Politecnico di Torino

Politecnico di Torino

Master's Degree in Cybersecurity

A.y. 2025/2026

Graduation Session March 2026

Automatic Synthesis of Smart Building Topologies with Integrated Access Control Policies

In collaboration with University of Calgary

Supervisors:

Prof. Riccardo Sisto

Prof. Lorenzo De Carli

Co-supervisor:

Prof. Fulvio Valenza

Candidate:

Giovanni Luca Di Bella

Summary

Smart Buildings are composed of heterogeneous sensing, actuation, and control devices organized across hierarchical layers, field, automation, and management, to provide energy and operational efficiency, security, comfort and safety. However, the increasing complexity and attack surface of Building Automation Systems have created the need to implement advanced security measures. Studies, aimed at improving security in Smart Buildings implementations, have been slowed by the scarcity of realistic network topologies for developing, testing and evaluating security measures. Existing topology generators focus more on structural graph properties than domain-specific constraints essential in Smart Buildings, such as protocol compatibility and device capacity limits. This thesis presents a constraint-based topology generation technique specifically designed for Smart Building environments, capable of producing realistic topologies that reflect real deployments. The approach is parameterized by the **number of floors** and **sensors per floor**, with optional parameters for topology control (**room capacity factor**), solver optimization (**split size**), and reproducibility (**random seed**). Additional command-line options control the output format and visualization. It generates a hierarchical topology composed of sensors, actuators, controllers, routers and servers, interconnected via BACnet and ZigBee protocols, but extendable to any protocol. The topology generation is structured as a constraint satisfaction problem, approached using Google OR-Tools CP-SAT. The architecture executes four sequential phases: sensor-room allocation, sensor-to-controller assignment, controller-to-router assignment, and router-to-server assignment. Protocol link correctness is enforced using whitelist and blacklist mechanisms, preventing devices from connecting to devices that do not support the same protocol. Due to the complex and time-consuming solving process introduced by the solver, a critical aspect is the **floor-splitting strategy** in the sensor-to-controller assignment phase. The strategy consists of dividing the floor assignment into consecutive room groups solved independently, while still respecting the overall floor constraints. This approach ensures adequate performance and provides scalability for the entire process. The generated topologies serve as input to an XACML-based access control policy generator, producing role-based, device-based, and zone-based access control policies.

This work addresses a gap in Smart Building security research by providing the first topology generator that encodes domain-specific protocol constraints compatibility, spatial coherence, and capacity limits, making the generated topologies not just structurally plausible, but semantically valid. Unlike general-purpose network generators, our tool produces topologies suitable for access control policy testing, intrusion detection evaluation, and Building Management System simulation, enabling researchers to evaluate security mechanisms in realistic Smart Building environments without requiring access to production systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rapid adoption of Smart Building (SB) technologies over the past decade has significantly transformed the way modern infrastructures are designed, managed and operated. With the integration of sensors, actuators, communication networks and control systems, the concept of SBs can optimize essential aspects such energy efficiency, comfort, safety and operational efficiency. However, this evolution also brought new cybersecurity challenges, causing systems to have a larger attack surface. In particular, the communication protocols used in these architectures were originally designed for isolated local networks, where the only security threats originated from within the building itself. Nowadays, SB face adversaries both from the local area and from the external network, calling for extra effort for the development of secure architectures. As SB infrastructures increasingly connect to the internet, the original security assumptions of communication protocols no longer hold. The new threats pose a danger not only the operational integrity of infrastructure, but can also breach the digital-physical boundary, impacting the safety of building occupants. For instance, an attacker gaining control of the fire suppression system during an emergency could prevent its activation, or the manipulation of security fogging systems (used to deploy obscuring fog to disorient intruders and protect valuables during burglary) could trap occupants during an evacuation. Attacks targeting building automation components may compromise heating, ventilation and air conditioning systems (HVAC), lighting control, surveillance systems, or even compromise physical access mechanisms.

Over the last decade, the research community has increasingly recognized the urgency of strengthening security in SB environments. Significant efforts have been made for enhancing communication protocols, intrusion detection mechanisms and formalizing access control policies. However, the development and validation of these security mechanisms need realistic environments and use cases. In particular, researchers need realistic Smart Building topologies in order to evaluate new techniques, simulate attacks and test mitigation strategies.

The biggest obstacle in the context of research is the limited availability of real-world SB topologies, since commercial vendors rarely disclose detailed architectural information about existing buildings. Multiple studies cite the same problems related to confidentiality, operational risks, and regulatory constraints. For example, Elnour et al. (2021) note: *"It is challenging to obtain actual data or gain access to real building management systems due to confidentiality, unfeasibility, etc."* [1]. Li et al. (2023) add: *"launching cyber-attacks in real buildings may be unacceptable for building owners"* [2]. Alrumaih et al. (2023) explicitly state: *"The availability of industrial network topologies is limited, which presents a significant challenge for researchers in this field"* [3].

As a result, the research community lacks realistic datasets and models for experimentation. A systematic review by Li et al. (2023) found that 38% of Building Automation System security studies relied on simulation data, while 25% proposed purely conceptual approaches without any real BAS data [2]. While topology generators have emerged for industrial control systems, such as GENIND [3], which produces hierarchical industrial network topologies, these tools do not support building-specific protocols such as BACnet, KNX, or ZigBee, leaving SB security research without realistic, reproducible network scenarios. This limitation severely slows the improvement of security research in the field.

Beyond security evaluation, realistic topology generation is essential for protocol development and validation. Protocol designers developing extensions to BACnet/IP, optimizing ZigBee routing, or evaluating KNX scalability, currently lack standardized network scenarios. A BAS-specific topology generator would enable systematic comparison of protocol performance, routing efficiency, and energy consumption across diverse building configurations, supporting both security research and next-generation protocol design. The motivation of this work arises from this gap.

1.1 Thesis Objectives

The main goal of this project is to conduct a comprehensive, detailed and systematic analysis of the necessity and design of a SB topology generator to support security research in the field. Contrary to Smart Home systems, SB must follow specific restrictions as regards the placement of safety devices, physical access control devices and the application of access control policies.

The first step consists of reviewing the existing literature on SBs and topology generation techniques; the intent is to give the reader a high-level view of the landscape. Although substantial research has been conducted independently in both domains, a research gap still exists in the integration of topology generation and SB Environments. This thesis aims to address this research gap by developing

a structured tool that enforces correctness constraints. Additionally, a focus is given on access control policies. The generated topologies are used as foundation for modeling authorization mechanisms that manage the interaction between devices, entities and zones.

After the technical specification and implementation of the generator, performance benchmarks are conducted to evaluate the scalability of the code. Particularly, the execution time is analyzed as the number of devices, floors and network components increases. Finally, this thesis discusses the limitations of the proposed implementation and outlines potential directions for future research. These include improving realism in device behavior modeling, extending protocol support, and refining automated access control policy generation.

1.1.1 Thesis Outline

This section gives an overview of the contents of each chapter of this thesis.

Chapter 2 – Background and Related Work

The second chapter provides the theoretical foundation and summarizes related work. It opens with an overview of SB, their role in modern infrastructure, and the challenges introduced by integrating heterogeneous devices and communication protocols. The chapter presents the layered Building Automation System architecture and discusses common network topologies. An overview of open communication protocols used in building automation is provided. Finally, the chapter addresses the lack of dedicated tools for automatic SB topology generation, reviewing existing work on topology generators across different network domains and motivating the need for a domain-specific solution.

Chapter 3 – Topology Generation

The third chapter presents a comprehensive description of our topology generation technique. It begins by defining the assumptions and constraints under which the system was designed, followed by a detailed description of the tool's architecture and working principles. The chapter describes the device model, the constraint definitions, and the reasoning made for the solver selection. The imperative approach is presented first, explaining its limitations which motivated the transition to constraint-based optimization using Google OR-Tools CP-SAT. The topology generation pipeline is described in detail across four sequential phases: sensor allocation to rooms, sensor-to-controller assignment, controller-to-router assignment and router-to-server assignment. Pseudo-code is also provided for each phase of the construction process, making the implementation fully understandable and transparent.

Chapter 4 – Access Control Policy Generation

The fourth chapter describes the automatic generation of XACML-based access control policies starting from the generated topologies. An overview of the XACML standard is provided. The model defines a custom attribute schema and a layered policy hierarchy. The policy generation pipeline is described, including input initialization, the action intersection mechanism that maps role capabilities to device-supported actions, and the generation of three policy types: global policies, zone-based policies, and role-based policies. A concrete policy application example is then given to show the framework result.

Chapter 5 – Evaluation and Performance

The fifth chapter presents a comprehensive experimental evaluation based on 540 experiments. The experimental setup is described, including parameter ranges 1-4 floors, 1,000-4,000 sensors per floor, split sizes 100-300, room capacity factors from 0.1 to 1.0, and three random seeds). The scalability analysis reveals approximately linear trend with fixed floors number and variable sensor count but super-linear scaling with variable floors. Phase-by-phase performance analysis shows Phase 1 (sensor allocation) dominates execution time (60-94%). Room capacity factor analysis reveals performance invariance despite significant structural differences.

Chapter 6 – Conclusion

The final chapter reflects on the insights from this work. The split-based approach, explained in Chapter 3 demonstrates that partitioning outperforms monolithic formulations. Limitations are discussed, including computational constraints, simplifying assumptions (uniform room treatment, no spatial coordinates, no fault tolerance), and protocol coverage (BACnet and ZigBee only). Future work directions include enhanced realism through spatial modeling and temporal dynamics, protocol extensions, and tool improvements for research integration. The chapter concludes by positioning the work as enabling reproducible SB security research without requiring access to proprietary production systems.

Chapter 2

Background and Related Work

This chapter gives an overview of SB and related literature. The chapter is divided in two sections: The first section explores SBs features and the threats affecting them; while the second reviews past research on SB and topology generation.

2.1 Background

SBs are automated buildings designed to save costs and increase safety and comfort by automating processes (heating, ventilation, lighting, air-conditioning) and interacting with other smart things, while improving environmental sustainability [4]. SBs are also classified as a type of Cyber Physical System (CPS), but with specific characteristics, protocols, standard and technologies [5].

The concept has evolved alongside the development of the Internet of Things (IoT). The IoT refers to a paradigm in which physical devices are equipped with sensing, processing and communication capabilities, enabling them to exchange data and interact over networks. This paradigm includes not only connected devices but also communication standards, distributed architectures, and security frameworks. SBs are generally different from the concept of Smart Homes. They span many different sectors: factory buildings, greenhouses, server rooms, airports. Building automation has existed for a long time, even before electric components were introduced[4]. It is only in recent years that companies have begun connecting all devices to the Internet to enhance functionality and improve services. A Building Automation System (BAS) creates an infrastructure of sensors, actuators and control units to monitor, increase and regulate environmental parameters in a building. A few examples include temperature and humidity control in HVAC systems, smoke detection and fire alarms, elevator management, lighting control

and access management systems.

With the expansion of network connectivity and the increasing computational capabilities of embedded devices, traditional BAS evolved into more interconnected and intelligent systems. This transformation is closely aligned with the "Industry 4.0" concept, which leverages digitization and automation to improve industrial environments. However, this digital transformation also introduces new layers of complexity. The integration of heterogeneous devices, multiple communication protocols, and control architectures creates complex network topologies that must be carefully designed and secured. Understanding and modeling these topologies is therefore a prerequisite for developing robust security solutions. This thesis exists at the intersection of SB architecture, network topology modeling, and cybersecurity research, with the objective of providing a systematic approach for generating realistic topologies for experimental purposes.

A few examples of sensors and actuators used in the SB environments are:

- **Climate Control**: heating, ventilation and air-conditioning (HVAC) systems, humidification
- **Visual Comfort**: artificial lighting, daylighting, shutters, blinds
- **Safety**: Fire alarm, gas alarm, emergency lighting
- **Security**: Intrusion alarm, access control, closed-circuit television (CCTV), audio surveillance
- **Transportation**: Elevators, escalators, conveyor belts
- **Supply**: Power distribution, waste management, water dispensing

A peculiar aspect of SBs, which is not often underlined, is that achieving interoperability between devices is not straightforward. SBs integrate a diverse ecosystem of applications and IoT devices from multiple vendors, each employing different communication protocols (BACnet, ZigBee, KNX, LonWorks, Modbus) and operating at different network layers (field, automation, management). This heterogeneity creates significant interoperability challenges, as devices must communicate across protocol boundaries. Beyond interoperability, this complexity introduces additional challenges including security vulnerabilities (due to the larger attack surface), network management difficulties (coordinating diverse devices and protocols), energy efficiency constraints (particularly for battery-powered wireless sensors), quality of service provisioning, latency reduction and bandwidth efficiency.

2.1.1 Topologies

For the development of a tool capable of generating synthetic SB Topologies, it is essential to discuss different network topologies. A network topology defines how

nodes and communication links are organized in a network. In Building Automation Systems, topology selection directly impacts energy consumption, latency, reliability, and scalability [6][7]. The choice is application-specific, involving trade-offs between **simplicity**, **fault tolerance**, and **deployment cost**. This section presents the most common topologies.

Point-to-Point

It is the most elementary structure; it consists of a dedicated high-capacity link between two nodes. It is simple and secure, but vulnerable to single-point-of-failure; any link failure immediately cuts all communication with no recovery path.

Bus

It is the simplest way to interconnect multiple nodes with each other. All nodes share a single communication medium (the bus). Data transmitted by any node travels through the bus and is readable by all others. It is low-cost and easy to deploy, but suitable only for small networks; heavy traffic degrades performance, and the bus itself is a single point of failure.

Star

All nodes are connected to a central hub through which all communication passes. To be able to communicate, each message must first pass by the central node. This topology achieves the lowest power consumption, since peripheral nodes only transmit to the hub without forwarding [6]. It is also easy to scale and diagnose, but if the central node fails, the entire network collapses.

Ring

Nodes form a closed loop; each node connects to two neighbors while data travels from a node to the next. It is a suitable architecture when all nodes handle approximately the same workload. As cons, adding or removing any node disrupts the entire network.

Mesh

Each node maintains direct connections to multiple peers, rearranging the route whenever a peer fails. This makes mesh the most fault-tolerant topology, with no single-point-of-failure weakness. The trade-off is high energy consumption and complexity, since every node participates in forwarding traffic.

Tree (Hierarchical)

Nodes are connected in a hierarchical structure. Each branch comes from a parent node and connects to leaf nodes. Data is aggregated upward toward the root, avoiding network-wide flooding and reducing average power consumption. The main drawback is uneven energy distribution: nodes near the root handle far more traffic than leaves, and parent node failures disconnect entire subtrees. The topology generation approach presented in Chapter 3 produces hierarchical-tree topologies.

Hybrid

Hybrid architectures combine two or more topologies to leverage their respective advantages while mitigating weaknesses. Past studies [6], [7] conclude that hybrid architectures are the most practical choice for real-world deployments, balancing fault tolerance, energy efficiency and scalability at the cost of greater design complexity.

2.1.2 Building Automation System Architecture

BAS improves the control, automation, and operational efficiency of building subsystems. BAS architectures are typically organized into three layers, each serving different functional roles.

Figure 2.1 shows an example of Building Management System configuration, divided into the three defined layers.

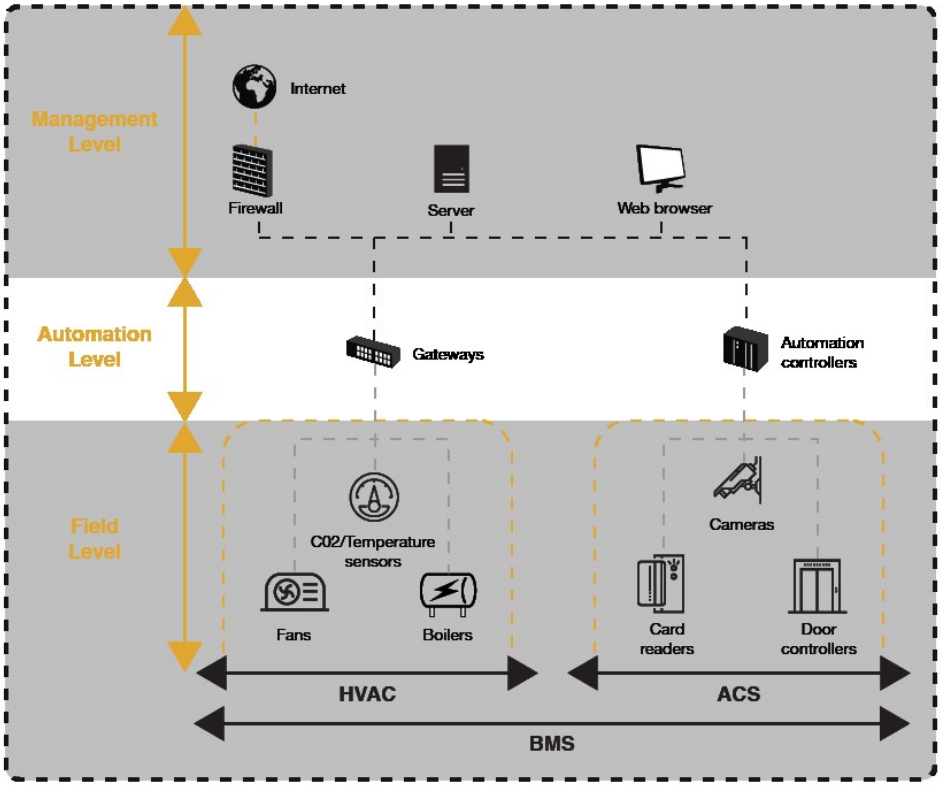

Figure 2.1: BMS topology example [8]

Field Layer

The **field layer** comprises all devices that directly interact with the physical environment, including:

- **Sensors**: Temperature, proximity, motion, optical, pipe pressure, water quality, humidity, CO_2, air quality, occupancy, light level, and fire detection sensors that monitor environmental conditions.
- **Actuators**: HVAC units, lighting controllers, Closed-circuit television (CCTV), and fans that perform physical actions.
- **Access control devices**: Card readers, keypads, biometric scanners, and door locks.

Field devices generate data from the environment or execute commands received from higher layers. They communicate upward to controllers at the automation layer or directly to servers at the management layer. Communication protocols at this layer are typically optimized for low power consumption and short-range communication.

Automation Layer

The **automation layer** contains controllers and intermediate devices that aggregate data from field devices, execute local control logic, and forward processed information to the management layer. Furthermore, controllers receive commands from higher layers and forward them to field devices. This layer performs:

- **Local control**: Real-time regulation of HVAC zones, lighting scenes, and access control.
- **Data aggregation**: Collect sensor data and forward them to Building Management Systems (BMS).
- **Command forward**: Translating high-level commands from the management layer into device-specific actions.

Controllers may also communicate with other peers using field-level protocols (BACnet MS/TP, KNX) and vertically with the management layer using IP-based protocols (BACnet/IP).

Management Layer

The **management layer** contains systems, routers, and servers that coordinate building operations.

This layer performs:

- **Centralized monitoring**: Building Management Systems (BMS) and Energy Management Systems (EMS) aggregate data from all layers, manipulating and storing it while generating commands to perform global control tasks.
- **Optimization and scheduling**: Control algorithms to optimize energy consumption, comfort, and efficiency based on historical data.
- **External connectivity**: Servers may be hosted externally to the physical building infrastructure, hence, accessed via IP networks.

The management layer connects to lower layers via IP-based communication and is typically protected by firewalls, intrusion detection systems (IDS), and VPN gateways.

Figures 2.2 and 2.3 show the hierarchical organization of a typical BAS infrastructure. This three-layer architecture forms the basis for the topology generation model described in Chapter 3, where devices are instantiated at each layer subject to protocol compatibility and capacity constraints.

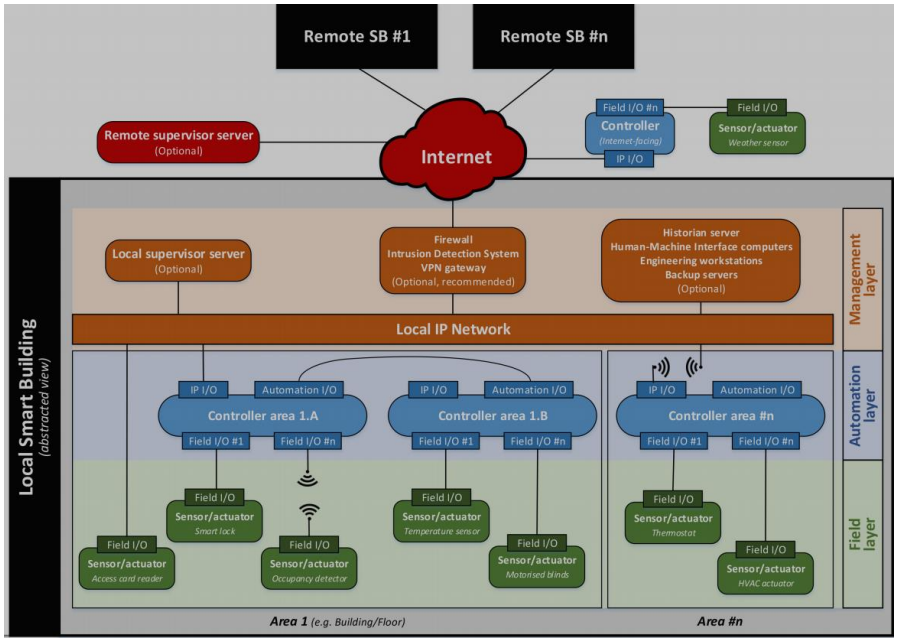

Figure 2.2: Building Automation System-Example Implementation [5]

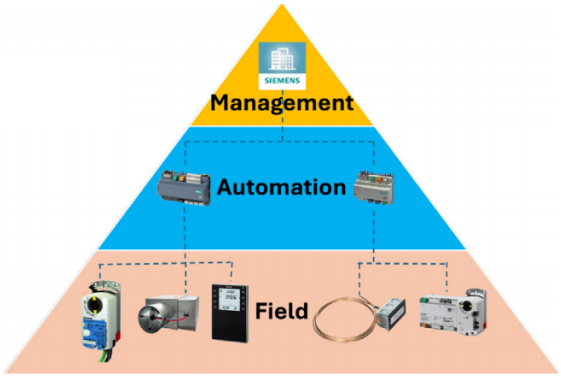

Figure 2.3: Building Automation System—Hierarchical Layer Structure [9]

2.1.3 Communication Protocols in Building Automation

Building Automation protocols define the communication rules between heterogeneous devices in SBs. Since it is not required to follow a single communication standard, each device vendor may decide to adopt either **open protocols** (publicly available standards like BACnet, KNX, ZigBee) or **proprietary protocols** (vendor-specific solutions). SBs relying on proprietary protocols require gateway devices to translate between incompatible systems. As system complexity increases,

maintaining gateways for every proprietary protocol becomes unfeasible due to integration overhead, communication latency, and maintenance complexity. Open protocols address this problem by enabling direct interoperability: devices from different vendors can communicate without translation.

Open protocols

An Open Communication Protocol allows vendors' solutions to interoperate without the need for proprietary interfaces or gateways. Open protocols offer several advantages: public specifications allow community-driven security audits and performance improvements and standardized data models eliminate the need for protocol-specific gateways. However, open protocols are not interchangeable, each is optimized for specific use cases based on factors such as transmission medium (wired vs. wireless), network topology (star, mesh, tree), data rate requirements, power constraints, and security features. Wired protocols (BACnet, Modbus, KNX) are preferred when performance and reliability are critical, particularly for backbone infrastructure. On the other hand, Wireless protocols (ZigBee, EnOcean, LoRa) are favored for battery-powered sensors, where ease of installation and low deployment cost win over performance. Table 2.1 summarizes the most widely adopted open communication protocols in building automation, classified by transmission medium.

This thesis focuses exclusively on open protocols. In particular, it implements **BACnet**, which is the most widely adopted wired protocol in BAS [10][11] and **ZigBee**, a commonly used wireless protocol. The topology generation architecture described in Chapter 3 is designed to be extended to support additional open standards such as KNX, LonWorks, and Modbus.

Type	Protocol	Characteristics
Wired	BACnet	Most popular BAS protocol; supports multiple data-link layers (Ethernet, MS/TP, ARCnet); designed for HVAC and lighting control
Wired	Modbus	Application-layer protocol; medium-independent; widely used in industrial automation
Wired	LonWorks	Proprietary LonTalk protocol; supports twisted-pair, power line, RF, and fiber; flexible peer-to-peer architecture
Wired	DALI	Digital Addressable Lighting Interface; specialized for lighting control; bidirectional communication
Wired	KNX	International standard for building automation; supports twisted-pair, power line, RF, and IP; tree topology for large networks
Wireless	ZigBee	Low-power, IEEE 802.15.4 standard; mesh network topology with auto-routing and self-healing; short range, low data rate
Wireless	EnOcean	Energy-harvesting technology; battery-free sensors; wireless ISM band; simple registration-based networking
Wireless	LoRa	Long-range LPWAN; low power consumption; chirp spread spectrum modulation; 2-15 km range; bidirectional communication
Wireless	KNX	Also supports wireless (KNX RF) in addition to wired media; tree topology; robust for large-scale deployments

Table 2.1: Classification of Open Communication Protocols by Type

Figure 2.4 illustrates the distribution of open communication protocols across the layered architecture of BAS. At the **field layer**, protocols such as KNX, LonWorks, and ZigBee are used to connect field devices. The **automation layer** includes protocols like BACnet and Modbus, used to manage controllers and enable coordination as well as in the **management layer** [12].

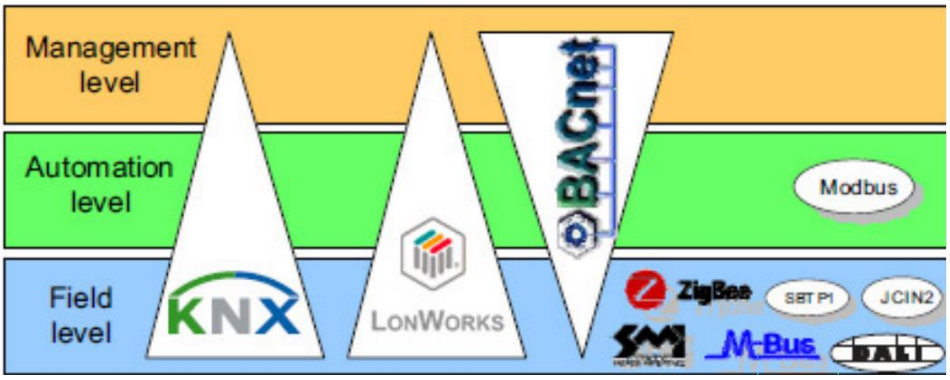

Figure 2.4: Building Automation System - Protocols layer distribution [12]

BACnet Protocol

Building Automation and Control network (BACnet) is a communication protocol designed to manage HVAC systems, lighting systems, and other building services. It is by the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE)[12], it is the most widely adopted network protocol in BAS [10][11] and offers a flexible standard that allows vendors to provide custom and unique applications while maintaining interoperability.

It was designed specifically to address the needs of building automation and control systems of all sizes and types [12]. It supports multiple data-link layer protocols, including **Ethernet**, which supports several types of media, including UTP, fiber or even wireless, and is mainly used to form the network backbone; and **MS/TP (Master-slave/token-passing)**, which is defined for devices with lower requirements in terms of speed. The remaining supported protocols are ARCnet, PTP (Point-to-Point), LonTalk, BACnet/IP. Since all Data-Link layer protocols are wired, BACnet has no native support for wireless communications. BACnet operates on four of the seven levels of the OSI model: Application, Network, Data Link and Physical. An important conceptual distinction in BACnet is between an **object**, intended as a set of information representing an input or output, and a **service**, which is the mechanism used to access a property of an object or request an action [12].

A deployment example, shown in Figure 2.5, involves two networks connected through BACnet/IP routers: Network A uses the BACnet MS/TP protocol over RS-485 connectors, while Network B uses BACnet/IP over Ethernet cables. Within such a network, it is possible to have multiple segments. One using BACnet/IP to encapsulate BACnet messages within IP packets, and another using BACnet MS/TP, where a BACnet controller and an air quality sensor communicate over a twisted-pair (TP) wire connected to the segment's router [11][9]. BACnet is often the first choice at the Management Level [13].

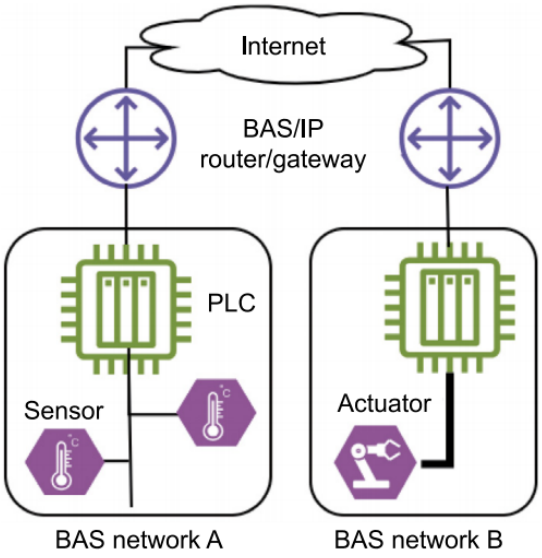

Figure 2.5: Building Automation System - Layers [9]

ZigBee ZigBee is a low-power, short-range wireless communication protocol, based on the IEEE 802.15.4 standard. It supports mesh networking, which improves coverage and reliability, and is primarily used in home automation (smart lighting, thermostats, sensors), industrial monitoring and healthcare devices (scenarios characterized by short-range and low data rates) [11]. The most important feature is its mesh network topology, which is both self-routing and self-healing, according which, if a link breaks, devices can automatically discover and alternate route [11]. This also gives it a long reach, making it suitable for large buildings. ZigBee is designed for Wireless Personal Area Networks (WPANs) and supports three network topologies: star, tree and mesh [9].

A ZigBee network is composed of three types of entities: ZigBee Coordinator, which acts as the root of the network tree, ZigBee Router, which forwards messages and End Devices, which in a tree topology can only communicate through their parent node and cannot communicate directly with each other.

2.1.4 BACnet over ZigBee

As defined above, BACnet is the most common wired open communication protocol used in SBs and BAS, while ZigBee is widely used as a wireless protocol. However, since BACnet does not provide native wireless communication support, several solutions have been proposed to integrate ZigBee into BACnet infrastructures.

Song et al. [14] explore two approaches for BACnet-ZigBee integration: a **gateway solution** using ZigBee's Key-Value Pair (ZigBee KVP) service model

with mirror objects to translate between ZigBee attributes and BACnet objects, and a **direct stack integration** approach placing BACnet's application and network layers on top of ZigBee's Application Support Layer. The gateway approach suffers from manual commissioning overhead, meaning that when a node is added, mirror objects and BACnet objects must be manually commissioned in the gateway, maintenance complexity, and potential data loss during object mapping; the direct integration, instead, eliminates gateways but requires specialized BACnet-over-ZigBee routers.

Tian et al. [15] present a hybrid transmission approach addressing the inefficiencies of traditional polling-based gateways, where ZigBee devices are encapsulated as BACnet devices with BACnet object attributes, BACnet NPDUs are carried as ZigBee application payloads, using the ZigBee 64-bit MAC address as device identification. This eliminates polling overhead and enables direct BACnet client access to ZigBee devices. Validation demonstrates query response times of approximately *100 ms* per device, significantly faster than traditional gateway approaches. These integration strategies inform the protocol compatibility mechanisms required for realistic SB topology generation, as discussed in Chapter 3.

2.1.5 Smart Buildings Security

As mentioned previously the integration of networks of sensors, actuators and controllers for the automation of buildings procedures introduces new possible threats. These systems rely on open standardized protocols and are connected to the Internet, enabling remote managing and data processing. These innovations inevitably increase the attack surface of the system.

Whereas traditional BAS relied on mostly electronic and mechanical elements to manage building functions through isolated and proprietary networks, modern SBs are accessible via the Internet, exposing them to attackers who can exploit standard protocols not originally designed to defend against external threats [5].

Vulnerabilities and attacks can target each layer of a SB independently. More critically, complex attacks can span multiple layers: an attacker might compromise field-layer sensors, inject false data, consequently manipulating automation-layer controllers [14].

Root Causes of Vulnerability

Security weaknesses in modern SBs originate from several fundamental structural problems:

- **Legacy protocol problem**: Protocols such as BACnet, KNX, LonWorks were developed many years before the modernization of public structures, with the intent of creating communication protocols for isolated environments.

Security features such as authentication, confidentiality, integrity, and access control were either absent or minimal as the original design assumption was that only trusted operators within the building would have network access. All protocols have in common the lack of security-by-design principle, meaning security was not considered a core requirement during their development. When security features were eventually added, they were often implemented as optional extensions rather than fundamental protocol components. The building automation industry and vendors began addressing seriously protocol security only many years after the disclosure of severe protocol security weaknesses. As a result, millions of deployed devices still operate using insecure protocol versions [4].

- **Internet Exposure**: Driven by market demand for remote monitoring and control, building automation vendors integrated Internet connectivity into previously isolated systems. However, this connectivity was implemented without adequate security in the underlying protocols. Consequently, security researchers discovered thousands of BACnet and KNX devices directly exposed to the Internet with no authentication requirements, default credentials, or unpatched vulnerabilities, making them easily accessible to remote attackers [4].
- **Attack Taxonomy**: Christopher Morales-Gonzale et al [9] provide a comprehensive taxonomy of attacks against BAS, categorized by attack surface and protocol vulnerabilities. Table 2.2 summarizes the principal attacks types and the protocols they affect.

Attack Type	Description	Affected Protocols
Physical / Hardware	Tampering with or destroying field devices, wires, and sensors to cause DoS or gain direct network access.	BACnet, KNX, ZigBee, Z-Wave
Denial of Service (DoS)	Resource exhaustion, RF jamming, or device lockout that renders BAS components unavailable.	All major protocols
False Data Injection (FDI)	Injecting forged sensor readings or commands to manipulate actuators (e.g. falsifying temperature readings to alter HVAC behavior).	BACnet, KNX, ZigBee, Modbus
Man-in-the-Middle (MITM)	Intercepting and silently altering traffic between BAS nodes; feasible due to absent message authentication in legacy protocols.	BACnet, KNX, LonWorks, ZigBee
Eavesdropping & Replay	Passively capturing cleartext traffic; replaying captured commands to reproduce their physical effects.	KNX, ZigBee, EnOcean, Z-Wave
Spoofing	Impersonating a legitimate BAS node to issue unauthorized commands or retrieve sensor data.	BACnet, KNX, ZigBee, EnOcean
Device Reprogramming	Overwriting firmware to alter device behavior, permanently disable it, or pivot into the rest of the network.	KNX, ZigBee, Z-Wave, Modbus
Privacy / Surveillance	Extracting occupancy, energy, or timing data from Internet-connected SBs to infer the behavior of building occupants.	Internet-connected SBs

Table 2.2: Building Automation Systems - Main attacks [9]

At the **field layer**, the most prevalent threats are physical attacks and wireless attacks targeting ZigBee, Z-Wave, and EnOcean communications. At the

automation and management layer, Denial of Service attacks are the most documented category, followed by attacks taking advance of absent authentication methods in BACnet and KNX messages.

2.2 Related Work

Following the overview of SBs, this section analyzes the current state of scientific research in network topology generation for industrial systems and BAS. The main objective is to identify the research gap regarding topology generation for SB environments. While topology generation has been studied extensively in industrial control systems and IoT networks, limited attention has been given to BAS with their specific protocol requirements and hierarchical constraints.

2.2.1 Industrial Network Topology Generators

Alrumaih and Alenazi [3] present GENIND, a graph-theoretic topology generator for industrial networks addressing the limited availability of realistic test topologies for Industrial Internet of Things (IIoT) environments. GENIND generates hierarchical three-layer network topologies (Network, Controller, and Sensor layers) validated against real-world industrial datasets using structural metrics including clustering coefficient, edge density, transitivity, and average distance. The generator operates through two sequential algorithms: a **multi-graph generator** that builds each layer independently, followed by a **combine-and-link** phase that merges the layers into a unified topology. Results demonstrate that generated topologies closely match real-world networks across these structural parameters. However, GENIND has notable limitations for SB applications: it does not account for **protocol compatibility constraints** between devices, **generates layers independently** before linking them (risking unbalanced configurations), and does not consider device **capacity limits or physical connection constraints** that real controllers and routers impose in operational deployments.

Bechtel et al. [16] introduce GeNESIS, a tool designed to generate reproducible industrial network evaluation scenarios for factory automation. GeNESIS addresses the lack of standardized, realistic industrial network topologies that hinders comparative research evaluation. The tool generates hierarchical network topologies following the Purdue model with four layers (Field, Control, Supervisory, Enterprise), supporting four topology patterns per layer (daisy chain, ring, star, mesh). GeNESIS differentiates seven device categories (OT devices, PLCs, SCADA workstations, IT devices, servers, switches, routers) and implements three traffic profiles (strict isolation, converged networks, distributed control) to model realistic communication patterns. The generator produces complete network configurations

including device addressing and firewalls in an IETF-standard format, with a compact exchange format (GeNESIS-TAG) enabling reproducibility without requiring dataset repositories. GeNESIS supports industrial protocols including PROFINET, EtherCAT, OPC UA, and EtherNet/IP, but does not address building-specific protocols such as BACnet, KNX, or ZigBee.

2.2.2 Synthesis of Large-Scale Instant IoT Networks

Cheng et al. [17] address automated network synthesis for Instant IoT Networks satisfying a rapid deployment, creating a temporary sensor networks with coverage, visibility, and connectivity requirements. The synthesis problem is formulated as a constraint satisfaction problem over a discretized grid representation, with four constraints: **coverage** (each location covered by k sensors), visibility (line-of-sight requirements), connectivity (network reachability), and placement (obstacle avoidance). The authors compare two solvers: Satisfiability Modulo Convex Optimization (SMC), which extends Boolean SAT with convex constraints and uses Z3 for solving, and Mixed-Integer Linear Programming (MILP), which approximates coverage as a k-vertex cover problem and connectivity as a Steiner tree[17]. SMC provides superior expressivity for convex geometry constraints, while MILP's linear approximation degrades with small, numerous obstacles. For large deployment regions, the authors introduce hierarchical synthesis: the area is divided into sub-regions solved independently for coverage, followed by cover repairing phases to remove redundant sensors and connectivity repair to link sub-networks. Results show SMC consistently outperforms MILP for large regions, producing solutions with approximately 50% lower coverage redundancy. MILP is preferable only for smaller problem sizes with moderate obstacle density. This work demonstrates the effectiveness of constraint-based approaches for network generation problems, though it focuses on physical sensor placement rather than protocol-aware hierarchical topology generation for building automation.

2.2.3 Wireless Sensor Network Topology Generators

Li et al. [18] propose an optimal design method for communication topology of wireless sensor networks implementing fully distributed optimal control in IoT-enabled SBs. The authors address the challenge that communication topology affects three critical aspects: system control performance (convergence speed of distributed optimization algorithms), network energy consumption (data transmission and routing efficiency), and network stability (balanced energy distribution among nodes). They formulate an optimization problem using genetic algorithms to design optimal topologies for a wireless sensor network controlling a multi-zone dedicated outdoor air system (DOAS). The Laplacian matrix representation enables optimization

of $n(n-1)/2$ design variables representing communication links between sensors. Results demonstrate that the optimal topology provides satisfactory HVAC control performance (CO_2 average 784 ppm, energy 122.50 kWh/day), low network energy consumption (2564.12 J/day), and high network stability (53.90 days). However, the approach focuses on optimizing communication topology for distributed control algorithms rather than generating realistic physical network layouts that reflect building automation deployment constraints.

Camilo et al. [19] introduce GenSeN (Generator for Sensor Networks), a topology generator specifically designed for wireless sensor networks based on real deployment experiences. The authors identify that network simulators require realistic topologies since node placement significantly influences simulation results, Existing tools either target wired Internet topologies (BRITE, GT-ITM, Inet) or provide only basic random placement (TopoGen). GenSeN implements six deployment strategies validated through real-world experiments: **grid** (optimal uniform distribution), **one-by-one** (manual individual placement), **two-by-two** (paired deployment), **three-by-three** (grouped deployment), **cliff** (deployment from elevated positions simulating aerial drops), and **propellant** (rapid dispersal from central point). The tool generates configuration files specifying node positions, energy parameters, and antenna orientations. Validation experiments demonstrate that grid deployment achieves uniform coverage, while propellant and cliff strategies concentrate nodes in the center, leaving edge regions empty. GenSeN addresses wireless sensor network physical deployment but does not model building-specific hierarchical structures (field/automation/management layers) or protocol compatibility constraints (BACnet, ZigBee interoperability).

Medina et al. [20] present BRITE (Boston university Representative Internet Topology gEnerator), a universal topology generation framework addressing the lack of standardized, realistic network topologies for Internet research. The authors identify that existing generators (Waxman, GT-ITM, Tiers, Barabási-Albert models) produce topologies with different structural properties, making comparative evaluation difficult and forcing researchers to learn multiple tools. BRITE follows three design principles: **representativeness** (accurate reflection of Internet topology properties), **inclusiveness** (combining strengths of multiple generation models in a single tool), and **interoperability** (interfaces to ns-2, SSF simulators). The tool implements both **flat topologies** (devices are connected each other, making it easy to set up for small environments) and **hierarchical topologies** (top-down and bottom-up). BRITE uses graph-theoretic representations and supports extensible model development through an abstract Model base class. Validation against AS-level data demonstrates that models reproduce power-law out-degree distributions observed in real Internet topologies, while GT-ITM models fail to capture degree-related properties. However, BRITE targets the Internet and general network topologies rather than BAS, lacking support for building-specific protocols

(BACnet, KNX, ZigBee), hierarchical BAS layers (field/automation / management) and physical deployment constraints (device capacity limits, spatial room/floor organization).

Chapter 3

Smart Building Topologies generation

This chapter describes the architecture, design decisions and implementation of the tool developed for the automatic generation of realistic SB topologies. The chapter is structured as follows: First, we introduce the **motivations** behind the **design choices** and the overall processing pipeline; Then, the **device model** and **configuration system** are described in detail. Successively, we discuss the **constraint-based generation** and the possible **solvers** considered, motivating at the end of the discussion the final choice. Each **phase of the pipeline** is described, providing pseudo code and explaining the constraints enforced at each step. Then, it is given a **description of the output** format, necessary for the second phase of the tool: access control policies generation.

3.1 Motivation and Design Principle

The main objective of this work is to produce topologies that are not only realistic but also valid for the specific domain of SB. Validity, in this context, includes protocol compatibility, hardware connection limits (maximum supported endpoints), and spatial organization of devices across floors and rooms. The reason why SBs are sensitive to these parameters is the huge number of low-computational-capacity devices from different vendors and the huge variety of devices types they comprise. This requirement distinguishes the proposed tool from general-purpose topology generators like GENIND [3], which prioritizes structural properties such as clustering coefficients and degree distribution without considering realistic domain-related constraints.

SBs present unique challenges compared to industrial automation environments.

While industrial facilities typically deploy standardized set of sensors and actuators, SBs integrate thousands of different devices from diverse vendors. These devices often follow only minimal quality assurance standards defined by regulatory bodies, without following any international standard for communication and interoperability.

The tool addresses three fundamental user needs in Smart Building topology generation:

- **Scalability Control**: Users must specify the building scale through floor count and sensor density. This reflects the real-world design process where system integrators start from physical constraints (building size, occupancy requirements) to determine sensor deployment.
- **Topology Realism**: Users require control over spatial sensor distribution patterns to match different deployment scenarios. This flexibility ensures generated topologies reflect diverse real-world building configurations.
- **Reproducibility and Optimization**: For experimental validation, users need deterministic topology generation and the ability to tune solver performance for large-scale scenarios.

These requirements distinguish our approach from graph-based generators that focus solely on structural properties without domain-specific realism.

3.2 Device Model and Configuration

To implement the design principles described above, the topology generator is built upon a device model that captures both device type characteristics and device instance constraints. The device types included in the model (sensors, actuators, controllers, routers, servers) and their protocol assignments are derived from the survey literature and protocol specifications cited throughout this thesis [5][9][11][12][21]. The capacity constraints (*device_limit* attribute) represent illustrative values chosen to demonstrate the generator's ability to enforce realistic hardware limitations, rather than referring to specific commercial product models. The values used in this implementation serve to validate the constraint-based approach and can be easily adjusted to match specific deployment scenarios through the external configuration files.

This section describes the device representation, the complete device set, and the layered organization used during topology generation.

3.2.1 Device Representation

The device model distinguishes between *device types* and *device instances*. Each device type is represented by a **Device** object that serves as a template containing all characteristics common to devices of that type. Table 3.1 lists the attributes of a Device object.

Attribute	Description
id	Unique string identifier used throughout the system. It specifies both the device type and the instance number (e.g., `bacnet_ms_tp_plc_controller_inst_3`).
name	Unique string used as visual name for visualization (e.g., `bacnet ms tp plc controller #inst1`).
layer	The BAS layer to which this device belongs: *field_layer*, *automation_layer*, or *management_layer*. This information is used exclusively for organizing the NetworkX graph representation and is subsequently exported to the output JSON file for hierarchical visualization and analysis.
protocol	List of supported communication protocols (BACnet, ZigBee). Determines which devices can be connected.
type	String value specifying whether the device is a sensor, actuator, controller, router, or server.
upper_white_list	List of device type IDs that this device is permitted to connect to (upward in the hierarchy).
lower_white_list	List of device type IDs that are permitted to connect to this device (downward in the hierarchy).
actions	List of possible actions an entity can perform on the device. Essential for Access Control Policy generation to verify permitted actions.
device_limit	Maximum number of devices that can connect to this device.

Table 3.1: Device Object definition

At runtime, concrete instances of devices are represented by ***DeviceInstance*** objects, which combine a Device type descriptor with an instance number and, for room-level devices, the spatial information (floor and room identifiers). The distinction between static type descriptors and runtime instances is necessary because the optimal number of devices, particularly controllers, coordinators, routers and servers, cannot be determined a priori. The actual quantity depends

on the solver's allocation decisions during solving: for instance, the number of BACnet controllers versus ZigBee coordinators needed depends on how sensors are assigned to protocols, which is determined randomically by the solver at run-time. Additionally, the total number of controllers required depends on sensor distribution patterns and the capacity of each controller type. To handle this uncertainty, the generator pre-allocates a pool of *DeviceInstance* objects for each device type, sized to accommodate the maximum possible number needed. The solver then decides which instances to activate via Boolean decision variables (*exists_var*): instances with *exists_var = 1* are included in the final topology, while inactive instances (*exists_var = 0*) are discarded. This approach allows the solver to optimize device instantiation as part of the constraint satisfaction problem, minimizing resource usage while satisfying all connectivity and capacity constraints.

Table 3.2 describes the attributes of a DeviceInstance object.

Attribute	**Description**
device_type	Is of type *Device* and contains all the essential characteristics of the specific device type.
instance_num	Integer value defining, sequentially, the instance number for the specific device.
floor, room	Integer values defining respectively which floor and room a device belongs to.
exists_var	Boolean decision variable in the solver model.

Table 3.2: DeviceInstance Object definition

3.2.2 Device Catalogue

The device catalogue is loaded from the file *devices.json* and spans three layers of the BAS hierarchy. The following sections describe the device types available at each layer.

Field Layer Devices

The field layer comprises 17 sensor and actuator types, all supporting both BACnet and ZigBee protocols. These devices can be categorized into three groups:

- **Environmental Sensors:** Motion Sensor, Light Sensor, Air Quality Sensor, Temperature Sensor, Humidity Sensor, Pressure Sensor.
- **Safety and Security Devices:** Fire Detection, Fire Alarm & Suppressor, Fire Sprinkler, Infrared Sensor, Infrared Camera, Camera, Card Reader, Keypad.

- **Actuators:** HVAC Units, Fans, Air Ventilation System.

All have a device_limit of 1, meaning they are leaf nodes with no downstream connections.

Automation Layer Devices

The automation layer contains four controller types that aggregate field devices and forward data to the management layer. The implementation focuses on BACnet, the most widely adopted wired open protocol in building automation, and ZigBee, a widely used wireless open protocol, providing representative coverage of both wired and wireless communication paradigms (see Section 2.1.3):

- **BACnet MS/TP PLC Controller**: A BACnet-only controller with configurable device limit. It aggregates BACnet sensors and actuators over RS-485 connections using the Master-Slave/Token-Passing protocol.
- **Access Control Controller**: A protocol-generic controller dedicated exclusively to access control devices such as card readers, keypads, and cameras.
- **ZigBee Router**: A ZigBee-only controller that aggregates ZigBee sensors and forwards data to a ZigBee Coordinator.
- **ZigBee Coordinator**: The root node of a ZigBee subnetwork that aggregates ZigBee Routers. The coordinator bridges the ZigBee network to the BACnet/IP backbone.

Management Layer Devices

The management layer contains protocol gateway routers and servers:

- **Gateway Routers:**
  - **BACnet IP to MS/TP Router**: Connects BACnet MS/TP controllers to the BACnet/IP backbone. This router translates between the MS/TP protocol and BACnet/IP over Ethernet.
  - **BACnet IP to ZigBee Router**: Connects ZigBee Coordinators to the BACnet/IP backbone using the MSG (generic message) encapsulation model described in Section 2.1.4. This enables communication between ZigBee devices and BACnet management systems.
- **Supervisory Servers:**

- **Central BMS Server**: The Building Management System server providing supervisory control, real-time monitoring, alarm management, and data visualization for all building subsystems[8].
- **Central EMS Server**: The Energy Management System server dedicated to energy monitoring, optimization algorithms, and reporting for regulatory compliance and operational cost reduction.

3.2.3 Layer Builders

Each layer is managed by a **LayerBuilder** object, responsible for pre-instantiating device instances for its layer before the solving process, registering their *exists_var* Boolean decision variables with the CP-SAT model, and enforcing the sequential instantiation constraint described in Section 3.4.5. The maximum number of instances per device type is computed analytically before solving, using the *compute_capacity()* and *calculate_controllers_needed_for_sensors()* functions. This pre-solving estimation, described in detail in Section 3.7, provides the solver with a bounded search space, improving solving time while ensuring that enough device instances are available to satisfy constraints.

3.3 Implementation Details

3.3.1 Input Parameters

The generator accepts command-line parameters to control topology generation described in Table 3.3.

Parameter	Type	Description	Default
Mandatory Parameters			
`floors`	int	Number of building floors.	—
`sensors_per_floor`	int	Number of sensor instances to allocate per floor.	—
Topology Control (Optional)			
`room_capacity_factor`	float	Room capacity scaling factor ($rm\_dist \in (0, 1]$). **Higher values** produce fewer, larger rooms with concentrated sensor placement; **lower values** create more, smaller rooms with distributed placement.	0.5
Solver Optimization (Optional)			
`split_size`	int	Batch size for Phase 2 sensor-to-controller assignment. Controls the trade-off between solver performance and overhead.	200
`seed`	int	Random seed for reproducible topology generation. When specified, produces identical topologies across multiple runs with the same parameters.	random
Output Configuration (Optional)			
`output_json`	path	File path for the generated JSON topology file containing the complete device hierarchy and connections.	topology.json
`output_xml`	path	File path for the XML export of the topology, compatible with external analysis tools.	topology.xml
`timing_report`	path	File path for the detailed timing report containing phase-by-phase execution statistics.	timing.txt
`no_visualization`	bool	If enabled, suppresses the interactive graph visualization window. Useful for batch processing and automated experiments.	false

Table 3.3: Topology Generator Command-Line Parameters

3.3.2 Configuration Files

Device specifications and constraints are loaded from external JSON files:

- `devices.json`: Device types, capacities, protocols
- `constraints.json`: Compatibility whitelists, constraints

This externalization enables domain customization without modifying source code.

3.4 Constraints definition

The topology generation process enforces two categories of constraints with different sources and purposes:

- **Configurable domain constraints** (from *constraints.json*): Protocol compatibility whitelists and room-specific device requirements that vary across building types and deployment scenarios.
- **Structural solver constraints** (embedded in CP-SAT model): Device capacity limits, room bounds, sequential instantiation.

The distinction between these categories reflects a design principle: domain knowledge that varies across deployments (protocols, room requirements) is externalized in JSON configuration files, while fundamental structural properties (capacity limits, connectivity rules, optimization objectives) are embedded in the solver model. This separation enables the tool to be reconfigured for different building types.

3.4.1 Protocol Compatibility through whitelists

The whitelist mechanism enforces protocol compatibility between devices at different layers of the hierarchy. Each device type has two whitelist attributes that restrict which connections are permitted: *upper_white_list* (restricts upward connections) and *lower_white_list* (restricts downward connections). For each potential connection between a source device *src* and a destination device *dst*, the *check_whitelist()* method in the phase builder performs two checks:

- **Lower whitelist check**: If the destination device defines a *lower_white_list*, the source device type id must appear in the list for the connection to be permitted. Used to prevent unauthorized devices from connecting to lower-layer devices that don't support the connection.
- **Upper whitelist check**: If the source device defines an *upper_white_list*, the destination device type id must appear in the list for the connection to be permitted. Used to prevent devices from connecting to incompatible to upper-layer devices.

If a check fails, no connection variable is created for the (*src*, *dst*) pair. This filtering phase improves both performance and correctness by preventing the solver from evaluating connections that violate protocol compatibility rules.

Table 4.1 shows the whitelist configuration used in the experimental phases. The table is organized by device type and shows which connections are permitted through upper or lower whitelists.

Device	**List Type**	**Permitted Connections**
ZigBee Router	upper	ZigBee Coordinator only
ZigBee Coordinator	upper	BACnet/IP to ZigBee router only
BACnet MS/TP PLC Controller	upper	BACnet/IP to MS/TP router only
Access Control Controller	upper	BACnet/IP to MS/TP router only
Keypad	upper	Access Control Controller only
Card Reader	upper	Access Control Controller only
Central BMS server	lower	BACnet/IP to ZigBee router, BACnet/IP to MS/TP router
Access Control Controller	lower	Card Reader, Keypad only
BACnet/IP to ZigBee router	lower	ZigBee Coordinator only
BACnet/IP to MS/TP router	lower	Access Control Controller, BACnet MS/TP PLC Controller

Table 3.4: Constraints definition - Whitelist and Blacklist Building

The combined effect of the two lists is to enforce the correct protocol-defined connections: ZigBee devices route exclusively through the ZigBee subnetwork (ZigBee Router, then ZigBee Coordinator and finally BACnet/IP-to-ZigBee Router); BACnet MS/TP devices route through the wired BACnet backbone (BACnet MS/TP PLC Controller and BACnet/IP-to-MS/TP Router); access control devices, instead, connect to the generic Access Control Controller, converging to the BACnet backbone.

3.4.2 Device Connection Limit

Each device type has a maximum connection capacity specified by the *device_limit* attribute. This limit represents the maximum number of downstream devices that can connect to an instance of a device, based on hardware specifications and protocol constraints (e.g., a ZigBee coordinator supports up to 32 devices, a BACnet controller handles 64-128 points depending on the model).

To enforce these constraints, the solver creates a Boolean decision variable ***assignment*[(*src*, *dst*)]** for each potential connection, where *src* is the source (child) device and *dst* is the destination (parent) device. The effective capacity for each device is counted as $device\_limit - 1$. This convention has been chosen to reserve one connection slot for the upward link of the device. To enforce device limit constraints, the solver defines two complementary constraints:

- **Downward connection limit:** For every destination device, *dst*, the total number of incoming connections from source devices cannot exceed its available capacity:

$$\sum_{\text{src}} \text{assignment}[(\text{src}, \text{dst})] \leq \text{dst.device\_type.device\_limit} - 1 \tag{3.1}$$

- **Upward connection requirement:** For every source device, *src*, exactly one connection to a destination device must exist to ensure hierarchical connectivity:

$$\sum_{\text{dst}} \text{assignment}[(\text{src}, \text{dst})] = 1 \tag{3.2}$$

This constraint ensures that every device has exactly one parent in the hierarchy. Consequently, the model assumes a single communication path between devices and does not consider redundant links or alternative routing paths.

Additionally, the solver enforces a connection-existence constraint:

$$\text{dst.exists\_var} = 0 \implies \sum_{\text{src}} \text{assignment}[(\text{src}, \text{dst})] = 0 \tag{3.3}$$

This ensures that non-instantiated devices cannot participate in the topology. After solving, any device with zero connections is guaranteed to have *exists_var* = False.

3.4.3 Room-level Sensor Constraints

Rooms are the elementary spatial units in the topology generation. Each room is represented by a **Room** object containing the following attributes:

Attribute	Description
floor	Integer identifier of the floor containing the room.
room_id	Integer identifier unique within the floor.
is_used	Boolean solver variable indicating whether the room contains any devices.
min_capacity	Minimum number of sensors required if the room is used.
max_capacity	Maximum number of sensors permitted in the room.
device_counts	Dictionary working as a counter for each sensor type in the room (strictly limited to field layer sensors).

Table 3.5: Room Object Definition

In the constraint configuration file, it is specified *minimum* and *maximum device counts* for specific device types per room.

Room-level constraints are enforced only for active rooms (*is_used* = True). For inactive rooms, the solver defines a conditional constraint that forces total sensor count to zero:

3.4.4 Room Capacity Bounds

The topology generation technique introduces the concept of **rooms** as a spatial unit for sensor allocation. Rooms serve two critical purposes in the generation process:

- **Realism**: Real building automation deployments organize sensors spatially (e.g. temperature sensors, occupancy detectors, and lighting controls are placed in specific rooms, not arbitrarily distributed across floors). Defining the concept of room ensures that generated topologies reflect realistic layouts.
- **Problem decomposition**: Rooms provide a natural partitioning mechanism for the constraint satisfaction problem. By allocating sensors to rooms in Phase 1 before connecting them to controllers, the solver decomposes a large allocation problem into manageable subproblems. This decomposition reflects how BAS are actually designed and significantly improves solver performance.

Each room on each floor has a dynamically computed minimum and maximum total sensor capacity bounds. The minimum capacity is fixed by the configuration file as the sum of all *minimum_devices* values specified for required device types. The maximum capacity is randomly drawn from the range [*minimum_capacity*, *maximum_devices_per_room*] at initialization. Additionally, to provide the user control over the sensor distribution, a *room capacity factor* ($rm\_dist \in (0, 1]$) parameter has been added to scale the theoretical maximum capacity of each room.

Lower values produce more rooms with smaller capacities (distributed deployment), while higher values produce fewer rooms with larger capacities (concentrated deployment). As demonstrated in the performance evaluation (Chapter 5), the room capacity factor has minimal impact on generation time (<5% of variance), allowing the user adjustment based on topology requirements without performance penalties.

The capacity bounds are computed as follows:

$$\text{max\_possible\_rooms} = \left\lceil \frac{\text{sensors\_per\_floor}}{\text{min\_devices\_per\_room}} \right\rceil \tag{3.4}$$

$$\text{max\_avg\_capacity} = \frac{\sum \text{max\_device\_per\_room}}{\text{max\_possible\_rooms}} \tag{3.5}$$

$$\text{min\_rooms\_needed} = \left\lceil \frac{\text{sensors\_per\_floor}}{\text{max\_avg\_capacity}} \right\rceil \tag{3.6}$$

$$\text{max\_rooms\_needed} = \left\lceil \frac{\text{sensors\_per\_floor}}{\text{min\_devices\_per\_room}} \right\rceil \tag{3.7}$$

Regardless of the number of rooms needed, the solver determines the exact number by minimizing total room usage.

3.4.5 Sequential Instantiation Constraint

To ensure consistent device numbering, a **sequential instantiation constraint** is enforced in the automation and management layer. This constraint serves three important purposes:

- **Deterministic identification**: A sequential numbering creates a predictable mapping between device instances and their roles. For example, if the solver activates 5 controllers on a floor, they are guaranteed to be numbered 1, 2, 3, 4, 5 rather than arbitrary identifiers. This makes the generated topology easier to interpret, validate, and integrate into building management systems.
- **Preventing fragmented allocation**: Without sequential ordering, the solver could arbitrarily distribute sensors across controller instances (e.g., 10 sensors on controller 1, 15 on controller 5, leaving 2, 3, 4 empty). Controller i must be utilized before controller $i + 1$ is activated.
- **Capacity-based activation constraint**: Device instance $i + 1$ can only be instantiated if device instance i has reached or nearly reached its connection capacity.

Sequential Existence Constraint

For each device type, the solver enforces that the i-th device instance cannot exist unless the $(i-1)$-th instance already exists:

$$\text{instance}[i-1].\text{exists\_var} = 1 \implies \text{instance}[i].\text{exists\_var} \in \{0, 1\}$$

$$\text{instance}[i-1].\text{exists\_var} = 0 \implies \text{instance}[i].\text{exists\_var} = 0$$

This constraint eliminates gaps in device numbering: if n controllers are needed, they are numbered 1 through n consecutively.

Capacity-Based Activation Constraint

To prevent wasteful allocations, the solver enforces that each device instance $i+1$ can only be activated if the previous instance i has reached its device capacity:

$$\text{instance}[i+1].\text{exists\_var} = 1 \implies \sum_{\text{src}} \text{assignment}[(\text{src}, \text{instance}[i])] \geq \text{instance}[i].\text{device\_limit} \quad (3.8)$$

This constraint ensures that devices are utilized efficiently. For example, if each controller supports 32 devices and 50 sensors need allocation, the solver must assign 32 sensors to controller 1 before activating controller 2 (which receives the remaining 18 sensors). This prevents scenarios where 50 sensors spread across 3 or 4 partially-filled controllers. These sequential instantiation constraints apply only to automation and management layer devices (controllers, coordinators, routers, servers). Field layer devices (sensors, actuators) use a different instantiation mechanism based on integer counters, as explained in Section 3.6.

3.5 Solver description and selection

The topology generation is formulated as a **constraint satisfaction and optimization problem**. At each phase, the system must find an assignment of devices and connections that satisfies all **hard constraints** and optimizes a given objective (minimize rooms used, minimize device instances, or find any feasible assignment). In this thesis work, three classes of solvers have been considered and tested, but only one was chosen for its performance and constraint satisfaction results.

3.5.1 Imperative Approach

The initial version of the tool was developed following an imperative programming approach, with the objective of avoiding the complexity and computational overhead associated with constraint solvers. While the structure follows the same principles explained in the previous sections, the core topology generation logic was implemented through heuristic mechanisms.

In particular, the instantiation of devices occurs using a random selection mechanism following these steps:

- First, the system verifies whether there is remaining capacity in the current room to be filled.
- Then, it computes the number of minimum required devices needed for each room.
- If the remaining capacity is equal to the minimum required devices, no random extraction is performed and the minimum devices are instantiated.
- If, instead, the remaining capacity is greater than the minimum required devices, a random instantiation is performed among the sensor set.

Once a sensor is selected, the tool checks whether an available controller, capable of connecting to it exists. If no compatible controller is available, a **recursive upward instantiation procedure** is called: the algorithm moves upward across layers, attempting to instantiate a new compatible controller. If the instantiation of the controller requires additional devices in higher layers (e.g. routers or servers), the process continues recursively, creating the necessary infrastructure until all dependencies are satisfied.

Limitations of the Imperative Approach

The imperative approach presented significant challenges. The implemented mechanisms, upward recursive instantiation, device capacity checking, and protocol compatibility verification across layers, resulted in a complex codebase with tightly coupled components.

Several limitations distinguished among the others:

- **No backtracking mechanism**: During random selection, it can occur a resulting invalid configuration (e.g., room capacity violations, device compatibility conflicts). When this happens, the algorithm cannot backtrack to explore alternative allocations, instead produces a log file with the unsatisfied constraints.

- **Code complexity**: Although adding constraints and protocol compatibility is trivial, adding new features, such as optimization objectives, or advanced backtracking mechanisms, required modifying deeply nested recursive logic across multiple classes, making enhancements error-prone and difficult to debug.
- **Limited optimality**: The heuristic approach provided no guarantees about solution quality. The tool could generate valid topologies but had no mechanism to minimize resource usage.

Implementing a backtracking mechanism in the framework would have required substantial architectural changes: maintaining state at each decision point, tracking explored branches, and implementing rollback mechanisms. Given the code complexity, this enhancement would have further increased maintenance and made future extensions even more challenging.

3.5.2 Python Standard Library: Constraint Solving

The Python *constraint* library provides a simple backtracking constraint satisfaction solver. It supports domain definition, constraint addition and backtracking search. The main advantage is the simplicity of implementation. However, it has significant limitations for this use case:

- It is purely a constraint satisfaction solver, with no optimization capabilities.
- It does not support **parallel search**.
- Its performance drastically degrades as the number of variables grows.

For a topology with hundreds of sensors distributed across multiple rooms and floors, the backtracking search becomes infeasible within a reasonable time.

3.5.3 Mixed-Integer Linear Programming (MILP)

Mixed-Integer Linear Programming is a mathematical optimization framework where both the objective function and constraints are expressed as linear functions of decision variables, all, or some, of which are required to take integer values [22][23].

For the network synthesis problem described in Chapter 2, particularly in the approach proposed by [17], MILP is an optimal tool when every constraint can be expressed in a linear form. However, for the topology generation problem addressed in this thesis, several constraints have an inherent conditional structure, which makes them less suitable for MILP modeling.

Constraints such as **whitelist constraints**, used for protocol compatibility checking, can be modeled through a filtering step that avoids creating the corresponding connection variables in the model. Other constraints in the problem are naturally linear; for example, capacity constraints can be expressed as:

$$\sum_{\text{connections}} \text{assignment}[(\text{src}, \text{dst})] \leq \text{device\_limit}$$

which simply limits the number of connections a device can handle.

However, many constraints in the problem are conditional, meaning that they must be enforced only when certain conditions hold. **Conditional constraints** are more naturally expressed as logical implications rather than linear inequalities. Encoding them in MILP requires the introduction of auxiliary binary variables and "big-M" reformulations, increasing complexity and reducing the readability.

For instance, an implication of the form:

$$x = 1 \implies y \leq b$$

can be applied in MILP as:

$$y \leq b + M \cdot (1 - x)$$

where x is a binary variable and M is a sufficiently large constant [22]. Adapting this example to the topology generation problem; considering the device existence constraint governed by the *exists_var* binary model variable. A controller can have connections only if it is instantiated (*exists_var = 1*). This is expressed through the logical implication:

$$\text{exists\_var} = 0 \implies \sum_{\text{src}} \text{assignment}[(\text{src}, \text{ctrl})] = 0$$

In MILP, this implication must be reformulated using the big-M technique:

$$\sum_{\text{src}} \text{assignment}[(\text{src}, \text{ctrl})] \leq M \cdot \text{exists\_var}$$

where M is a sufficiently large constant. When *exists_var = 1*, the constraint says: "the number of connections must be smaller than a very big integer value", which is always satisfied if M is large enough, deactivating the constraint. When *exists_var = 0*, the constraint forces all connections to zero.

While this transformation preserves correctness, it introduces two main drawbacks:

- it increases the number of auxiliary variables and constraints required by the model;

- it reduces the readability and maintainability of the formulation, especially when many conditional constraints must be represented.

In topology generation problems such as the one considered in this thesis, where constraints frequently appear as conditional rules, the number of required big-M encodings can grow rapidly. This leads to more complex optimization models and potentially weaker solver performance. For these reasons, modeling the topology generation problem using MILP would require substantial reformulation effort and may result in models that are difficult to maintain and extend.

3.5.4 Google OR-Tools CP-SAT

The final solver being considered is the Google OR-Tools **CP-SAT (Constraint Programming with Boolean Satisfiability)** [24]. It is a hybrid solver that combines:

- Conflict-Driven Clause Learning (CDCL) from SAT solving [25][26]
- Constraint propagation from Constraint Programming [27]
- Optional Linear Relaxation from MILP [22]

CP-SAT is designed specifically for combinatorial optimization problems over discrete decision variables. Among its many advantages, CP-SAT offers:

- **Native Boolean variables**: Device existence decisions can be directly mapped to boolean decision variables (no Big-M linearization required).
- **Conditional Constraints**: Using the *OnlyEnforceIf* construct, in combination with Boolean variables, allows specific constraints to be enforced under certain circumstances, in a simple and expressive way.
- **Parallel search**: Allows to perform a multi-threaded search with shared clause learning [28]. Specifically, through the function *os.cpu_count()*, the solver automatically uses all available CPU threads.
- **Time-bounded solving with early termination**: The solver can return the first feasible solution found, suitable for problems where feasibility is the primary goal.

CP-SAT was selected as the solver for this thesis based on several characteristics of the topology generation problem. First, the majority of constraints involve Boolean conditions (device existence, room activation, protocol compatibility), which CP-SAT handles natively without requiring linearization techniques. Second,

the primary objective in most generation phases is to identify a feasible solution that satisfies all constraints, rather than proving optimality. Third, the problem involves discrete decision variables over combinatorial search spaces, which aligns with the design focus of constraint programming solvers. These characteristics make CP-SAT more suitable than MILP for this application, as documented in constraint programming literature [27][29].

3.6 Topology generator algorithm - Pipeline

The topology generation is orchestrated by the *HierarchicalTopologyBuilder* class. The pipeline is structured in four phases, each executed sequentially. As mentioned earlier in the Chapter, the design principle of this topology generator is that each layer has access to information about the previous one, enabling allocation without approximations. Each phase corresponds to a connection step between two adjacent layers of the BAS hierarchy. The phase sequence is defined through the *TopologyConfiguration* object, which specifies **source layer**, **target layer**, **connection type** and **phase-specific parameters**. This design makes it straightforward to add new layers or reorder phases without modifying the core solver logic.

The phases, in order of execution, are:

- **Phase 1**: Sensor allocation to rooms (field-layer structure building)
- **Phase 2**: Sensor-to-controller assignment (field-layer → automation layer)
- **Phase 3a**: ZigBee Router-to-ZigBee Coordinator assignment (protocol-specific connection)
- **Phase 3b**: Controller-to-Router assignment (automation layer → management layer)
- **Phase 4**: Router-to-Server assignment (management layer → server layer)

3.6.1 Protocol-Specific Intermediate Phases

The subdivision of Phase 3 into two sub-phases reflects a characteristic of building automation protocols: **different protocols have different hierarchical depths**. While BACnet devices follow a relatively flat structure (sensors, controllers and routers), ZigBee requires an additional intermediate layer between ZigBee routers and management-lever routers which is the ZigBee Coordinator.

To handle this situation while maintaining a unified pipeline, the framework adopts **intermediate phases**, managed by the same *execute_connection()* function as any other layer, that resolve protocol-specific hierarchies before converging to the common layer. Specifically:

- **Phase 3a** handles ZigBee-specific aggregation: ZigBee routers (which directly manage sensors) connect to ZigBee coordinators. This phase is executed only for ZigBee devices and has no effect on BACnet or other protocol paths.
- **Phase 3b** unifies all protocols at the management layer: both BACnet MS/TP controllers and ZigBee coordinators, connect to BACnet/IP routers.

3.6.2 Phase 1 - Sensor Allocation

Phase 1 determines how many sensors of each type are placed in each room of each floor. The sensor-to-controller connections are not established in this phase. The output of Phase 1 is a mapping from *(floor, room)* pairs to sensor type counts.

Rooms pre-allocation

To formulate the sensor allocation problem as a constraint satisfaction problem, the solver requires a finite set of decision variables representing potential room allocations. Since the exact number of rooms cannot be determined a priori, the system pre-allocates a set of possible rooms for each floor according to the expected capacity of each room (whose computation is described in Section 3.4.4). Since the calculation represents an upper bound, the solver typically allocates fewer rooms.

Each candidate room in the set is represented by a Boolean decision variable ***room_used[rm]*** managed by the ***RoomsHandler*** object. When the solver activates a room, it sets the *room_used* variable to 1 and keep the inactive ones to 0.

Phase 1 structure

The CP-SAT model for Phase 1 operates on integer variables, representing the count of sensors of a given type in a given room. For the handling of floor and rooms, the process starts from the *Connection Handler* object, described in Table 3.6:

Attribute	Description
model	This is the model variable, initialized at each phase
layers	Array that, for each phase of the generation, contains the set of source and target layer device types. Each layer is added through the method *addLayer()*, which also includes the initialization of device instances (those that the model will use to choose).
floor_handlers	Array containing, for each floor, the *FloorHandler* object, used to handle floor information (*floor_exists*, *floor_complete*, *number of rooms*, and *rooms_handler*). *RoomsHandler* is an object used to model the existence of each room.

Table 3.6: Connection Handler Object - Attributes Description

The *FloorHandler* object contains a *RoomsHandler* instance that manages the set of possible rooms for that floor. For each room, the handler maintains:

- **max_rooms**: Maximum number of rooms per floor
- **floor_num**: Floor id the handler has been assigned to
- **room_used**: Boolean variable indicating whether the room is active
- **min_devices_per_room, max_devices_per_room**: Capacity bounds for the room if activated

The model enforces the following constraints:

- **Zero sensors in unused rooms**: If *room_used[rm]* = False, the solver enforces that the sum of all sensors in that room must equal zero:

$$\text{room\_used[rm]} = 0 \implies \sum_{\text{sensor types}} \text{sensor\_vars}[fl, rm, \text{type}] = 0$$

- **Min/Max sensors per active room**: If a room is active, the total sensor count must fall within $[min\_devices\_per\_room, max\_devices\_per\_room]$
- **Required sensors by room type**: Each room is assigned a type (office, corridor, server room, etc.) at random according to a building type distribution.

- **Per type minimum counts**: The *constraints.json* minimum_devices constraints are applied per room.
- **Per type maximum counts**: The *constraints.json* maximum_devices constraints are applied per room.
- **Total sensors per floor**: The sum of all sensors across all rooms on a given floor must equal exactly *sensors_per_floor*, which is provided as an parameter.

Apart from the existence of a solution, the objective of this phase is to **minimize the number of active rooms**.

This task does not permit the solver to simply allocate all sensors to a single room. Each room has a strict minimum and maximum capacity constraints that must be satisfied if the room is activated. The room capacity factor parameter (*rm_dist*) influences this behavior by scaling maximum room capacities. Algorithm 1 presents the pseudocode for this phase.

Algorithm 1 Room and Sensor Allocation

Require: $floors$, $sensors\_per\_floor$, $sensor\_types$, $room\_constraints$
Ensure: $sensor\_allocation$, $room\_types$

Create CP-SAT model
for all floors f, rooms r **do**
    Assign random $room\_type[f, r]$ based on building distribution
    for all sensor types s **do**
        Create variable $x_{f,r,s} \in [0, max\_instances]$
    end for
end for
Add constraint: $\sum_s x_{f,r,s} = 0$ if room r unused, else $\in [min, max]$
Add constraint: $\sum_{r,s} x_{f,r,s} = sensors\_per\_floor$ for each floor f
Add constraint: Required sensors ≥ 1 per room type
Add constraint: Per-type min/max from $room\_constraints$
Minimize $\sum_{f,r} room\_used[f, r]$
Solve with time limit 60s
return sensor allocation per room

3.6.3 Phase 2 - Sensor-to-Controller Assignment

This phase connects the sensors allocated in Phase 1 to controller instances. Each sensor must be assigned to exactly one controller [1], and each controller cannot have more than *device_limit - 1* connections. Protocol compatibility is enforced through whitelist constraints. However, these constraints apply only to Access Control devices, which can connect exclusively to the Access Control Controller. All other sensors and actuators are assumed to be compatible with both BACnet and ZigBee protocols.

This is the most computationally expensive phase due to the high number of connections between sensors and controllers. To reduce the complexity of the model, Phase 2 applies a **floor-splitting strategy**, which divides the rooms on a floor into consecutive groups. Each group is then solved independently by a separate CP-SAT model.

After each sub-phase, the state is saved for the next ones (e.g. controller remaining capacity, last instance of already connected sensors), ensuring that by the end of the phase no sensor remains unconnected and all constraints are satisfied as if the entire process were performed in a single procedure. Controllers that still have available capacity are carried forward as *previous_phase2_result*.

The size of the split for sub-phases is controller by the parameter ***split_size***, whose optimal value range has been determined through testing, to find the best trade-off between computational overhead:

- A small value would result in fast model resolution time and faster constraint configuration for each sensor, but, for a large number of sensors it would create too many sub-phases, making the model setup and configuration a bottleneck.
- A large value would create less sub-phases but a slower configuration time and, most importantly, a slower resolution time.

The optimal trade-off for the *split_size* parameter is identified through the performance analysis in Chapter 5, where different values are evaluated. Based on that analysis, a value of 200 is shown to be the optimal one. Each sub-phase performs the same actions, with the exception of the first one, which uses a new array of controllers instead of *previous_phase2_result*.

The model is built as follows:

- **Assignment variables**: For each (sensor instance, controller instance) pair that passes the whitelist check, a CP-SAT boolean variable is created and stored in *assignment[(sensor,controller)]*.

[1] As stated in Section 3.4.2, this thesis does not consider link redundancy or alternative paths between sensors and other devices.

- **One-link only Constraint**: Each sensor instance must connect to exactly one controller:
$$\sum_{\text{ctrl}} \text{assignment}[(\text{s,ctrl})] = 1$$

- **Capacity constraint**: Each controller instance must receive between 1 and $device\_limit - 1$ sensors if it exists, or 0 if it does not exist:

$$1 \leq \sum_{\text{sensor}} \text{assignment}[(\text{sensor}, \text{ctrl})] \leq \text{device\_limit} - 1 \quad \Longleftrightarrow \quad \text{ctrl.exists\_var} = \text{True} \tag{3.9}$$

- **Sequential instantiation**: Controller instance $i + 1$ of a given type cannot exist unless instance i already exists and is already full.

The objective of Phase 2 is to find the first feasible solution ($stop\_after\_first\_solution = True$). A pseudocode of Phase 2 is provided in Algorithm 2.

Algorithm 2 Sensor to Controller Assignment

Require: phase1_result, controller_types, split_size, config
Ensure: controller_assignments, active_controllers

for all floors fl **do**
  Divide rooms into splits of size *split_size*
  $previous\_result \leftarrow None$
  for all splits **do**
    Compute *max_instances_needed*
    Create new CP-SAT model
    for all valid pairs $(sensor, controller)$ **do**
      Create BoolVar $assignment[sensor][controller]$
    end for
    for all sensors s **do**
      Add constraint: $\sum assignment[s][c] = 1$
    end for
    for all controllers c **do**
      Add constraint: $\sum assignment[s][c] \leq device\_limit - 1$
      if $c.exists = False$ **then**
        Add constraint: $\sum assignment[s][c] = 0$
      end if
    end for
    Add sequential instantiation constraint
    if $previous\_result \neq None$ **then**
      Carry forward non-full controllers
    end if
    Solve (first solution, time limit: 300s)
    Update controller usage
    $previous\_result \leftarrow current\_result$
  end for
end for
return controller_assignments, active_controllers

3.6.4 Phase 3 - Controller to Router Assignment

Phase 3 is divided into two different sections: the first, following the protocol structure, is dedicated to the connection of ZigBee Routers to ZigBee Coordinators, while BACnet controllers are not considered; the second, takes both BACnet controllers and ZigBee Coordinators and connects them to the router layer.

ZigBee Coordinator Assignment

This sub-section handles the ZigBee-specific sub-path. Only *zigbee_router* devices are eligible for this phase. Each ZigBee Router must connect to exactly one ZigBee Coordinator, and following the same pattern as Phase 2, each Coordinator can connect to multiple ZigBee routers below its capacity limit.

After this phase is solved, the resulting ZigBee Coordinators are **merged into the controller layer** using *_merge_layer_into_source()* method. This allows treating Coordinators as controllers connecting to routers in the following sub-section. Algorithm 3 provides the pseudocode for this sub-section.

Algorithm 3 ZigBee Router to Coordinator Assignment

Require: active_zigbee_routers, coordinator_types
Ensure: active_coordinators
Create new CP-SAT model
for all valid pairs $(router, coordinator)$ **do**
    Create BoolVar $assignment[r][c]$
end for
for all routers r **do**
    Add constraint: $\sum assignment[r][c] = 1$
end for
for all coordinators c **do**
    Add constraint: $\sum assignment[r][c] \leq device\_limit - 1$
end for
Add sequential instantiation constraint
Solve model
Merge coordinators into controller layer
return active_coordinators

Controller-to-Router Assignment

This sub-section connects the full controller layer (including ZigBee Coordinators) to the router layer. The whitelist constraints ensure that BACnet devices (BACnet MS/TP PLC Controller, Access Control Controller) connect only to BACnet/IP-to-MS/TP Routers, and that ZigBee Coordinators connect only to BACnet/IP-to-ZigBee Routers. Starting from this phase-on, every other connection is performed through the *execute_connection()* method, which applies the same assignment structure as Phase 2. However, since the number of devices at this level is far less than the sensor layer, splitting is not performed. Algorithm 4 provides the pseudocode for this subsection.

Algorithm 4 Controller to Router Assignment

Require: active_controllers_and_coordinators, router_types
Ensure: active_routers
Create new CP-SAT model
for all valid pairs $(controller, router)$ **do**
    Create BoolVar $assignment[c][r]$
end for
for all controllers c **do**
    Add constraint: $\sum assignment[c][r] = 1$
end for
for all routers r **do**
    Add constraint: $\sum assignment[c][r] \leq device\_limit - 1$
end for
Add sequential instantiation constraint
Solve model
return active_routers

3.6.5 Phase 4 - Router-to-Server Assignment

This phase connects routers to servers. Both servers types, Building Management Systems (BMS) and Energy Management Systems (EMS) accept BACnet/IP connections. This is enforced through the *lower_white_list* of the servers, which restricts incoming connections to the two router types only. The solver assigns each active router to exactly one server, which is also subject to capacity limits. Algorithm 5 provides the pseudocode for this phase.

Algorithm 5 Router to Server Assignment

Require: active_routers, server_types
Ensure: active_servers
Create new CP-SAT model
for all valid pairs $(router, server)$ **do**
    Create BoolVar $assignment[r][s]$
end for
for all routers r **do**
    Add constraint: $\sum assignment[r][s] = 1$
end for
for all servers s **do**
    Add constraint: $\sum assignment[r][s] \leq device\_limit - 1$
end for
Add sequential instantiation constraint
Solve model
return active_servers

3.7 Pre-Solving capacity Estimation

Before the solver is invoked, the system estimates the number of devices instances needed for each layer. This step is essential for two reasons:

- The CP-SAT solver requires a finite domain for each decision variable. Otherwise, the number of *exists_var* would be unbounded, making the problem unsolvable.
- As a consequence, if the number of needed instances is over-approximated the solving time may increase significantly due to the larger search space the solver must explore.

The capacity estimation proceeds layer-by-layer, starting from controllers upward to servers.

3.7.1 Controller Layer Estimation

The *calculate_controllers_needed_for_sensors()* function estimates the required number of controller instances per type:

- **General controllers (BACnet MS/TP PLC, ZigBee Router)**: The function assumes that 50% of all sensors will be served by each controller type, then divides this quantity by (device_limit − 1) to compute the number of controllers needed.

- **Access control controller**: The function estimates the number of rooms and multiplies by the expected number of access control devices per room, then divides by the controller capacity.

3.7.2 Coordinator, Router and Server Layer Estimation

The *compute_capacity()* function groups both source and destination devices by protocol. It then computes the minimum number of destination instances needed to serve all source devices with matching protocol compatibility, considering each destination device's *device_limit*.

Since these are only estimations, an upper bound safety margin is applied to ensure sufficient instances are available during solving.

3.8 Output and Export

If the processing pipeline executes successfully, the complete topology is assembled into a graph structure by the *TopologyExport* class which uses the *NetworkX* library to create an undirected graph G. The use of a graph library simplifies the exporting of the graph structure. The topology is then exported in one of three possible formats:

- **JSON file**: A structured representation of all device instances and connections, organized by layer. This file is used as the input to the access control policy generation phase described in Chapter 4.
- **XML**: It is a format derived from the JSON topology, intended for integration with external tools or simulation environments.
- **Visual graph**: A layered graph visualization is rendered using the *draw_layered_graph()* function, showing all device instances and their connections colored by layer. However, this visualization is primarily used for debugging and validation purposes; for topologies with hundreds of devices, it becomes difficult to read.

Each export includes metadata in addition to connectivity informations: seed used for the generation, number of sensors per floor (input parameter), number of total rooms, number of total devices and total connections. This allows any generated topology to be exactly reproduced by re-running the tool with the same seed and parameters.

Chapter 4

Access Control Policies generation

This chapter describes the architecture and implementation of the XACML-based access control policy generation system. The generator receives as input the network topology produced by the process described in Chapter 3, and generates a hierarchical set of access control policies that define all user interactions with the SB devices. The chapter begins with an introduction to XACML, the standard markup language for specifying access control policies, followed by a description of the three layers of the policy architecture (global, zone-based and role-based policies), the action intersection mechanism that determines permitted operations, and the XML policy generation pipeline.

4.1 XACML - eXtensible Access Control Markup Language

XACML is an OASIS standard (Organization for the Advancement of Structured Information Standards) for expressing and evaluating access control policies in an XML format [30] [31]. It provides a policy language that separates the specification part of the policies from the effective enforcement.

4.1.1 XACML Architecture

The XACML architecture is based on a **Policy Decision Point** (PDP), which is a component that evaluates access requests against a set of policies and returns authorization decisions. An access request consists of four different attributes:

- **Subject**: The entity requesting access (e.g. userID, role)

- **Resource**: The target of the access request (e.g. *device_id*, *device_type*, *room_type*, *floor_number*).
- **Action**: The operation to be performed (e.g. *read_value*, *set_temperature*, *unlock_door*)
- **Environment**: Contextual information (*current_time*, current day of week, emergency status)

The PDP evaluates the request by matching it with the policy set, applying combining algorithms to resolve conflicts between multiple applicable policies, and returning one of the four standard decisions: **Permit**, **Deny**, **Not Applicable**, and **Indeterminate**.

4.1.2 XACML Policy Structure

A XACML policy consists of a hierarchy of three elements:

- **PolicySet**: It is the root element that contains one or more Policies. A *PolicyCombiningAlgorithm* (*deny-overrides*, *permit-overrides*, *first-applicable*) defines how conflicting policy decisions are resolved.
- **Policy**: A Collection of Rules. Each Policy defines a *RuleCombiningAlgorithm* that determines how conflicting rules within the policy are resolved.
- **Rule**: It is the smallest decision element of the standard, it consists of a *Target*, *Condition* (Optional) and an *Effect* (Permit or Deny).

A Target specifies applicability conditions using *AnyOf* and *AllOf* constructs, each containing a *Match* element that compare attribute values using matching functions (defined by the standard). A condition allows more complex Boolean expression using *Apply* functions.

4.1.3 Combining Algorithms

XACML defines standard combining algorithms for resolving conflicts when multiple policies or rules are applicable to a request.

- **Deny-Overrides**: If any policy or rule evaluates to Deny, independently from other results, the overall result is Deny. This algorithm is commonly used for security-critical contexts, where everything should be denied by default.
- **Permit-Overrides**: If any policy or rule evaluates to Permit, independently from other results, the overall result is Permit.

- **First-Applicable**: The result is the decision of the first policy or rule which applies in the sequence.
- **Deny-Unless Permit**: The result is always Deny unless there's at least one policy or rule that evaluates Permit.

In this thesis, the PolicySet uses a **deny-overrides** combining algorithm, following a security principle where explicit denials always take precedence over any permits. This ensures that security restrictions cannot be accidentally bypassed by conflicting role-based permissions.

4.2 Smart Building Policy Model

The access control model used in this thesis was developed independently and defines a set of attribute identifiers and a hierarchy composed of three layers. This model aligns with the requirements and objectives of this thesis, making it suitable for generating access control policies from the synthetic topologies.

4.2.1 Custom Attribute Schema

The attribute schema follows a hybrid approach: the XACML attribute categories and their identifiers are fixed to ensure policy consistency and enable automated validation, while the attribute values are fully customizable through external JSON configuration files.

The schema was designed based on three criteria:

- **Alignment with BAS architecture**: Attributes map directly to the three-layer topology structure (field, automation, management), with `resource:floor`, `resource:roomType`, and `resource:deviceType` reflecting the hierarchical organization of generated topologies.
- **Role-based access control support**: The `subject:role` attribute enables organizational role definitions (Employee, Manager, Administrator, Technician, Security, Customer), while `action:action-id` maps to device-specific capabilities defined in the capability configuration file.
- **Temporal and contextual constraints**: Environment attributes (`isEmergency`, `currentTime`, `currentDay`) support time-based restrictions and emergency override scenarios common in building automation security policies.

Customization is achieved through three JSON configuration files:

- **roles.json**: Defines subject roles, their associated capabilities, admitted/prohibited room types, and temporal restrictions
- **capabilities.json**: Maps capability groups to device action sets (e.g., `employee_control` includes `set_temperature`, `read_status`)
- **devices.json**: Specifies device types and their supported actions.

This design allows organizations to adapt the policy model to their specific requirements (different roles, custom capabilities, organization-specific room classifications) without modifying the policy generation code or the core XACML schema structure.

The policy generation system extends the standard XACML attribute categories with SB-specific attributes:

Category	**Attribute ID**	**Description**
Subject	subject:role	Role assigned to the user (Employee, Manager, Administrator, etc.)
Resource	resource:deviceID	Unique identifier of the device instance
Resource	resource:deviceType	Unique identifier of the device instance
Resource	resource:roomType	Room type (OFFICE, LAB, SERVER_ROOM, etc.)
Resource	resource:floor	Floor number where the device is located
Action	action:action-id	Operation requested (read_value, set_temperature, unlock_door, etc.)
Environment	environment:isEmergency	Boolean flag indicating emergency status
Environment	environment:currentTime	Current system time
Environment	environment:currentDay	Current day of week (monday, tuesday, ...)

Table 4.1: Access Control Policies - Custom Attribute Schema

Three-tier Policy Hierarchy

The policy set is structured as a three-tier hierarchy, evaluated in order of precedence:

- **Global Policies**: This is a subset of policies that applies generically to any room and for any actions. It defines emergency full access to two specific roles: Administrators and Security Personnel. Furthermore, it denies camera access in washrooms.
- **Zone-based Policies**: These policies define the privileges to specific roles in specific areas (room types). Zones are classified as public (open workspaces, reception, lobbies, corridors, washrooms, cafeterias), restricted (offices, meeting rooms, conference rooms, labs, kitchens) and critical (server rooms, storage rooms). Public zones permit read access to all roles, while, critical zones restrict access to administrators and security.
- **Role-based Policies**: These are the most common policies. They concern role-specific policies generated for each user role (defined in *roles.json*. For each role, the policy specifies permitted device types, allowed actions, admitted rooms, prohibited rooms and temporal restrictions.

4.3 Policy Generation Pipeline

The policy generation process is handled by the *HierarchicalPolicySetGenerator* class, which coordinates three policy generators, each for a specific context: *GlobalPolicyGenerator*, *ZonePolicyGenerator* and *RoleBasedPolicyGenerator*.

4.3.1 Input and Initialization

The policy generator receives four inputs:

- **Topology**: The JSON representation of the generated topology from the first phase of the pipeline (see Chapter 3), containing each floor and room structure with device placement.
- **Role configuration**: A JSON file defining the organizational **roles** within the building (e.g. Employee, Manager, Administrator, Technician, Security, Customer). Each role specifies:
  - *Capabilities*: Capabilities (e.g., *employee_control*)
  - *Admitted rooms*: Room types the role can access (e.g., OFFICE, MEETING_ROOM)

  - *Prohibited rooms*: Room types explicitly denied (e.g., SERVER_ROOM)
  - *Restrictions*: Temporal constraints (allowed days, hours) and permission requirements

- **Capability configuration**: A JSON file containing capabilities. Each capability contains a list of allowed actions. For example, the ***employee_control*** capability includes actions like ***set_temperature***, along with constraints like temperature range (18-26°C). This structure allows roles to inherit predefined action sets rather than specifying individual permissions for each device type.

- **Devices**: An array containing metadata for all device types in the system, including supported actions for each device. This information is used to validate that requested capabilities match actual device functionality.

The role and capability configurations are provided as external inputs rather than automatically generated. This design choice reflects several practical considerations:

1. **Domain expertise requirement**: Role definitions and capability assignments reflect organizational policies, and operational procedures that cannot be inferred automatically. For instance, determining whether technicians should have emergency override access or whether customers can access certain zones requires domain knowledge about the organization's security policies.

2. **Validation and testing**: During the development and validation phase of this thesis, using fixed, manually defined configurations allowed systematic testing of the policy generation pipeline.

3. **Customization**: External configuration files permit one to adapt the policy model to specific requirements without modifying the generation code. Automatic generation would impose a single, potentially inappropriate, organizational structure.

Future work could explore automatic role suggestion based on topology characteristics, but the final role definitions should remain human-validated to ensure compliance with organizational security policies.

The topology is extracted from the file and processed to extract connection information, and passed as input to the zone and role generators, which access it to determine which room types and device types are actually present in the topology.

4.3.2 Action-Intersection Mechanism

Most of the computation for the generation of the access control policies comes from the role-based policy generation. In particular, the core of this component is the **action intersection** algorithm, implemented in the *_generate_permit_rules_by_type* method of the *RoleBasedPolicyGenerator* class.

For each role, the process starts from the array of admitted rooms defined in *roles.json*. For every room in this set, all device types present in the room are examined. The intersection between the capabilities of the entity and the allowed actions of each device is then computed. A pseudocode representation of this process is reported in Algorithm 6.

Algorithm 6 Access Control Policies - action intersection

Require:
- *role* (user role)
- *device_type* (device type identifier)
- *devices* (device configuration)
- *capabilities* (capability configuration)

Ensure: *allowed_actions* (set of permitted actions for the role on the device type)

Extract device actions: $device\_actions = devices[device\_type].actions$
Initialize *allowed_actions* (empty)
for all $capability \in role.capabilities$ **do**
    $capability\_actions \leftarrow capabilities[capability].capabilities$
    if "all" $\in capability\_actions$ **then**
        $allowed\_actions \leftarrow device\_actions$
        break ▷ No need to check further capabilities
    else
        $allowed\_actions \leftarrow (device\_actions \cap capability\_actions)$
    end if
end for
return *allowed_actions*

This approach ensures that a role can perform only actions both defined by the device and allowed by its capabilities. For example, if an Employee has *employee_control* capability (which includes *set_temperature*), but temperature sensor does not support it, the policy will not contain it.

4.3.3 Global Policy Generation

The *GlobalPolicyGenerator* class produces a policy that applies across the entire building and it is always the same independently of the generated topology.

Emergency Override Policy

Grants full access to Administrator and Security roles when the environment attribute *"isEmergency"* is True. The entities are checked through the *Target* element, and the environmental variable through the *Condition* element. The rule

combining algorithm is *permit-overrides*, ensuring that if any of the two roles is true and the state is *emergency*, the result is Permit.

4.3.4 Zone-based Policy Generation

The *ZonePolicyGenerator* classifies rooms in three zones:

- **Public**: *open_workspace*, *reception*, *lobby*, *corridor*, *washroom*, *cafeteria*
- **Restricted**: *office*, *meeting_room*, *conference_room*, *lab*, *kitchen*
- **Critical**: *server_room*, *storage_room*

The generator looks at the topology to determine which room type is actually present in the building (not all topologies include all room types). It generates two policies:

Public Zone Policy

Permits any role to perform actions starting with "read_" on devices located in public zones. The Target uses a *string-starts-with* match function to include all read operations (*read_value*, *read_status*, *read_history*, etc.).

Critical Zone Policy

Restricts access to critical zones to Administrator and Security roles only. The rule combining algorithm is *deny-unless-permit*, meaning that access is denied unless a rule explicitly permits it (protecting critical zones even if no role-based policy applies).

4.3.5 Role-based Policy Generation

For each role defined in *roles.json*, the *RoleBasedPolicyGenerator* produces a policy containing two sets of rules.

Permit Rules

The generator iterates over all device types present in the rooms into which a user is allowed (information provided in *roles.json* in the field *admitted_rooms*). For each device type, it computes the allowed actions using the intersection mechanism described above, then generates a PERMIT rule with a Target that matches:

- Subject role = *role_name*
- Resource device type = *device_type*

- Resource room type = *admitted_rooms*
- Action = *allowed_actions*

If the role has temporal restrictions, the rule includes a Condition that checks the environment attributes *currentTime* and *currentDay* using XACML's *time-in-range* and *string-equal* functions.

Deny Rules

For each room type in the role's *prohibited_rooms* list, the generator creates a DENY rule with a Target matching the role and the prohibited room types. This rule has no Condition (if a role denies access to a room, regardless the action, that entity is not allowed to that room).

4.3.6 Policy Set and XML generation

The *generate_policy_set* method constructs the entire PolicySet by collecting all the generated policies in the previous phases. Each policy is equipped with an *order* field, defined as an attribute of the class *Policy*. The entire hierarchy of policies is unified in a unique Policy set which uses a *deny-overrides* combining algorithm, ensuring that DENY decisions from global and zone-based policies take precedence over the last role-based PERMIT decisions. Each *Policy* object is serialized into the XML format using the built-in *to_xml()* method, and the final policy set is written in the *policy_set.xml*. Global and Zone-based policies are directly inserted inside the PolicySet, while, the Role-based policies are referenced through the *PolicyIdReference* component, coming from standard XACML, and identified through a unique urn (Uniform Resource Name) identifier.

4.4 Policy application example

To illustrate the policy generation output, we show a simple example of PolicySet, generated starting from a SB topology containing 100 sensors. For sake of readability a few parts of the policies have been removed since they are not essential for the example. Consider the case where an Employee requests to *read_value* of a temperature sensor located in an OFFICE room at 10 am on Monday.

- **Step 1**: The set of global policies is checked. The subject is Employee, which does not fit Administrator nor Security, so the result is *Not Applicable*.
- **Step 2**: The algorithm checks whether "OFFICE" belongs to the set of critical zones or to the public zones. Since it does not belong to either, both zone-based policies return *Not Applicable*

- **Step 3**: The algorithm then checks, in the role-based policies list, whether Employee corresponds to a role for which a policy has been created. As reported in Listing 4.2 this is indeed the case. Therefore, the algorithm checks the list of parameters: *subject role (employee)* , *device type (temperature_sensor)*, *room type (office)*, *action (read_value)*. Then, it also checks whether the currentTime is in the interval [8am, 8pm] and currentDay is in [mon,...,fri].

Comparing the parameters from the request with the values in the policy, the policy evaluates to PERMIT. The PolicySet combining algorithm (*combine-overrides*) combines the results (Not Applicable, Not Applicable, Not Applicable, PERMIT) and evaluates to PERMIT. However, if the employee changes the request to 10pm, the Condition would not be satisfied, hence the final decision would be DENY.

```
<PolicySet ...
PolicyCombiningAlgId="urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:deny-overrides">
  <Policy PolicyId="urn:oasis:names:tc:xacml:3.0:example:policyid:global-emergency-override" RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permit-overrides" Version="1.0">
    <Rule RuleId="emergency-security-admin" Effect="Permit">
      <Target>
        <AnyOf>
          <AllOf>
            <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">security</AttributeValue>
              <AttributeDesignator Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject" AttributeId="urn:smartbuilding:subject:role" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
            </Match>
          </AllOf>
          <AllOf>
            <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">administrator</AttributeValue>
              <AttributeDesignator Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject" AttributeId="urn:smartbuilding:subject:role" DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
            </Match>
          </AllOf>
        </AnyOf>
      </Target>
      <Condition>
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:boolean-equal">
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:boolean-one-and-only">
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-category:environment" AttributeId="urn:smartbuilding:environment:isEmergency" DataType="http://www.w3.org/2001/XMLSchema#boolean" MustBePresent="false" />
          </Apply>
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#boolean">true</AttributeValue>
        </Apply>
      </Condition>
    </Rule>
```

```
  </Policy>
  <PolicyIdReference>urn:oasis:names:tc:xacml:3.0:example:policyid:role-Employee</
    PolicyIdReference>
</PolicySet>
```

Listing 4.1: Emergency Override Policy (Global)

```
<Policy
...
PolicyId="urn:oasis:names:tc:xacml:3.0:example:policyid:role-Employee"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny
    -overrides">
  <Target />
  <Rule RuleId="Employee-admitted-rooms-actions" Effect="Permit">
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    Employee</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:1.0:subject-
    category:access-subject" AttributeId="urn:smartbuilding:subject:role" DataType
    ="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
          </Match>
        </AllOf>
      </AnyOf>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    temperature_sensor</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-
    category:resource" AttributeId="urn:smartbuilding:resource:deviceType"
    DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
          </Match>
        </AllOf>
      </AnyOf>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    OFFICE</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-
    category:resource" AttributeId="urn:smartbuilding:resource:roomType" DataType=
    "http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
          </Match>
        </AllOf>
      </AnyOf>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    read_value</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-
    category:action" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
    DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
          </Match>
        </AllOf>
      </AnyOf>
    </Target>
    <Condition>
```

```
41      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
42        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-in-range">
43          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-
    only">
44            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-
    category:environment" AttributeId="urn:oasis:names:tc:xacml:1.0
    :environment:current-time" DataType="http://www.w3.org/2001/XMLSchema#time"
    MustBePresent="false" />
45          </Apply>
46          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">08
    :00:00</AttributeValue>
47          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">20
    :00:00</AttributeValue>
48        </Apply>
49        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
50          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-
    only">
51            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-
    category:environment" AttributeId="urn:smartbuilding:environment:current-day"
    DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent="false" />
52          </Apply>
53          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
    monday</AttributeValue>
54          ...
55        </Apply>
56      </Apply>
57    </Condition>
58  </Rule>
```

Listing 4.2: Role-based policy

Chapter 5

Performance Evaluation & Results

This chapter presents a comprehensive analysis of the topology generation system's performance. The evaluation examines how varying the configuration parameters: network size, split size, room capacity factor and building floors, affect the execution time and the overall system efficiency.

The analysis is organized in four main sections, each focusing on a specific aspect of the system performance:

- **Scalability Analysis**: Examines how execution time grows with network size across different parameters combinations.
- **Split Size impact**: Analyzes the effect of split_size parameter over performance, evaluating which is the best choice according to the building size.
- **Phase analysis**: Provides detailed timing analysis for each phase, showing where the most computational overhead is located.
- **Room Capacity Factor effect**: Analyzes which is the impact of room distribution parameters on the resulting topology structure (number of active rooms) and performance (execution time).

The expected results for this analysis should demonstrate linear scaling with respect to sensor count, as each additional sensor introduces a proportional increase in problem complexity. However, increasing the number of floors is expected to exhibit some increases in total generation time. This occurs because the floor-by-floor construction process incurs cumulative overhead: each floor requires independent variable instantiation in the constraint solver, graph structure construction, and state management operations. While the constraint solving time per floor remains

constant for a fixed sensor count, the cost of these operations grows with building height. Additionally, we expect the room capacity factor to have minimal impact on performance, enabling users to freely adjust sensor distribution characteristics without degrading generation efficiency.

As regards the phase execution, we expect that, since most of the devices is handled in the first phases, the main overhead is concentrated in that section. As described in the previous chapters, the value of the split_size parameter has been found experimentally since a trade-off must be found between solver resolution time and data management overhead.

5.1 Experimental Setup

All experiments were conducted on a laptop equipped with an AMD Ryzen 7 5700U CPU with 8 cores running at 2.60GHz, 16 GB DDR4 3200MHz RAM and Windows 11 as operating system. To ensure statistical stability every configuration attempt has been executed three times, each with a different seed, to make sure the results were coherent. All the results presented in this chapter represent the mean values across the three runs.

The experimental testbed consisted of 540 individual test cases, each varying four key parameters: **number of floors**, **sensors per floor**, **room capacity factor** and **split size**, as showed in Table 5.1. The complete test suite required 354 minutes of total execution time with an average CPU temperature of 80 °C so no thermal throttling has occurred during the execution. Each experiment generated a detailed **timing data report** across all five generation phases, along with **structural informations** described in Table 5.2.

Parameter	**Range**
Number of floors	1, 2, 3, 4
Sensors per floor	1000, 1500, 2000, 3000, 4000
Room distribution	0.1, 0.5, 1.0
Split size	100, 200, 300

Table 5.1: Experimental parameter ranges

Parameter	Description
experiment_id	Unique identifier for each run. The complete dataset comprises 540 test cases.
timestamp	Execution timestamp for the test. The complete experimental suite required 354 minutes to complete.
seed	Randomization seed used for the constraint solver to ensure reproducibility.
floors **sensors_per_floor** **room_distribution** **split_size**	Parameters subject to iteration in the testbed.
total_time **phase1_time** **phase2_time** **phase3_time** **phase4_time**	Time traced for each phase and total execution time in seconds.
total_rooms **active_rooms** **total_devices** **total_connections**	Topology structure metrics: total and active rooms, total devices, and total connections for the specific run.
success **error_message**	Values used to identify whether a solution has been found. The final result comprises 533 successful cases and 7 fails. Hence, the success rate is 98.7%

Table 5.2: Experimental data parameters collected for performance analysis.

5.2 Scalability Analysis: Impact of network size

5.2.1 General overview analysis

The **scalability** of the topology generator is a critical aspect for realistic deployments and practical use cases, as it determines whether the tool is applicable to any building sizes, keeping the execution time within feasible limits. Figure 5.1 presents a comprehensive visualization of the experimental results, showing how execution time varies as network size increases under different combinations of split size and room capacity factor (*rm_dist*) parameters. For each subplot, the y-axis represents the total execution time, while the x-axis shows the number of sensors per floor. Each row keeps the split_size fixed while varying the room capacity factor.

The first analysis evaluates how the execution time scales as the network size

increases. Network size is measured in terms of both the number of sensors per floor and the number of floors, under different combinations of split size (100, 200 and 300) and room distribution parameters (0.1, 0.5 and 1). This analysis allows one to evaluate whether the algorithm is effectively scalable, and identifies potential performance bottlenecks by increasing the size of the building.

Figure 5.1 presents six subplots, each representing the **Total Generation Time** (in seconds) as the number of sensors per floor increases. In particular, as expressed earlier, the experiments consider four floors configurations: one, two, three and four floors. Each row of the figure corresponds to a fixed split_size, while the room capacity factor changes between 0.1, 0.5 and 1.

The most consistent pattern across all configurations is the approximately linear relationship between the number of sensors and execution time within each floor configuration. Considering the best-performing configuration ($split\_size = 200$), the single-floor building (blue line) exhibits the lowest slope. The execution time grows from 2.5 seconds for 1,000 sensors to almost 15.40 seconds for 4,000 sensors. This indicates that, for a single floor, the execution time slightly increases when increasing the number of sensors. However, as the building complexity increases, the trend becomes steeper, but execution time still remains manageable even for larger configurations:

- Two-floor buildings (red line) require approximately double the execution time of the single-floor configuration. This behavior is expected, since the total number of sensors doubles. The execution time increases from 4.20 seconds for 1,000 sensor to 41 seconds for 4,000 sensors, maintaining a ratio of approximately 2:1 with respect to the single-floor configuration.

- Three-floor buildings (pink line) exhibit the same trend. Execution time ranges from 7.30 seconds for 1,000 sensors to 80 seconds for 4,000 sensors, resulting in a ratio close to 2:1 compared with the two-floor configuration and 4:1 compared with the single-floor case.

- Four-floor buildings (cyan lines) show the steepest trend, starting from approximately 11.30 seconds for 1,000 sensors and reaching 140 seconds for 4,000 sensors, corresponding to roughly a 7:1 ratio compared with the single-floor configuration.

Analyzing the slope ratios reveals an interesting pattern. When the number of floors increases, the execution time grows proportionally as well, maintaining a near-constant ratio of 2:1 between consecutive floor configurations. While the relationship remains close to linear up to three floors, the four-floor configuration begins to show super-linear behavior. This effect is expected, since increasing the number of floors not only adds more devices but also introduces additional overhead due to constraints related to the inter-floor connectivity and sequential

instantiation of network components. Despite the increase in complexity, the overall execution time remains practical for offline topology generation. Even the most complex configuration considered in this study (4 floors, 16,000 sensors) completes in approximately 140 seconds.

Figure 5.2 further confirms this behavior. When the number of floors is fixed and only the number of sensors increases, the execution time grows approximately linearly. This result is consistent with the initial expectations, as increasing the number of sensors introduces additional complexity but does not significantly alter the constraint structure.

Table 5.3 summarizes the most impacting testing cases and compares these results by providing the ratios with respect to the first floor solution and the previous one.

Floors	**Time - 1k**	**Time - 4k**	**Ratio vs 1 Floor**	**Ratio vs prev. case**
1	2.5 s	15.40 s	1.0x	–
2	4.2 s	41 s	2.5x	2.5x
3	7.3 s	80 s	5.0x	2.0x
4	11.3 s	140 s	**9.0x**	**1.75x**

Table 5.3: Execution time and scaling ratios by floor count (*split_size=200*)

5.2.2 Impact of Split Size

In Figure 5.1 it is also revealed the effect of split size on performance. The split size parameter controls how the *sensor-to-controller phase* divides the problem into smaller subproblems:

- *Split size 100*: Produces more granular partitions with a higher number of splits to handle, but generally with a faster resolution time.
- *Split size 200*: Provides balanced partitioning that appears optimal for most configurations.
- *Split size 300*: Creates larger subproblems with fewer splits but more complex solver instances.

The performance difference between split sizes is not pronounced for simple buildings (1-2 floors) but becomes more evident for larger architectures. For instance, with 4,000 sensors and 4 floors, split size 200 achieves approximately 140 seconds, while split size 100 takes 220 seconds. In Section 5.3 it is deeply analyzed the effect of *split_size* parameter and identifies the optimal value.

5.2.3 Room Capacity Factor effects

The Room Capacity factor is a value between 0 and 1 that determines the room capacity during topology generation. A value close to 0 sets the room capacity near the minimum values (determined by the minimum devices constraint defined in *constraints.json* configuration file), while a value close to 1 sets the room capacity near the maximum (computed as the sum of the maximum instances for each sensor type in a room, defined by the *max_sensors_needed* variable, while still respecting the maximum devices constraint defined in *constraints.json* configuration file).

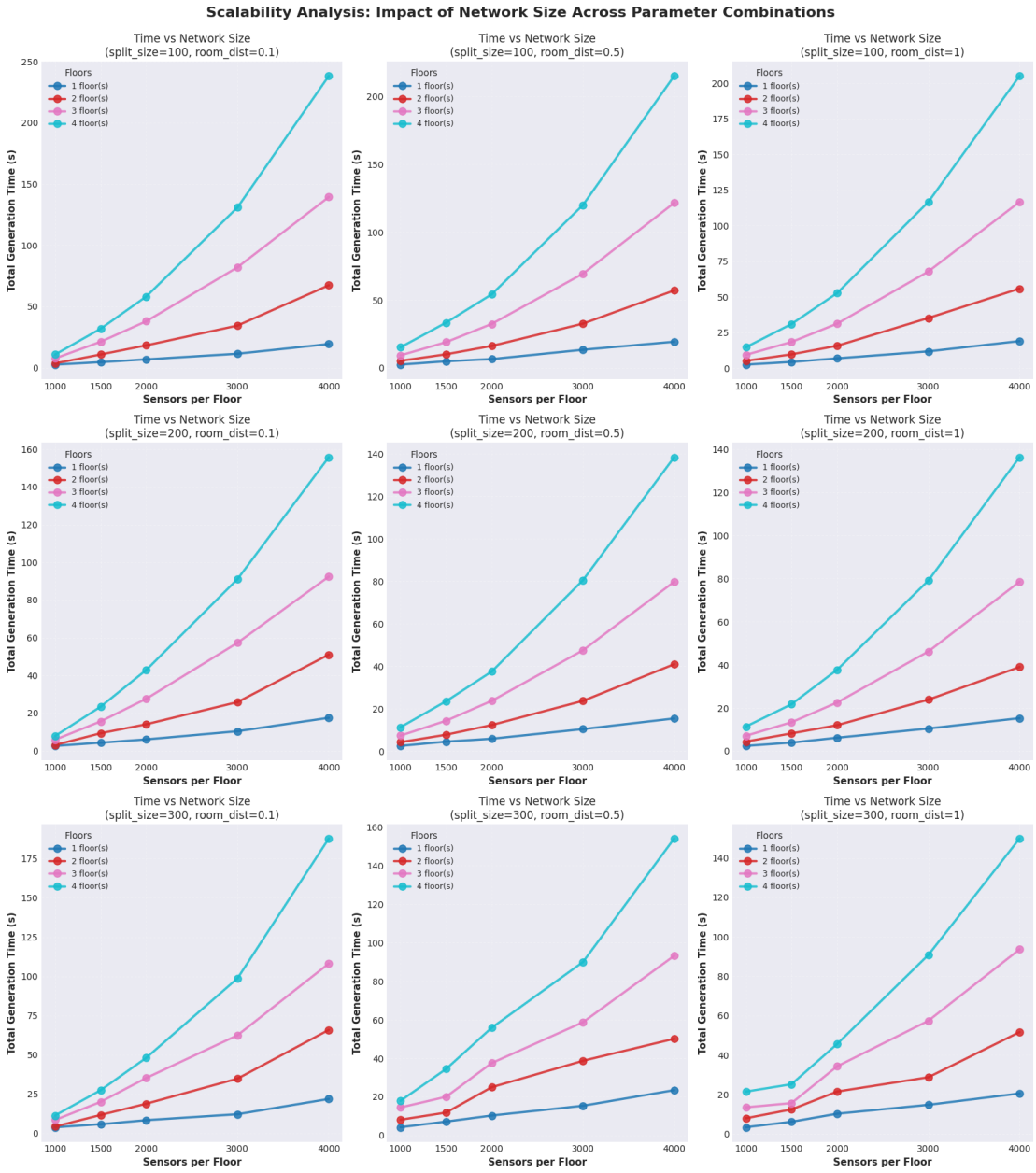

Figure 5.1: Scalability analysis: generation time vs network size. Combination of *split size* (100, 200, 300) and *room capacity factor* (0.1, 0.5, 1).

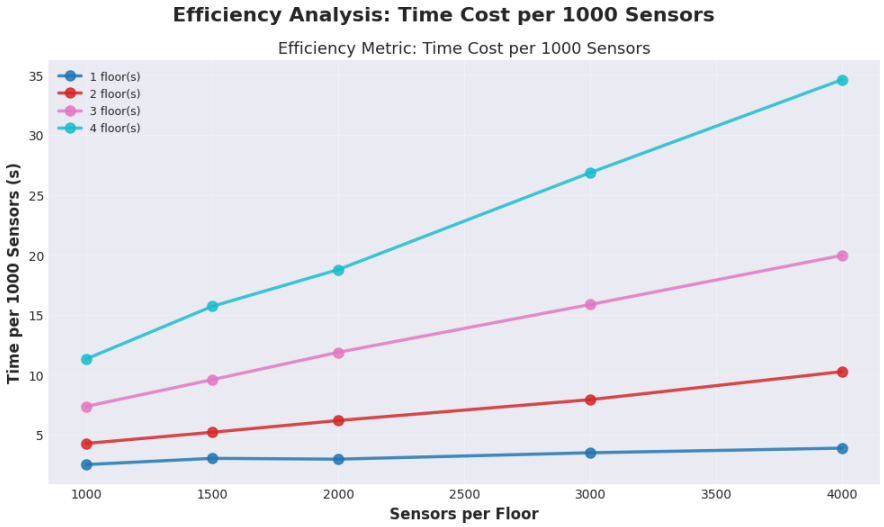

Figure 5.2: Scalability analysis: Time per 1000 sensors (split_size=200, rm_dist=0.5)

5.3 Split size analysis by Building Complexity

While Section 5.2 examined split size through different plots, this section provides a focused analysis on how split size selection affects performance by directly analyzing it. Figure 5.3 presents two views that explicitly define the relationship between split size and performance. More precisely, the left panel shows average performance trends across all sensor counts, while the right one presents thee speedup ratios achieved across the three split size configurations.

5.3.1 Average Performance across sensor counts

Looking at the left panel of Figure 5.3, it is visible that larger split sizes do not consistently improve performance, especially for small building architectures.

However, as building complexity increases, performance differences become more pronounced. Apart from the simplest implementation (1 floor), whose average total time remains approximately constant across all split sizes, starting from two-floor implementations and extending to more complex architectures, the best average total time is always achieved at split_size=200.

The same result can be observed from the speedup analysis in the right panel, which quantifies relative performance using split_size=200 as the baseline for computing ratios with the other two cases. Values above the 1.0 line indicate faster executions, while values below indicate slower execution. It is clear that, starting from single-floor topologies and extending to more complex configurations, every

considered case lies below the reference value, confirming that the best split value is 200.

Figure 5.3: Split size analysis: Performance by floors

5.4 Phase-by-Phase performance analysis

The following section aims to understand how execution time is distributed across the five-phase topology generation pipeline to identify optimization opportunities. Figure 5.4 presents both absolute and relative execution times divided by each phase.

Despite the logical distinction introduced in Chapter 3 between Phase 1 (sensor-allocation) and Phase 2 (sensor-to-controller assignment), in the real implementation these two phases are merged into a single step since they involve the same elements. As a result, their execution time is measured as a single combined phase since the sensor allocation phase has a negligible execution time thanks to counter allocation technique explained in Section 3.6.2 which reduces the complexity of the task.

5.4.1 Time Distribution across Phases

Looking at the left bar chart and the percentage plot at the right, both representing the total execution time for each floor configuration while fixing the number of sensors per floor number to 3,000, it becomes clear that the sensor-handling phase is the most overheading phase:

- **Single-floor**: Phase 1 accounts for 66.2% of total time
- **Two-floor**: Phase 1 presence increases to 90.2%
- **Three-floor**: Phase 1 raises to 93.9%

- **Four-floor**: Phase 1 is 94.7%

In contrast, the remaining phases consume between 6 and 40% collectively of the total execution time.

These results indicate that:

- The sensor handling phase is the only performance-sensitive phase affected by the number of sensors per floor since the others minimally change.
- Device-to-Device allocation phases (excluding sensors-to-controllers) are already efficient.
- As the topology size increases, the dominance of the sensors-handling phase becomes even more dominant.

5.4.2 Phase scaling

When looking at the percentage plot, it is not clear what the trend is for each phase by increasing the complexity of the topology. Indeed, it might seem, as said before, that only the sensor-related phase is affected by the complexity of the topology. However, looking at the bottom pair of plots in Figure 5.4, which show the scaling trends of Phase 1 and Phase 2 (controller-to-router), a similar behavior can be observed for both phases.

Both plots show the exact same pattern analyzed in Section 5.2: increasing the complexity of the topology, the execution time increases following a non-linear trend, but still keeping feasible execution times. The key observation is that this non-linear trend is not limited to the sensor-related phase but it is also present in the other phases. The difference is that their execution time is so small that the effect is practically negligible.

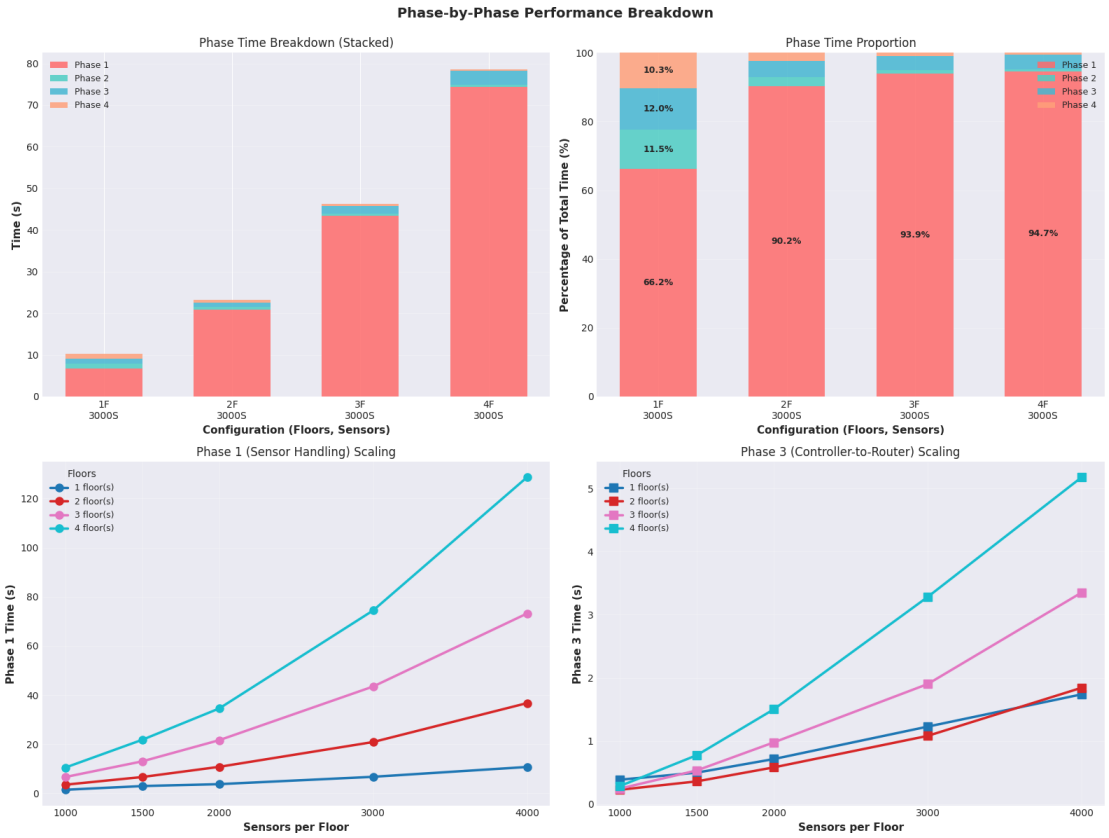

Figure 5.4: Phase breakdown analysis

5.5 Room capacity factor Impact

The room capacity factor parameter controls how sensors are allocated across building rooms, trading off between many small rooms and fewer large rooms. Figure 5.5 examines the parameter's impact on both topological structure and execution performance.

5.5.1 Room usage and Sensor Density

As already defined, the room capacity factor is a real number between 0 and 1 whose value contributes to determine the maximum theoretical capacity of a room. Basically, this value is used in the computation of *maximum_room_size* parameter as a multiplying factor of the total number of devices instantiable in a room.

From the top-left panel of Figure 5.5, this behavior is clearly visible: starting from the lowest value (0.1, very sparse), moving to 0.5 (sparse) and reaching the maximum value 1.0 (dense) the **room usage distribution** is defined by a

decreasing trend. This indicates that the number of used rooms decreases by increasing the room capacity factor.

More precisely, Table 5.4 shows the results obtained by fixing the number of floors to 2 and *split_size* to 200. It is noticeable a substantial change, especially in the initial range of values:

- 4,000-sensor deployments: the room count decreases from 365 to 200 (-45%) when moving from 0.1 to 0.5, reaching 140 (-61.5% reduction) at 1.0
- 3,000-sensor deployments: the room count decreases from 275 to 145 (-47%) when moving from 0.1 to 0.5, reaching 105 (-61.8%) at 1.0
- 1,000-sensor deployments: the room count decreases from 100 to 50 (-50%) when moving from 0.1 to 0.5, remaining at 50 (-0%) at 1.0

Room Capacity Factor	**Number of Active Rooms by Sensor Count**				
	1,000	**1,500**	**2,000**	**3,000**	**4,000**
0.1	100	140	180	275	365
0.5	50	75	100	145	200
1.0	50	55	65	105	140

Table 5.4: Number of active rooms vs sensor count for different Room Capacity Factors

The reduction percentage increases with network size, indicating that the room capacity factor becomes more impactful for larger deployments. This linear and predictable relationship allows designers to precisely control building layout by reducing room usage up to 60% in large topology architectures.

An interesting result emerges from the top-right panel, which reveals the convergence of the room allocation. Despite varying sensor counts (1,000 to 4,000 sensors per floor), all configurations converge toward a target sensor density per room for any given room distribution value. For example, with rm_dist = 0.5, all sensor counts stabilize around 40 sensors per room, regardless of the total size of the network.

This convergence demonstrates an important property: the constraint solver dynamically adjusts the number of active rooms to maintain a target density determined by the room capacity factor parameter.

This behavior reflects the constraint formulation, where room count acts as a dependent variable. The solver minimizes room usage while satisfying minimum and maximum device constraints, reaching a balanced sensor density. The consistency of this density across different scales validates the correctness of the implementation:

multiplying the average sensors per room (40) by the number of active rooms for the 4,000-sensor case (200 rooms) yields 8,000 total sensors across two floors, matching the expected value.

This self-organizing property ensures that the generated topologies maintain realistic patterns of occupancy of the room regardless of the building scale.

5.5.2 Performance Invariance

The most relevant finding from Figure 5.5 is revealed in the bottom two panels. Both represent the performance of the tool, expressed in Total Generation Time. In the left plot the number of floors is fixed to 2 and split_size to 200, while the number of sensors per floor varies. In the right plot the number of sensors is fixed to 1,500 and split_size to 200, while the number of floors changes.

Both plots show the same final result: execution time remains approximately invariant with respect to room distribution. Whether analyzing performance by sensor count or by floor count, the horizontal trends indicate that sparse and dense configurations execute in nearly identical time.

1. All sensor-count curves are almost horizontal: they show a slight descending trend in the interval between 0.1 and 0.5 but a completely flat behavior between 0.5 and 1.0. Overall variations range between 5% and 20% between 0.1 and 0.5 and between 5% and 8% across the remaining room distribution range.

2. An accentuated behavior is visible in the right chart: floor-count curves remain practically flat, confirming performance invariance regardless of building complexity.

This performance invariance has significant implications:

- Room distribution can be tuned to match building design requirements without performance penalties.

- Sparse configurations (many small rooms) offer better fault isolation while maintaining identical generation time.

- Dense configurations (fewer large rooms) simplify deployment without compromising solver performance.

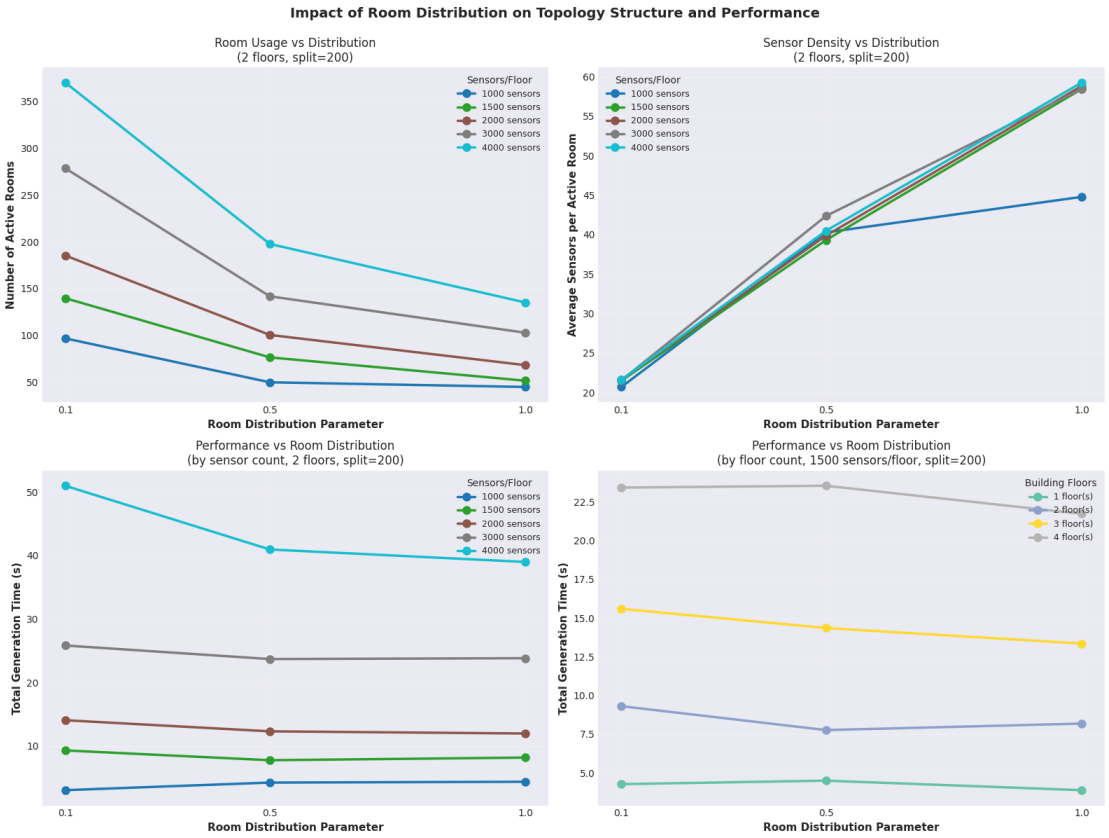

Figure 5.5: Room Distribution analysis

5.6 Summary of Findings

This experimental analysis revealed several key insights about the topology generation system performances.

- **Scalability Characteristics**: The system demonstrates approximately linear scaling behavior for smaller topologies structures, while showing a super-linear trend when increasing the number of floors. Furthermore the performance of the tool can be approximated to linear when fixing the number of floors and increasing the number of sensors.

  Room distribution has negligible impact on the execution time, allowing this parameter to be freely tuned to match building layout requirements. However, extremely small values may increase the possibility of not finding a solution (7 runs out of 540 failed). While it does not affect performance, room distribution significantly impacts the topology structure, with dense configuration reducing room count by up to 60% while maintaining identical generation times.

- **Split Size**: The configuration *split_size=200* resulted as the optimal default choice across the majority of the scenarios, achieving almost in every case the best performance. *split_size=100* resulted underperforming, especially in complex buildings with many floors, while *split_size=300* achieves decent performance only at very large scales (4000+ sensors).

- **Phase-specific behavior** Phase 1 (sensor allocation and sensor-to-controller assignment) dominates the execution time, accounting for 60-94% of total generation time, according to the specific configuration. This phase exhibits a super-linear trend increasing the number of floors. A similar trend can be visualized with the successive phase, although their absolute execution time are so small that the effect is practically negligible.

  For typical deployment scenarios (1,000-2,000 sensors per floor, 1-2 floors), the system generates topologies in under 15 seconds, which is acceptable for design-time usage. Larger deployments (4,000 sensors, 4 floors) require up to 140 seconds but remain practical for topology generation.

  The predictable scaling behavior within floor-count allows system designers to estimate generation time for new configurations based on the presented analysis. These findings establish the baseline performance expectations for the current implementation and provide clear direction for future optimization work, with Phase 1 sensor allocation identified as the critical element.

5.7 Correctness Assessment

The experimental evaluation presented in this chapter focuses primarily on the performance characteristics of the topology generation system. A natural question concerns whether the generated topologies are not only produced efficiently, but also correct with respect to the domain they are intended to model. This section discusses the notion of correctness in this specific context and the extent to which it can be assessed.

5.7.1 Constraint satisfaction correctness

The **constraint satisfaction correctness** consists of verifying whether every solution returned by the tool satisfies all hard constraints encoded in the CP-SAT model. This form of correctness is guaranteed by construction. When the CP-SAT solver returns a feasible solution, it provides a certificate that every constraint in the model is satisfied (e.g. each sensor is connected to exactly one controller, no controller exceeds its device capacity limit, inactive device instances have no connections, and sequential instantiation is respected. This guaranty does not

rely on testing or empirical observation, but follows directly from the semantics of constraint programming.

5.7.2 Limitations of Graph-Theoretic Validation

For general-purpose network topology generators such as GENIND [3], domain realism is validated by comparing graph-theoretic metrics between the generated topologies and publicly available real-world network datasets. This validation strategy is feasible because such reference datasets exist and because the structural properties under evaluation are not related to a specific domain.

For Building Automation System topologies, this approach encounters a fundamental obstacle. As documented in the introduction of this thesis and confirmed by the systematic reviews of Elnour et al. [1] and Li et al. [2], real-world BAS topologies are not publicly available. This unavailability of data represents the gap that motivated the development of the present tool. Attempting to validate the generated topologies against real BAS data would therefore require access to the very resources that the tool is designed to replace.

Chapter 6

Conclusion

This thesis addressed the challenge of generating realistic SB network topologies through automated constraint-based optimization. The work demonstrates that complex infrastructures can be modeled and synthesized using constraint satisfaction techniques while maintaining both realism and scalability.

The proposed framework introduces a hierarchical generation process capable of producing building automation network composed of heterogeneous devices and communication protocols. The solution is obtained by encoding each architectural constraint and device relationship in a constraint satisfaction model.

6.1 Key Insights

The main insight derived from this work is that a complex network generation becomes feasible when the problem is organized and reflects the hierarchy of the domain that is being modeled. The decomposition of the generation process into different phases is, indeed, a reflection of the layered structure of a SB environment, from the field layer, containing sensors and actuators, to higher layers components such as routers and servers. This feature showed to be fundamental for maintaining computational feasibility. Performance analysis showed that even with this approach, the scalability plots showed a super-linear scaling behavior by increasing the complexity of the floor. On one hand this behavior is expected since increasing the number of floor the total sensors involved progressively increases, but on the other hand it shows that if the topology generation would have been modeled as a monolithic constraint problem, without the hierarchical arrangement, the solver would likely encounter exponential growth in complexity.

A second insight emerges from the split-based strategy, introduced in the sensor-to-controller assignment phase. The combinatorial nature of the problem leads to a rapid growth in the number of possible configurations as the number of devices

increases. Partitioning the problem into coordinated subproblems significantly reduces the complexity of the search space keeping the validity of the result. As showed in the experimental results the current strategy provides a robust compromise between execution time and topological result.

Another important result concerns the performance invariance with respect to room distribution parameter, as observed in the experimental results. It has been showed, indeed, that different structural configurations produce distinct layouts but do not significantly affect the execution time. This behavior suggests that, independently from a sparse solution of small rooms and a dense version of large rooms, the solver adapts dynamically to the density within rooms, maintaining a stable behavior.

Finally, the performance analysis revealed that the majority of the execution time is concentrated in the first phase of the generation process, where sensors are allocated and connected to controllers. This phase represents a large portion of the total computation.

6.2 Limitations

Despite the positive results, this work also presents different limitations related to the used approach. The generated topologies focus on discrete relationships between devices to build the structure, such as connectivity assignments. However, aspects of real-world deployments are not currently addressed. In particular, the framework doesn't take in consideration spatial coordinates for devices, obstacle-aware device placement, or line-of-sight constraints for wireless communication. This limitation reveals an important weakness of constraint-based approaches: constraint satisfaction methods are effective at solving discrete allocation, however, it is less naturally suited for modelling continuous spatial relationship. This observation suggest that constraint-based formulations may be insufficient to fully define every aspect of a real-world building infrastructure. Instead, for future implementations the utilization of hybrid approaches, integrating both constraint satisfaction with spatial modeling offer a more comprehensive representation of BAS.

6.3 Implications for Future Work

From this work, different research directions emerge. First, the integration of spatial representation of devices would enhance the realism of the generated topologies. Adding geometric representations and obstacle-aware device placement would allow the resulting topology to better represent real deployments. However, such extensions should be carefully designed in such a way that adding a certain level

of realism doesn't affect in a substantial way the computational cost. A possible direction that can be taken is an hybrid approach involving constraint solvers for the allocation problems and specific algorithm addressing the spatial tasks.

Another area of development concerns additional optimization objectives. Although this thesis mainly focused on achieving the feasibility of automated topology generation, future work could develop further solutions such as minimization of connections length, balance device distribution across controllers, and maximizing fault tolerance within the network.

Further extensions could also include additional network components commonly present in modern SB infrastructures, such as gateways, firewalls, intrusion detection systems, or network segmentation mechanisms (VLANs). Including such elements would allow the framework to support more advanced use cases, particularly in the context of cybersecurity.

Finally, a final direction concerns the validation of the generated topologies against real-world BAS deployments. The current implementation guaranties constraint satisfaction correctness by construction, but the question of whether the model represents actual deployment practices remains open. Addressing this requires collaboration with BAS professionals and system integrators who can provide access to real building topologies, enabling a systematic comparison between the properties of generated and real-world networks.

6.4 Final Remarks

The proposed framework demonstrates that large-scale infrastructures can be synthesized efficiently while preserving key structural characteristics. The development of such a tool is valuable for research areas where access to real infrastructure data is limited due to privacy, security, or proprietary constraints. Synthetic topology generation enables experimentation for network security analysis and protocol evaluation. The framework developed in this thesis, based on hierarchical structuring, split-based optimization, and protocol-aware connectivity, provides a foundation for future research in SBs automated topology generation. The exploration of hybrid methodologies that combine constraint solving with additional spatial modeling techniques will be essential for building realistic and scalable representations of SB environments.

Bibliography

[1] Mariam Elnour, Nader Meskin, Khaled Khan, and Raj Jain. «Application of data-driven attack detection framework for secure operation in smart buildings». In: *Sustainable Cities and Society* 69 (2021), p. 102816. ISSN: 2210-6707. DOI: https://doi.org/10.1016/j.scs.2021.102816. URL: https://www.sciencedirect.com/science/article/pii/S2210670721001074 (cit. on pp. 2, 78).

[2] Guowen Li et al. «A critical review of cyber-physical security for building automation systems». In: *Annual Reviews in Control* 55 (2023), pp. 237–254. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2023.02.004. URL: https://www.sciencedirect.com/science/article/pii/S1367578823000032 (cit. on pp. 2, 78).

[3] Thuraya N.I. Alrumaih and Mohammed J.F. Alenazi. «GENIND: An industrial network topology generator». In: *Alexandria Engineering Journal* 79 (2023), pp. 56–71. ISSN: 1110-0168. DOI: https://doi.org/10.1016/j.aej.2023.07.062. URL: https://www.sciencedirect.com/science/article/pii/S111001682300649X (cit. on pp. 2, 19, 23, 78).

[4] Steffen Wendzel. «How to increase the security of smart buildings?» In: *Commun. ACM* 59.5 (Apr. 2016), pp. 47–49. ISSN: 0001-0782. DOI: 10.1145/2828636. URL: https://doi.org/10.1145/2828636 (cit. on pp. 5, 17).

[5] Pierre Ciholas, Aidan Lennie, Parvin Sadigova, and Jose Such. «The Security of Smart Buildings: a Systematic Literature Review». In: (Jan. 2019). DOI: 10.48550/arXiv.1901.05837 (cit. on pp. 5, 11, 16, 24).

[6] Dinesh Kumar Shamneesh Sharma and Keshav Kishore. «Wireless Sensor Networks- A Review on Topologies and Node Architecture». In: *International Journal of Computer Sciencesand Engineering* 1 (Oct. 2013), pp. 19–25 (cit. on pp. 7, 8).

[7] Sapna Singh Nivedita Bisht. «ANALYTICAL STUDY OF DIFFERENT NETWORK TOPOLOGIES». In: *International Research Journal of Engineering and Technology (IRJET)* 2 (Mar. 2015), pp. 88–90 (cit. on pp. 7, 8).

[8] Gjoko Krstic and Sipke Mellema. *I Own Your Building (Management System): Building Management Systems Security Research*. Security Research Whitepaper. Public whitepaper, 132 pages, accessed February 26, 2026. Applied Risk B.V., Nov. 2019. URL: https://www.zeroscience.mk/files/ioybms_gk_2019.pdf (cit. on pp. 9, 28).

[9] Christopher Morales-Gonzalez, Matthew Harper, Michael Cash, Lan Luo, Zhen Ling, Qun Z. Sun, and Xinwen Fu. «On building automation system security». In: *High-Confidence Computing* 4.3 (2024), p. 100236. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2024.100236. URL: https://www.sciencedirect.com/science/article/pii/S2667295224000394 (cit. on pp. 11, 14, 15, 17, 18, 24).

[10] Mary Catherine Heard. *BACnet Protocol Expands Dominant Market Share in Latest Market Research Report*. Published on AutomatedBuildings.com; accessed February 26, 2026. BACnet International. May 2024. URL: https://www.automatedbuildings.com/2024/05/bacnet-protocol-expands-dominant-market-share-in-latest-market-research-report/ (cit. on pp. 12, 14).

[11] Karan Lohia, Yash Jain, Chintan Patel, and Nishant Doshi. «Open Communication Protocols for Building Automation Systems». In: *Procedia Computer Science* 160 (2019). The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, pp. 723–727. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.11.020. URL: https://www.sciencedirect.com/science/article/pii/S187705091931720X (cit. on pp. 12, 14, 15, 24).

[12] Fábio Ferreira, A. Osório, João Calado, and C Pedro. «Building automation interoperability–A review». In: (2014) (cit. on pp. 13, 14, 24).

[13] Shobhit Jain, Vinoth Kumar N., A. Paventhan, V. Kumar Chinnaiyan, V. Arnachalam, and Pradish M. «Survey on smart grid technologies- smart metering, IoT and EMS». In: *2014 IEEE Students' Conference on Electrical, Electronics and Computer Science*. 2014, pp. 1–6. DOI: 10.1109/SCEECS.2014.6804465 (cit. on p. 14).

[14] Tae Jin Park, You Jin Chon, Dong Kyu Park, and Seung Ho Hong. «BACnet over ZigBee, A new approach to wireless datalink channel for BACnet». In: *2007 5th IEEE International Conference on Industrial Informatics*. Vol. 1. 2007, pp. 33–38. DOI: 10.1109/INDIN.2007.4384727 (cit. on pp. 15, 16).

[15] Chunwang Li Peizhe Tian * and Yimin Wu. «Wired/Wireless Hybrid Transmission of BACnet Messages». In: *The Open Electrical Electronic Engineering Journal* (Dec. 2016), pp. 189–196 (cit. on p. 16).

[16] Lukas Bechtel, Samuel Muller, Michael Menth, and Tobias Heer. «GeNESIS: Generator for Network Evaluation Scenarios of Industrial Systems». In: *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2024, pp. 1–4. DOI: 10.1109/ETFA61755.2024.10710752 (cit. on p. 19).

[17] Pradipta Ghosh, Jonathan Bunton, Dimitrios Pylorof, Marcos A. M. Vieira, Kevin Chan, Ramesh Govindan, Gaurav S. Sukhatme, Paulo Tabuada, and Gunjan Verma. «Synthesis of Large-Scale Instant IoT Networks». In: *IEEE Transactions on Mobile Computing* 22.3 (2023), pp. 1810–1824. DOI: 10.1109/TMC.2021.3099005 (cit. on pp. 20, 37).

[18] Wenzhuo Li, Rui Tang, Shengwei Wang, and Zhuang Zheng. «An optimal design method for communication topology of wireless sensor networks to implement fully distributed optimal control in IoT-enabled smart buildings». In: *Applied Energy* 349 (2023), p. 121539. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2023.121539. URL: https://www.sciencedirect.com/science/article/pii/S0306261923009030 (cit. on p. 20).

[19] Tiago Camilo, Jorge Sá Silva, André Rodrigues, and Fernando Boavida. «GENSEN: A Topology Generator for Real Wireless Sensor Networks Deployment». In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by Roman Obermaisser, Yunmook Nah, Peter Puschner, and Franz J. Rammig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 436–445. ISBN: 978-3-540-75664-4 (cit. on p. 21).

[20] A. Medina, A. Lakhina, I. Matta, and J. Byers. «BRITE: an approach to universal topology generation». In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2001, pp. 346–353. DOI: 10.1109/MASCOT.2001.948886 (cit. on p. 21).

[21] Anurag Verma, Surya Prakash, Vishal Srivastava, Anuj Kumar, and Subhas Chandra Mukhopadhyay. «Sensing, Controlling, and IoT Infrastructure in Smart Building: A Review». In: *IEEE Sensors Journal* 19.20 (2019), pp. 9036–9046. DOI: 10.1109/JSEN.2019.2922409 (cit. on p. 24).

[22] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998 (cit. on pp. 37–39).

[23] Dimitris Bertsimas and Robert Weismantel. *Optimization Over Integers*. Dynamic Ideas, 2005 (cit. on p. 37).

[24] Google. *OR-Tools: Google's Operations Research Tools*. `https://developers.google.com/optimization`. Accessed: 17 February 2025. 2024 (cit. on p. 39).

[25] João P. Marques-Silva and Karem A. Sakallah. «GRASP: A Search Algorithm for Propositional Satisfiability». In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521 (cit. on p. 39).

[26] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. IOS Press, 2009 (cit. on p. 39).

[27] Francesca Rossi, Peter Van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006 (cit. on pp. 39, 40).

[28] Laurent Perron, Paul Shaw, and Vincent Furnon. «Propagation Guided Large Neighborhood Search». In: *Principles and Practice of Constraint Programming*. 2004, pp. 468–481 (cit. on p. 39).

[29] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. «MiniZinc: Towards a Standard CP Modelling Language». In: *Principles and Practice of Constraint Programming*. 2007, pp. 529–543 (cit. on p. 40).

[30] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*. `https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`. 2013 (cit. on p. 51).

[31] Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, and Flemming Nielson. «The logic of XACML». In: *Science of Computer Programming* 83 (2014). Formal Aspects of Component Software (FACS 2011 selected extended papers), pp. 80–105. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2013.05.003`. URL: `https://www.sciencedirect.com/science/article/pii/S0167642313001238` (cit. on p. 51).

[32] D. Snoonian. «Smart buildings». In: *IEEE Spectrum* 40.8 (2003), pp. 18–23. DOI: `10.1109/MSPEC.2003.1222043`.