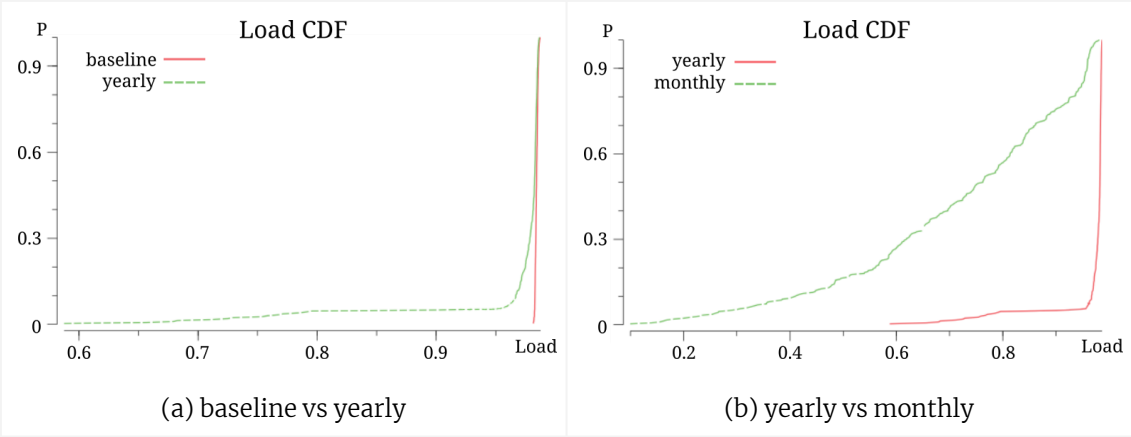


Figure 5.7: Distribution of the fraction of delivered power for baseline, yearly, monthly, weekly, and daily attacks.

The attacker that changes the configuration properties of the system has the same penetration algorithm, i.e. it attacks periodically through the firewall rules. Differently from the previously described attacker, instead of disconnecting the bay it changes the line overloading threshold to 110% of the current flow through the line. Changing the attack action has a significant result - even for the yearly attacks the effect of the stealthy configuration changes is worse than the effect from the simple attacker and the system quickly deteriorates when the frequency of the attacks is increasing.

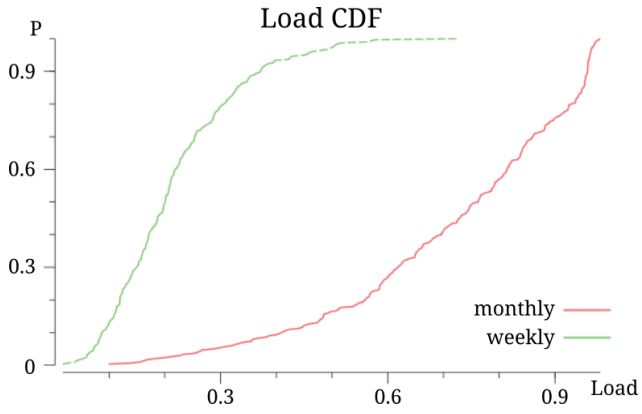
The subversive attacker counterparty is the "inspector" - the agent that resets the overloading thresholds. The inspector is implemented as a simple state machine with the states "idle" and "working". The "idle-working" transition is configurable, the "working-idle" is instantaneous, i.e. as soon as the state working is reached and the actions defined for that state are completed a transition will take place to the “idle” state. On entering the "working" state the machine enumerates all links and restores the thresholds.

CDFs of the supplied power are presented on Fig. 5.8 and 5.9.



(a) baseline vs yearly

(b) yearly vs monthly



(c) monthly vs weekly

Figure 5.8: Distribution of the fraction of delivered power for baseline, yearly, monthly, and weekly subversive attacks.

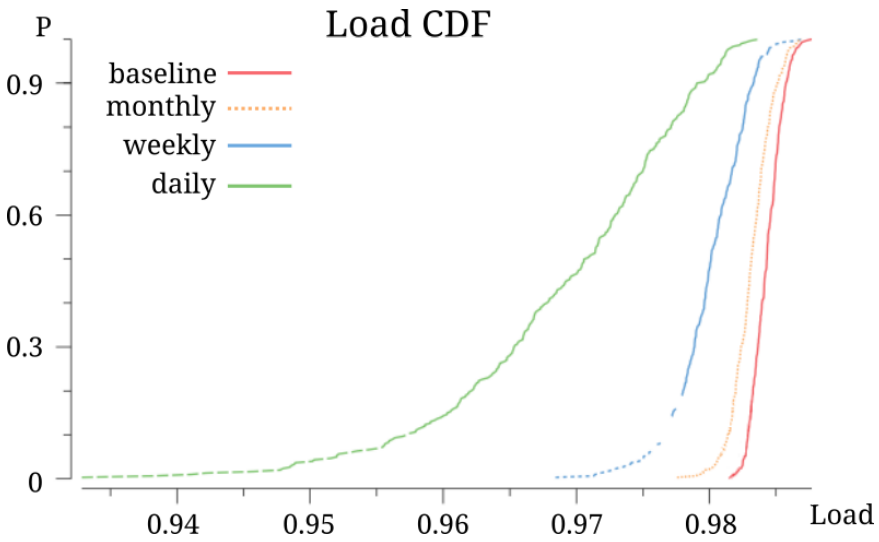


Figure 5.9: Distribution of the fraction of delivered power for baseline, weekly subversive attacks, monthly, weekly, and daily inspections.

Defence-in-depth, as defined by NIST, is an information security strategy that integrates people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organisation [62]. In terms of safety, the US NRC defines it as "creating multiple independent and redundant layers of defence to compensate for potential human and mechanical failures so that no single layer, no matter how robust, is exclusively relied upon" [63]. Effective placing of the controls requires assessing the effect the introduced controls have on the efficiency of the potential attacker, i.e. answering the questions like how long would it take for the attacker to get into the system. With the constructed models of system and attacker it is possible to answer such questions quantitatively.

For the Nordic 32 model the effect on introducing redundancy and diversity on the Breaker component of the network was investigated. The Breaker component was selected because it exists in all bays and its failure is immediately visible as it disconnects the component connected through the bay. The redundancy was added by replacing the Breaker component with the two Breaker Channel components. The Attacker in this study is the modified subversive attacker. On successful attack it transitions the breaker component into the compromised state, in which the component fails more often than in the normal state. The Inspector, in turn, transitions the breaker from compromised to normal state. Simulating the spread of the malicious agent within the network, several modes of acting were selected for the attacker, different on how many breakers and of which kind (single breaker, all line/generator/load breakers, or all breakers within the substation). Attacker's knowledge about two breaker components was modelled by analysing two different attack behaviours: either attacking both breaker components simultaneously or attacking them independently.

The following simulation campaigns were run for the network with the modified Breaker component and attackers:

- One Breaker Component: no attacks or attacks weekly either lines, loads, generators or any of those, either without inspections or with yearly or monthly inspections. 13 configurations in total.

- Two Breaker Components: no attacks or attacks weekly either lines, loads, generators or any of those, either without inspections or with yearly or monthly inspections. 25 configurations in total.

The main results of the simulation campaigns in both studies:

- when it is possible to build the models of the system and the malicious agents, the quantitative results can be obtained by running the simulation campaigns;
- the subversive attacker has significantly greater effect on the system, than the simple attacker;
- the inspections help reduce the effect of the attacks;
- the difference between single breaker component and two breaker components for the baseline models is negligible;
- introducing diversity of the breaker component reduces the effect of the attacks.

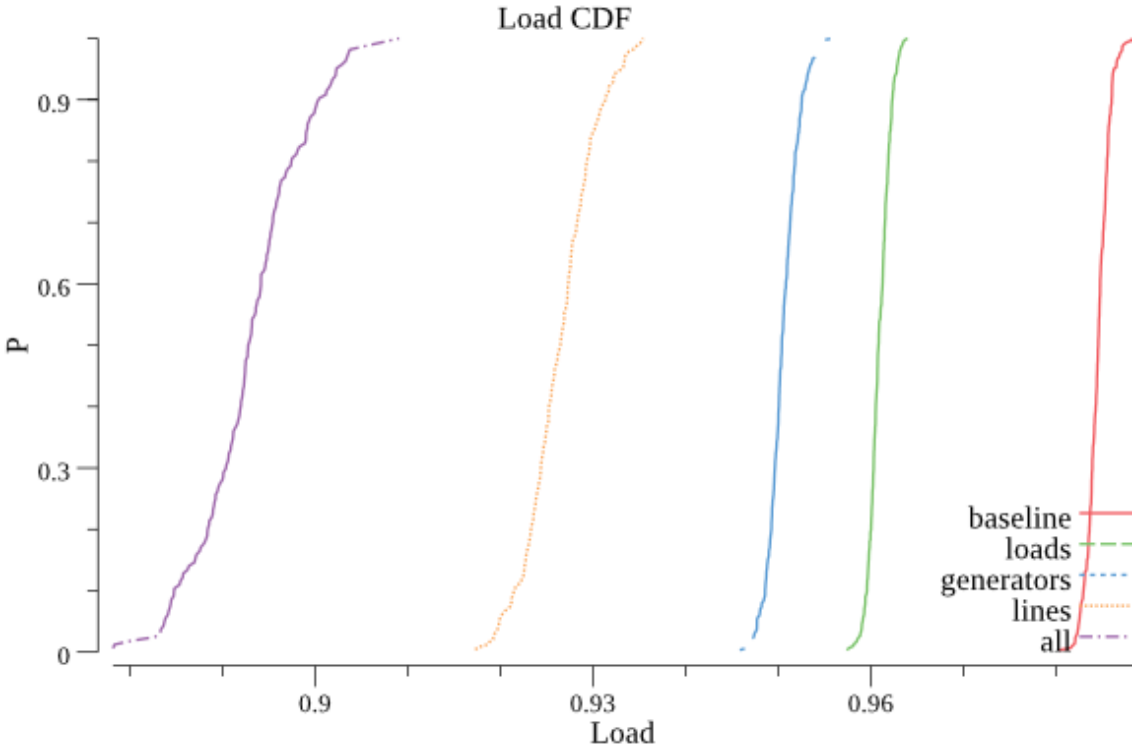


Figure 5.10: CDF of delivered power by different targets of attacks without inspections.

6. Conclusion

6.1. Summary

This thesis presents a new methodology for supporting the assessment of large cyber-physical systems, including critical infrastructures. The methodology offers a modelling approach and supports it with the new modelling language for defining hybrid hierarchical stochastic networks, the simulation engine to efficiently solve the models via Monte Carlo simulation, and an editor which allows analysts to build models *quickly*. The applications of this methodology were applied on non-trivial examples to demonstrate how the proposed methodology and tool support could support assessing the resilience of critical infrastructures.

The development of this new methodology was motivated by an interest in addressing issues discovered by applying other methodologies and tools for various case studies. While addressing these issues the following research questions were asked:

- Q1: What is the best approach for modelling large networks?
- Q2: What is the faster way to get the results of the simulation?
- Q3: Which editing features help the modeller the most?
- Q4: How do models and the results obtained with them support the existing assurance cases ?
- Q5: Are the constructed models reusable?

The proposed modelling methodology described in this thesis provides answers to the questions above.

Specifically, Chapter 3 addresses the questions related to supporting assessment methodologies and demonstrates how stochastic modelling can be used in CAE assessment, including the argumentation of substituting the system with the model and using the results of simulations as evidence. The results of this research are partially demonstrated in the works "Tool Support for Assurance Case Building Blocks" [64] and "Using Structured Assurance Case Approach to Analyse Security and Reliability of Critical Infrastructures" [52].

Chapter 4 provides solutions for modelling-related questions. E.g., it proposes hierarchical composition as the best way to model large multi-component systems, a plugin architecture for building hybrid models, and cloud-based HPC simulations for the best performance. The results of this research are published in the articles "Quantification of the impact of cyber attack in critical infrastructures" [49] and "Model-Based Evaluation of the Resilience of Critical Infrastructures under Cyber Attacks." [65].

Chapter 5 demonstrates the applicability of the methodology to use cases and the reusability of the constructed models. The research results from this chapter are published in the articles "Quantification of the impact of cyber attack in critical infrastructures" [49] and "Model-Based Evaluation of the Resilience of Critical Infrastructures under Cyber Attacks." [65] and in the chapter "Quantitative Evaluation of the Efficacy of Defence-in-Depth in Critical Infrastructures." of the book "Resilience of Cyber-Physical Systems" [50].

6.2. Future Work

Although the primary focus of the research was to support assessing the resilience of critical infrastructures, the proposed methodology is not limited to this one type of system and only reliability properties. It can be employed to assess other emerging properties and address uncertainties in any system that can be sufficiently represented by a stochastic model.

The hybrid modelling approach already provides support for integrating different kinds of models. However, it can be improved further by adding the concept of *actions*, explorable by a generic adversary. Such adversaries can actively examine the system for the shortest path that maximises their reward.

The simulation engine for running stochastic models can be further improved. The different optimisation techniques, including parallelisation, memoisation, adaptive scheduling, and applying hardware-specific optimisations can be further extended and incorporated into the engine.

This thesis explores how to support the CAE assurance case approach with stochastic modelling, a topic which recently has regained importance in the context of safety assurance of autonomous vehicles [48]. And of course, a

possible direction to further extend the proposed research is its alignment with other assurance methodologies.

One potential approach to enhancing the usability of the framework is to create a library of *reusable components*, models, and case studies. Such a library could offer significant benefits to practitioners and modellers involved in assurance cases in terms of reducing the time to adopt the methodology .

Appendix 1. Nordic32

Nordic32 is a model of the Swedish power transmission network [58]. The power production, transmission, and consumption elements of the model are power substations and transmission lines. The power substations are characterised by the amount of power produced on the substation (generation) and the amount of power transmitted by the substation to consumers (load).

The original Nordic32 model is a fully specified electrical network, which can be modelled with a physical model only. In the project AFTER this model was enhanced with an industrial distributed control system (IDCS) compliant with the international standard IEC 61850 "Communication networks and subsystems in substations" [66]. Later, in the project SESAMO, the network was extended even further by adding probabilistic parameterisation for failures and recoveries of the network elements [49].

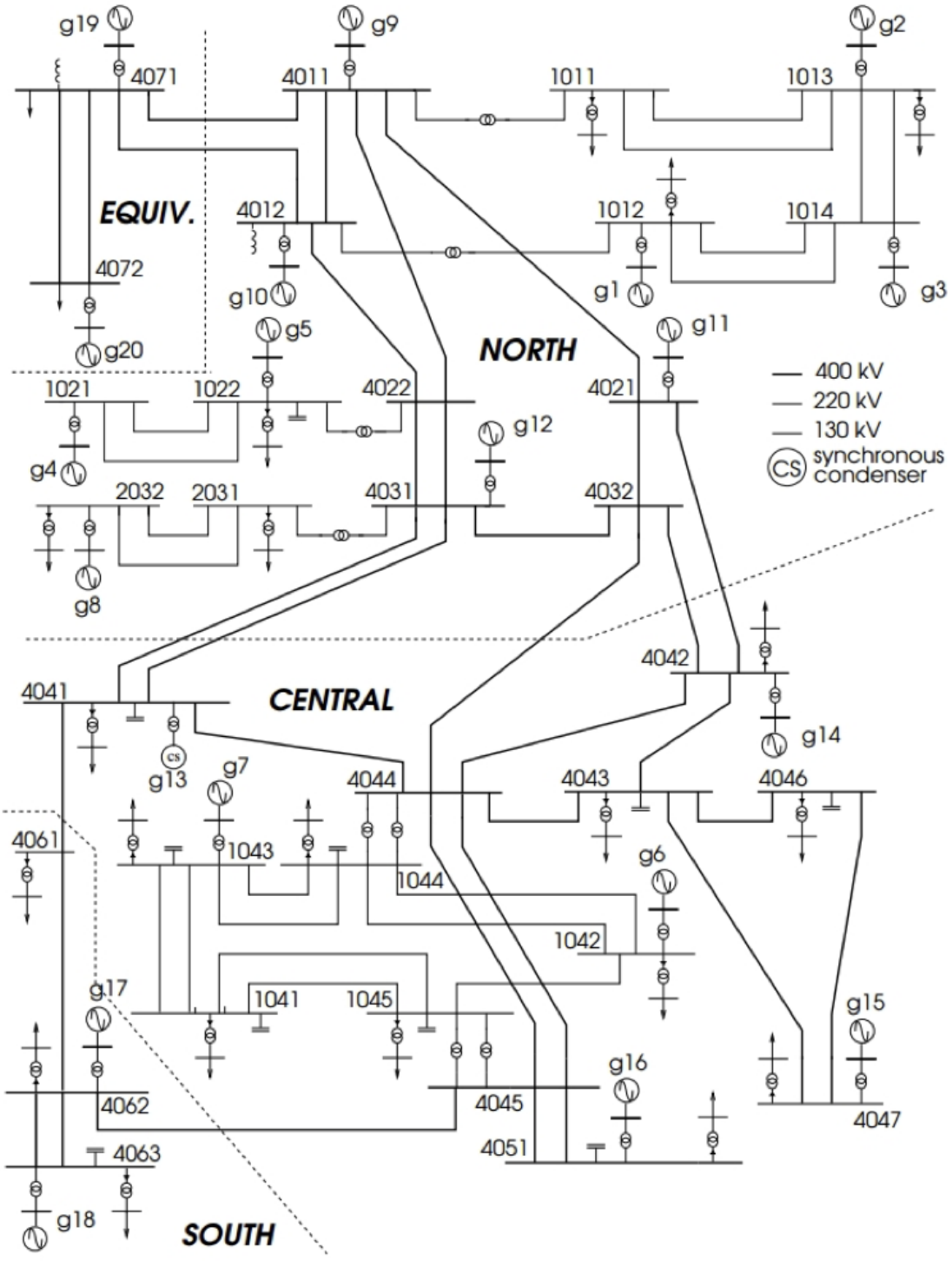


Figure A.1: Electrical components of Nordic32.

There are four groups of substations that corresponds to geographical regions:

- North - northern part of Sweden, high generation, low consumption;

- Central - central and southern parts of Sweden, moderate generation, high consumption;
- Southwest - Zealand island, low generation, high consumption;
- External - Finland, moderate generation, moderate consumption.

There are 32 substations in the network and 23 generators of different voltage levels 400, 200 and 130 kV. The first digit in the substation name corresponds to the voltage level. Substations with less than three connected lines are single busbar substations, with three or more lines connected are double busbar substations.

The communication network consists of DDCs, Northern, Central and South RCCs and ICC. The RCCs communicate with ICC in order to optimise production and consumption and reduce waste. All control centres have a fully redundant backup centre. Each substation is physically connected to one of the DDCs. Connectivity between the substation and its RCCs is maintained through the chain of communication links and DDCs. The whole communication network is "N-1" robust (failure of a single component does not interrupt connectivity if all other components operate normally).

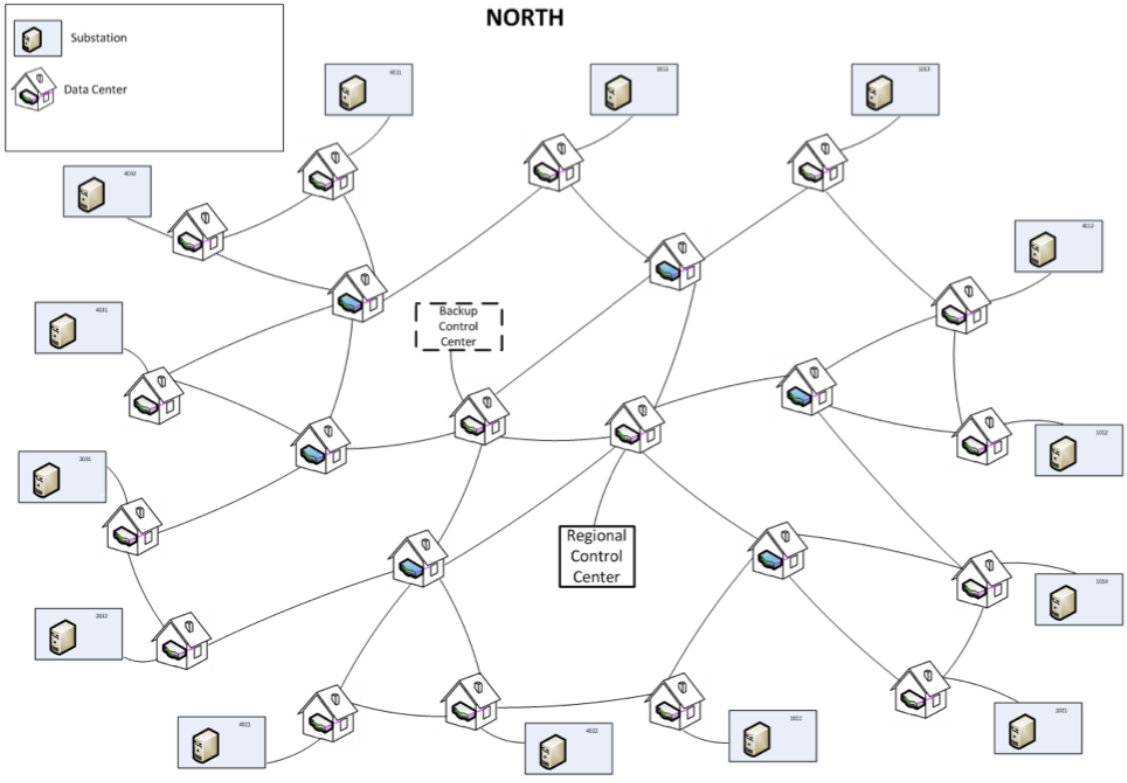


Figure A.2: The communication network of the North region.

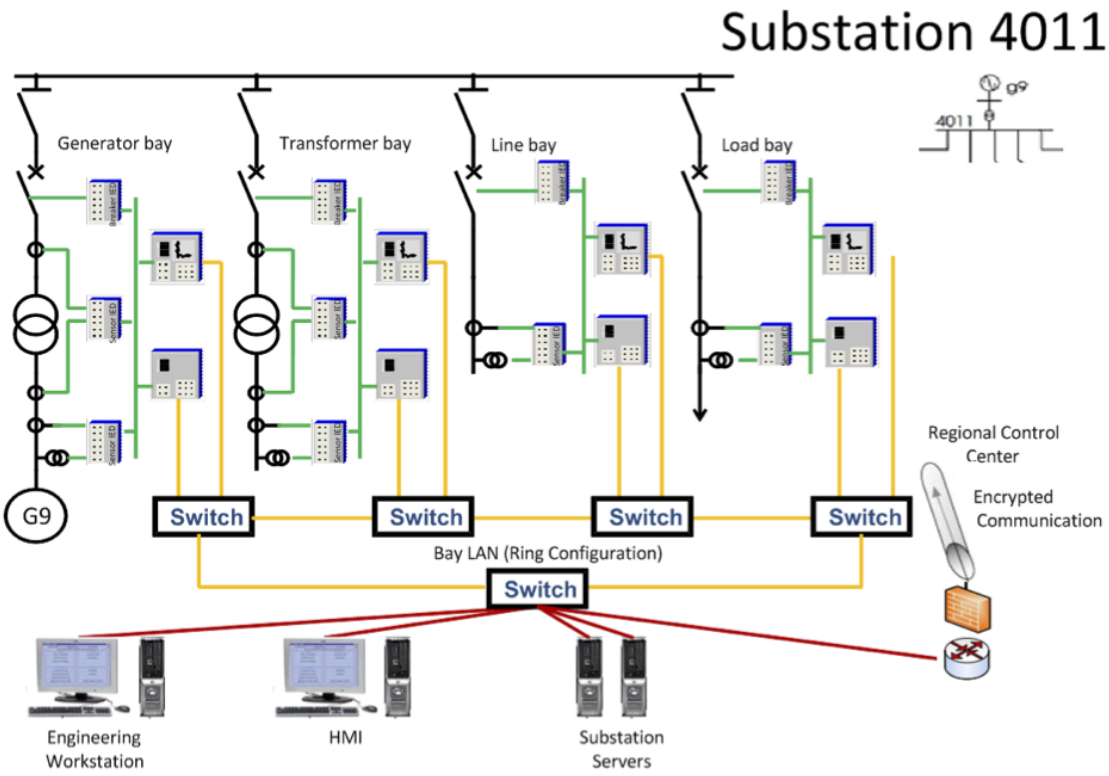


Figure A.3: Structure of the substation 4011.

The substation consists of a single busbar with bays of different types of bays connected to it. There are four bay types in the model: line, generator, transformer, and load. Every bay has a protection system of two protection devices. One of the protection devices also performs control functions. The protection (such as switching the line off in case of overload) is performed when one of the protected devices is on.

Although the physical model of the original Nordic32 model is detailed enough to perform power analysis in AC mode, such simulation is quite intense in terms of required computational resources. To simplify the load and keep the system behaviour close to the original, DC simulation can be used instead of AC simulation.

Solving the Direct Current Load Flow problem for the network of N nodes and M edges is equivalent to the following matrix multiplication [61]:

$$\theta = [\mathbf{B}^{-1}]\mathbf{P}; \mathbf{P_L} = (\mathbf{bA})\theta$$

where $\mathbf{P}$ - N-length vector of bus injections, $\mathbf{B}$ - $N \times N$ admittance matrix, Θ - N-length vector of bus voltage angles, $\mathbf{P_L}$ - M-length vector of branch flows, $\mathbf{b}$

- $M \times M$ diagonal matrix of branch susceptances, $\mathbf{A}$ - $M \times N$ bus-branch incidence matrix.

The network given to the load flow solver is the balanced network, i.e. total consumed power is equal to total generated power. For the purpose of this research the network balancer is just sets all the generators proportionally to their capacity:

$$p = \sum_{n \in N} l(n) \left(\sum_{n \in N} c(n) \right)^{-1}$$

where $l(n)$ is a power demand at the node n or 0 if the node is not a consumer and $c(n)$ is a generator capacity of the node n or 0 if the node is not a producer.

The power injections of generators then:

$$\mathbf{G}' = r\mathbf{G}$$

where $\mathbf{G}_c$ is a vector of generator capacities.

In the agent-based model of Nordic32, each element such as DDC, substation, busbar bay, or individual element is constructed as an agent. The agent is a combination of a Markov state machine and discrete event handlers [49].

Appendix 2. Preliminary Interdependency Analysis

This thesis appendix provides a concise description of the “Preliminary Interdependency Analysis” method, which is described in the report "Preliminary Interdependency Analysis (PIA): Method and tool support" by Bloomfield et al. (2010).

Preliminary Interdependency Analysis (PIA) is a scenario-driven process of examining, evaluating, and interpreting information about the system to discover and improve understanding of interdependencies between the system’s components and provide a justified basis for further modelling and analysis. Its objective is to develop a documented appropriate *service model* for the given system.

The PIA process is a continuous and cyclical activity of creation and refinement of interdependency models. In this process earlier stages of model development are revisited to refine assumptions and design decisions in response to resolving uncertainties and discrepancies discovered in later stages. The effect of modified assumptions and different design decisions propagate through stages of the PIA process and results in an improved model of the system.

PIA comprises two analytical approaches: qualitative and quantitative. Qualitative analysis objectives are to define the boundaries of the system, identify components and their interdependencies, and make proper assumptions about uncertainties. The goal of quantitative analysis is to simulate the model and interpret the results.

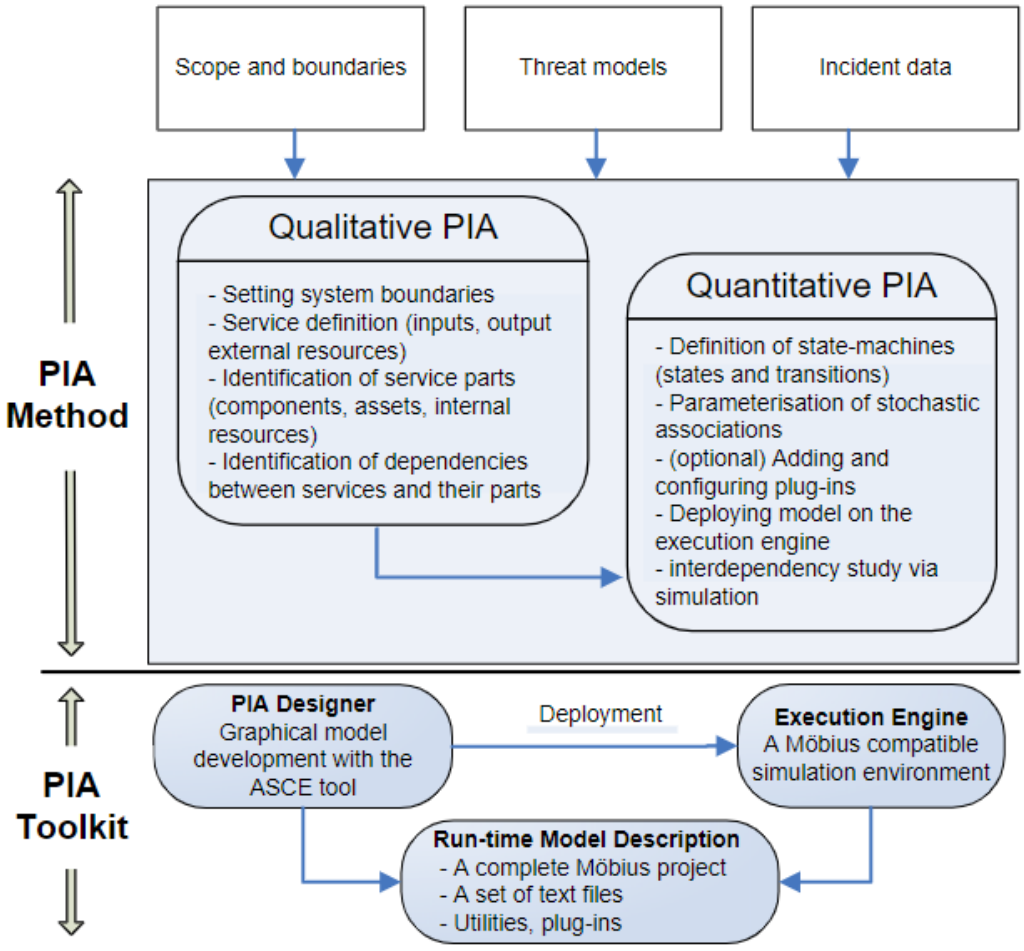


Fig. A.4. Overview of PIA method and toolkit [2].

PIA is supported by two tools: **PIA Designer** and **Execution engine**. The designer utilises an existing proprietary tool *ASCE* for visual representation and model navigation. The engine executes a model, developer with the designer, with *Möbius* [67].

PIA models are developed at two levels:

- **Service level**. The modelled system is represented by a set of interdependent services. This view is purposefully abstract, focusing only on the existence of dependencies, which are elicited from lower-level dependencies among each service's constituent entities, such as physical components and resources. These associations among components are referred to as *coupling points*, which are in a context of a component called *incoming* or *outgoing*.

- **Detailed service behaviour model** (DSBM). Individual services are implemented at this level. Implementation is supposed to be owned by the respective service operator, i.e. an organisation.

The PIA process comprises seven stages:

- Stage 1. Establishing system description and scenario context.
- Stage 2. Model development.
- Stage 3. DSVM model development.
- Stage 4. Initial dependency and interdependency identification.
- Stage 5. Probabilistic model development.
- Stage 6. Adding deterministic models of behaviour.
- State 7. Exploratory interdependency analysis.

During these stages, the following narrative information is relevant and useful:

- Scenarios.
- Incident description.
- Threat or attack model.
- Model of the threat agent.

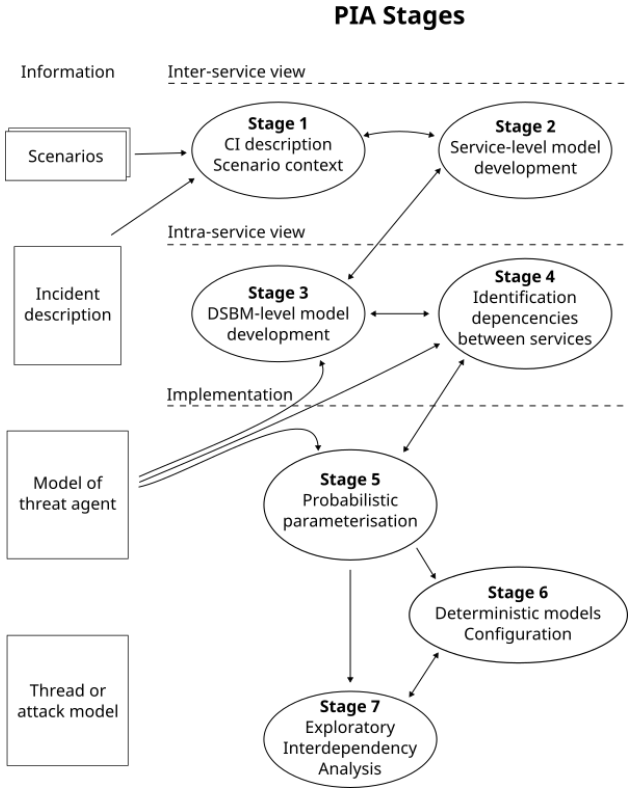


Fig. A.5. PIA method stages and associated information artefacts

References

1. SESAMO project web-site. In: SESAMO|Security and Safety Modelling [Internet]. Available: http://www.sesamo-project.eu/

2. Bloomfield RE, Chozos N, Popov PT, Stankovic V, Wright D, Howell-Morris R. Preliminary interdependency analysis (PIA): Method and tool support (D/501/12102/2 v2.0). Adelard LLP and City University London; 2010. Available: https://openaccess.city.ac.uk/id/eprint/3091/

3. Inspectorate NS, House S. A step by step guide on how to interpret each clause. 2016; 1–39. Available: http://www.nsi.org.uk/wp-content/uploads/2012/11/Annex-A-Step-by-Step-Guide-for-ISO-9001-2015-NG-FG-AG.pdf

4. The Council of the European Union. Council Directive 2008/114/EC of 8 December 2008 on the indentification and designation of European critical infrastructures and the assessment of the need to improve their protection. Official Journal of the European Union. 2008;345: 75–82.

5. Cabinet Office. Strategic Framework and Policy Statement on improving the Resilience of Critical Infrastructure to Disruption from Natural Hazards. London; 2010.

6. Critical National Infrastructure. In: National Protective Security Authority [Internet]. 20 Apr 2021 [cited 18 Jul 2023]. Available: https://www.npsa.gov.uk/critical-national-infrastructure-0

7. Rinaldi SM, Peerenboom JP, Kelly TK. Identifying, understanding, and analyzing critical infrastructure interdependencies. IEEE Control Syst Mag. 2001;21: 11–25. doi:10.1109/37.969131

8. Rosato V, Issacharoff L, Tiriticco F, Meloni S, Porcellinis SD, Setola R. Modelling interdependent infrastructures using interacting dynamical models. International Journal of Critical Infrastructures. 2008;4: 63. doi:10.1504/IJCIS.2008.016092

9. Box GEP. Science and Statistics. J Am Stat Assoc. 1976;71: 791. doi:10.2307/2286841

10. Pederson P, Dudenhoeffer D, Hartley S, Permann M. Critical Infrastructure Interdependency Modeling: A Survey of U.S. and International Research. Idaho National Laboratory. 2006;25: 27. doi:10.2172/911792

11. Eusgeld I, Henzi D, Kröger W. Comparative evaluation of modeling and simulation techniques for interdependent critical infrastructures. Scientific Report, Laboratory for Safety. 2008; 15–35. Available: https://bit.ly/2JdjcpF

12. Peterson JL. Petri Nets. ACM Comput Surv. 1977;9: 223–252. doi:10.1145/356698.356702

13. Molloy. Performance Analysis Using Stochastic Petri Nets. IEEE Trans Comput. 1982;C-31: 913–917. doi:10.1109/TC.1982.1676110

14. Ajmone Marsan M, Conte G, Balbo G. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Trans Comput Syst. 1984;2: 93–122. doi:10.1145/190.191

15. Ciardo G, Muppala J, Trivedi K. SPNP: stochastic Petri net package. Proceedings of the Third International Workshop on Petri Nets and Performance Models, PNPM89. IEEE Comput. Soc. Press; 2003. doi:10.1109/pnpm.1989.68548

16. Bonabeau E. Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences. 2002;99: 7280–7287. doi:10.1073/pnas.082080899

17. Luna F, Stefansson B. Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming. Springer Science & Business Media; 2000. doi:10.1007/978-1-4615-4641-2

18. Macal CM, North MJ. Introductory Tutorial: Agent-Based Modeling and Simulation. Proceedings of the 2011 Winter Simulation Conference, 11-14 Dec. IEEE Press; 2011. pp. 1456–1469. doi:10.1109/WSC.2011.6148117

19. Holling CS. Resilience and Stability of Ecological Systems. Annu Rev Ecol Syst. 1973;4: 1–23. doi:10.1146/annurev.es.04.110173.000245

20. Henry D, Emmanuel Ramirez-Marquez J. Generic metrics and quantitative approaches for system resilience as a function of time. Reliab Eng Syst Saf. 2012;99: 114–122. doi:10.1016/j.ress.2011.09.002

21. Attoh-Okine NO. Resilience Engineering: Models and Analysis. Cambridge University Press; 2016. Available: https://play.google.com/store/books/details?id=O_-lCwAAQBAJ

22. Selected current practices. ReSIST (Resilience for Survivability in IST) European Network of Excellence. 2009.

23. Cyber safety and resilience. Royal Academy of Engineering; 2018 Mar.

24. International Organization for Standardization. Security and resilience -- Organizational resilience -- Principles and attributes. 2017. Report No.: 22316.

25. Tierney K, Bruneau M. Conceptualizing and measuring resilience - A Key to Disaster Loss Reduction. TR News. 2007;250: 14–18. Available: http://onlinepubs.trb.org/onlinepubs/trnews/trnews250_p14-17.pdf

26. Strigini L. Resilience Assessment and Evaluation of Computing Systems. In: Wolter K, Avritzer A, Vieira M, van Moorsel A, editors. Resilience Assessment and Evaluation of Computing Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. pp. 3–24. doi:10.1007/978-3-642-29032-9

27. Resilience. [cited 22 Mar 2022]. Available: https://www.dhs.gov/topic/resilience

28. Cabinet Office. Keeping the Country Running: Natural Hazards and Infrastructure. Environment. 2011. p. 100. Available: http://www.cabinetoffice.gov.uk/resource-library/keeping-country-running-natural-hazards-and-infrastructure

29. Pimm SL. The Balance of Nature? Ecological Issues in the Conservation of Species and Communities. University of Chicago Press; 1991. p. 448.

30. Holling CS. Engineering Resilience versus Ecological Resilience. Engineering Within Ecological Constraints. 1996. pp. 31–44. doi:10.17226/4919

31. Bruneau M, Chang SE, Eguchi RT, Lee GC, O'Rourke TD, Reinhorn AM, et al. A Framework to Quantitatively Assess and Enhance the Seismic Resilience of Communities. Earthquake Spectra. 2003;19: 733–752. doi:10.1193/1.1623497

32. Adams TM, Bekkem KR, Toledo-Durán EJ. Freight Resilience Measures. J Transp Eng. 2012;138: 1403–1409. doi:10.1061/(ASCE)TE.1943-5436.0000415

33. Gluchshenko O, Foerster P. Performance based approach to investigate resilience and robustness of an ATM System. Tenth USA/Europe Air Traffic Management Research and Development Seminar (ATM2013). 2013; 7. Available: http://atmseminarus.org/seminarContent/seminar10/papers/277-Gluchshenko_0127130117-Final-Paper-4-8-13.pdf

34. Hollnagel E, Woods DW, Leveson N. Resilience Engineering: Concepts and Precepts. Ashgate Publishing, Ltd.; 2010.

35. Hager G, Wellein G. Introduction to High Performance Computing for Scientists and Engineers. CRC Press; 2010. doi:10.1201/EBK1439811924

36. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science and Engineering. 1998;5: 46–55. doi:10.1109/99.660313

37. The Computer Language Benchmarks Game. [cited 9 Feb 2018]. Available: http://benchmarksgame.alioth.debian.org/

38. Nanz S, Furia CA. A Comparative Study of Programming Languages in Rosetta Code. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE; 2015. pp. 778–788. doi:10.1109/ICSE.2015.90

39. Bik AJC, Girkar M, Grey PM, Tian X. Automatic intra-register vectorization for the Intel® architecture. Int J Parallel Program. 2002;30: 65–98.

40. Jones E, Oliphant T, Peterson P, Others. SciPy: Open source scientific tools for Python. [cited 9 Feb 2018]. Available: http://www.scipy.org/

41. The Swift Programming Language. [cited 9 Feb 2018]. Available: https://developer.apple.com/swift/

42. The Rust Programming Language. [cited 9 Feb 2018]. Available: https://www.rust-lang.org/

43. Meyerson J. The go programming language. IEEE Softw. 2014;31: 104–104. doi:10.1109/ms.2014.127

44. Toulmin SE. The Uses of Argument. Cambridge University Press; 1958.

45. Bloomfield R. E., Bishop P. G., Jones C. C. M., Froome P. K. D. ASCAD - Adelard safety case development manual. Adelard; 1998.

46. The Assurance Case Working Group. Goal Structuring Notation Community Standard Version 3. 2021. Available: https://scsc.uk/SCSC-141C

47. Bloomfield R, Netkachova K. Building Blocks for Assurance Cases. 2014 IEEE International Symposium on Software Reliability Engineering Workshops. 2014. pp. 186–191. doi:10.1109/ISSREW.2014.72

48. Bloomfield R, Rushby J. Assurance 2.0: A Manifesto. arXiv [cs.SE]. 2020. doi:10.48550/ARXIV.2004.10474

49. Netkachov O, Popov P, Salako K. Quantification of the impact of cyber attack in critical infrastructures. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer; 2014. pp. 316–327. doi:10.1007/978-3-319-10557-4_35

50. Netkachov O, Popov P, Salako K. Quantitative Evaluation of the Efficacy of Defence-in-Depth in Critical Infrastructures. In: Flammini F, editor. Resilience of Cyber-Physical Systems. Springer; 2019. pp. 89–121. doi:10.1007/978-3-319-95597-1_5

51. Bloomfield R, Bishop P. A Methodology for Safety Case Development. Safety-critical Systems Symposium 98. 1998.

52. Netkachova K, Bloomfield R, Popov P, Netkachov O. Using Structured Assurance Case Approach to Analyse Security and Reliability of Critical Infrastructures. In: Koornneef F, van Gulijk C, editors. Computer Safety, Reliability, and Security. Springer International Publishing; 2015. pp. 345–354. doi:10.1007/978-3-319-24249-1_30

53. State Chart XML (SCXML): State machine notation for control abstraction. W3C working draft. 2015. Available: https://www.w3.org/TR/scxml/

54. Gamma E, Johnson R, Helm R, . Johnson RE, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH; 1995. Available: https://play.google.com/store/books/details?id=tmNNfSkfTlcC

55. BLAS (Basic Linear Algebra Subprograms). Available:

http://www.netlib.org/blas/

56. Guennebaud G, Jacob B, Others. Eigen v3. 2010 [cited 2 Apr 2018]. Available: http://eigen.tuxfamily.org

57. Gonum Numerical Packages. [cited 9 Feb 2018]. Available: https://www.gonum.org/

58. Peppas D. Development and Analysis of Nordic32 Power System Model in PowerFactory. 2008; 77 pp.

59. Integrated Risk Reduction of Information-based Infrastructure Systems. [cited 9 Feb 2018]. Available: https://cordis.europa.eu/project/id/027568

60. Carreras BA, Lynch VE, Dobson I, Newman DE. Critical points and transitions in an electric power transmission model for cascading failure blackouts. Chaos. 2002;12: 985–994. doi:10.1063/1.1505810

61. Seifi H, Sepasian MS. Electric Power System Planning Issues, Algorithms and Solutions. Springer Science & Business Media; 2016. pp. 1055–1063. doi:10.1111/jce.13019

62. Joint Task Force Interagency Working Group. Security and privacy controls for information systems and organizations. National Institute of Standards and Technology; 2020 Sep. doi:10.6028/nist.sp.800-53r5

63. Drouin M, Wagner BJ, Lehner J, Mubayi V. Historical Review and Observations of Defense-in-depth. US Nuclear Regulatory Commission, Office of Nuclear Regulatory Research; 2016.

64. Netkachova K, Netkachov O, Bloomfield R. Tool Support for Assurance Case Building Blocks. Computer Safety, Reliability, and Security. Springer International Publishing; 2015. pp. 62–71. doi:10.1007/978-3-319-24249-1_6

65. Netkachov O, Popov P, Salako K. Model-based evaluation of the resilience of critical infrastructures under cyber attacks. Critical Information Infrastructures Security. Cham: Springer International Publishing; 2016. pp. 231–243. doi:10.1007/978-3-319-31664-2_24

66. After. A framework for electrical power systems vulnerability identification, defense and restoration. In: CORDIS [Internet]. 2012 [cited 9 Feb 2018]. Available: http://cordis.europa.eu/projects/261788

67. Clark G, Courtney T, Daly D, Deavours D, Derisavi S, Doyle JM, et al. The Mobius modeling tool. Proceedings 9th International Workshop on Petri Nets and Performance Models. ieeexplore.ieee.org; 2001. pp. 241–250. doi:10.1109/PNPM.2001.953373