Technical University of Munich
TUM School of Engineering and Design

Automatic Detection and Interpretation of Changes in Massive Semantic 3D City Models

Huynh Duc An Son Nguyen

Complete reprint of the dissertation approved by the TUM School of Engineering and Design of the Technical University of Munich for the award of the

Doktor der Naturwissenschaften (Dr. rer. nat.).

Chair: Prof. Dr.-Ing. Liqiu Meng

Examiners:

1. Prof. Dr. rer. nat. Thomas H. Kolbe
2. Prof. Dr.-Ing. André Borrmann
3. Prof. Dr.-Ing. Youness Dehbi

The dissertation was submitted to the Technical University of Munich on 19.08.2024

and accepted by the TUM School of Engineering and Design on 19.11.2024.

SCHOOL OF ENGINEERING AND DESIGN — AEROSPACE AND GEODESY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Doctoral Thesis

Automatic Detection and Interpretation of Changes in Massive Semantic 3D City Models

Huynh Duc An Son Nguyen

SCHOOL OF ENGINEERING AND DESIGN — AEROSPACE AND GEODESY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Doctoral Thesis

Automatic Detection and Interpretation of Changes in Massive Semantic 3D City Models

Author:	Huynh Duc An Son Nguyen
Supervisor:	Prof. Dr. rer. nat. Thomas H. Kolbe
Co-Supervisors:	Prof. Dr.-Ing. André Borrmann
	Prof. Dr.-Ing. Youness Dehbi
Submission Date:	August 19, 2024

I confirm that this doctoral thesis is my own work and I have documented all sources and material used.

Munich, August 19, 2024 Huynh Duc An Son Nguyen

Acknowledgments

First and foremost, I would like to express my wholehearted gratitude to my Doktorvater, Prof. Dr. Thomas H. Kolbe, for his continuous guidance and support throughout the years. His excellence expertise in the field and his knack for simplifying complex concepts for the masses have inspired me to become both a better researcher and communicator. But most importantly, his passion and relentless quest for knowledge have been a great source of motivation for me to keep pushing the boundaries of what is known. His unwavering faith in me and my abilities has empowered me to pursue what I love doing the most: teaching and conducting research at the TUM.

I am grateful to my co-supervisor, Prof. Dr. André Borrmann, for his invaluable trust and support, especially during my time as a doctoral researcher at the Leonhard Obermeyer Center (LOC). I am deeply appreciative of the opportunities he provided, allowing me to engage in fruitful exchanges with the students from his group.

I also would like to express my gratitude to my co-supervisor, Prof. Dr. Youness Dehbi, for his continuous guidance and encouragement. His unwavering interest in my professional growth from the very beginning of my academic journey has been a great source of motivation for me.

I would like to thank Prof. Dr. Liqiu Meng for chairing my doctoral defence and for her invaluable support in organizing it. I greatly appreciate our discussions on enhancing teaching quality and learning experience at the TUM.

I am deeply grateful to Dr. Andreas Donaubauer for his advice and guidance during my time as a doctoral researcher at the TUM. His experience and vision in the field have been invaluable.

My heartfelt thanks go to my long-time advisor, colleague, and friend, Dr. Tatjana Kutzner. Her belief in my potential and her enduring support and guidance have been instrumental in bringing me to where I am today. Her tireless professionalism and work ethic have taught me the importance of not aiming for good, but striving for excellence and perfection.

I am grateful to my colleague and dear friend, Dr. Zhihang Yao, for his valuable advice and support over the years. His encouragement opened doors for me, paving the way for the research opportunities I am honoured to have today.

I would like to extend my appreciation to Dr. Claus Nagel for his vision, professionalism, and experience. His innovative tool *citygml4j* has been instrumental, serving as one of the foundation libraries used in the implementation of this thesis.

I am deeply appreciative of the support, encouragement, and numerous discussions with all my colleagues at the TUM. I cherish the unforgettable experiences we shared during our conference stays both in Germany and abroad.

I am also thankful to the developers and community of Neo4j for their open-source software solutions, which enable the use of a comprehensive graph database management system employed in my study.

I would like to thank the Leonhard Obermeyer Center (LOC), the company Virtual City Systems, and its parent organization, the CADFEM Group, for their partial financial support of my doctoral research.

My heartfelt thanks go to my closest friends, who have always been there for me since the day I arrived in Germany as a teenager.

I am grateful to my aunt, my uncle, and my cousins, who have been supporting me since my early years at the university.

Last but certainly not least, I would like to thank my parents for their unconditional love and unwavering support. Even over long distances, they have been accompanying me, sharing my greatest moments with me through all my academic milestones and achievements. Their weekly phone calls, filled with eagerness to hear about my work, have always been my source of motivation and joy. Their constant encouragement to strive for the best has been my guiding voice. I am also grateful to my little puppy sister, who has been bringing joy to our parents every day. She reminds me of my younger self, motivating me to keep moving forward. She is indeed worthy of many long cuddles and delicious treats!

Abstract

In recent years, Digital Twins have emerged as a major driving force behind many global technological and economic advancements. Despite various existing definitions, the general consensus agrees that a digital twin must involve a physical entity, a corresponding digital representation, and a continuous feedback loop between them. Focusing on the digital aspect of urban digital twins with semantic 3D city models (primarily in CityGML) as key components, there has been no significant progress in recent years on providing effective methods to keep track of changes between different temporal versions of a virtual city model. As a result, many existing smart city deployments often replace older datasets with newer ones. This approach not only wastes time and resources, but also discards all meaningful progresses that occurred during the given time period. Therefore, an optimal solution would require the automatic and efficient detection of changes, particularly in large-scale virtual semantic 3D city models. However, identifying changes is only half of the problem; the other half is to understand them. A comparison between two versions of a massive semantic 3D city model often results in millions of changes, rendering them incomprehensible to humans. Moreover, while each individual change may not contain any substantial information, when considered collectively, they can reveal valuable insights into the changes in the city models, and, to a certain extent, into the changes of the city. By uncovering hidden patterns and constellations among these changes, meaningful interpretations can be achieved. On the other hand, different stakeholders of an urban digital twin perceive changes vastly differently. While a data manager may be interested in changes to the geometric representations of a building, a city mayor is more interested in changes that indicate impacts of urban transformations on the city's living space and performance.

Therefore, this research proposes various methods for the automatic detection and interpretation of changes in semantic 3D city models. Due to their structural similarities, CityGML documents are first mapped onto graphs, which serve as a basis for all subsequent processes. Based on the generated graphs, a comparison is performed between the old and new city model. As graph matching is a challenging problem in graph theory, especially in large graphs, this study employs various optimization and heuristic strategies that leverage the rich semantic and geometric contents available in CityGML objects. Identified changes are also represented as nodes, attached to the source nodes where they were detected. To describe the complex patterns among these changes and define rules for identifying them, this thesis proposes the use of a rule network, a type graph that can capture all interrelations and dependencies of change patterns without redundancy. Then, the interpreter applies this rule network to the entire graphs, aggregating lower-level changes into new changes of higher semantic levels. This process reduces the number of changes to report significantly while increasing their semantic content at the same time. Finally, this research presents both the stakeholders and the interpreted changes in a semantic layered network. Path-tracing techniques within this network are introduced, enabling the identification of relevant changes for a specific stakeholder and interested stakeholders for a detected change.

The concepts and methods introduced in this thesis, along with their results, are evaluated using the city model of Hamburg as a case study. The evaluation employs two versions of Hamburg's 3D city model from 2016 and 2022, each covering an area of 750 km^2. Each dataset contains approximately 400 thousand buildings and occupies 8 GB of storage space. The graph representations of both datasets hold over half a billion nodes and 9 million detected changes. These graphs are stored in a graph database that requires 100 GB of storage space.

All proposed methods are implemented using the open-source community version of the graph database Neo4j. The software developed as part of this thesis is also open-source and can be accessed via GitHub. All datasets used for testing are publicly available as open data. Furthermore, the tools provided by this research can be applied to other datasets supplied by users.

Contents

1. Introduction

There is nothing permanent except change.
— Heraclitus

In 2007, a significant shift was observed in the global demographics: for the first time in history, the number of people living in urban areas surpassed those in rural areas, as recorded by the United Nations' World Urbanization Report (UN DESA, 2019). This trend of urbanization has only accelerated since then. By 2030, the urban population is projected to reach 68 %, with an expected increase of 2.5 billion people by 2050. Within this time frame, 10 more megacities, defined as cities with more than 10 million inhabitants, will emerge. As urbanization continues its rapid ascent, cities and nations worldwide are faced with the challenge of balancing between economic, social, and environmental needs through the integration of modern technologies.

As progresses are being constantly made to both the physical and digital infrastructures, cities undergo numerous transformations over time, with each iteration representing a different temporal stage. However, many smart city deployments worldwide currently lack the capacity to detect changes between these temporal stages, a deficiency that also hinders the comprehension of the dynamics underlying these changes. As a result, both the detection and interpretation of changes have emerged as one of the most pressing challenges in modern virtual city developments.

1.1. Changes in Semantic 3D City Modelling

Like in Building Information Modelling (BIM), semantic 3D city models are established methods for the modelling, representation, and analysis of 3D urban objects, such as buildings, bridges, and tunnels. While BIM is predominantly employed in the construction sector for the planning, design, and construction of new buildings, as well as the maintenance and renovation of existing structures (Volk et al., 2014), semantic 3D city models allow for the storage, exchange, and analysis of a variety of urban objects across a larger geographical area, which can extend from a handful of city blocks to all spatial objects within an entire country. Thus, semantic 3D city models are utilized in several key cases (Kolbe & Donaubauer, 2021):

1. **Urban Inventory of City Objects**: Semantic 3D city models serve as an inventory for a variety of urban objects. They store all relevant appearance, semantic, topological, and geometric information in one place, providing essential data on the stored objects. This is particularly useful in applications such as property and asset management, and the life-cycle management of city objects.
2. **Urban Data Integration**: Information from various domains like energy, population, and navigation is linked to city objects with certain spatial properties, such as to a building with fixed coordinates. Semantic 3D city models, predominantly represented according to the international standard City Geography Markup Language (CityGML), allow for the linking of this thematic data to city objects. This enables the enrichment of the city objects with additional thematic information.

Semantic 3D city models in the CityGML standard have been employed in more than 20 countries worldwide (Wysocki et al., 2022), including Australia, France, Germany, Japan, the United Kingdom, and the United States of America. Applications of semantic 3D city models can be broadly classified into visualization-based applications, such as flight simulation and light pollution assessment, or non-visualization-based applications, such as energy demand prediction and living space calculation (Biljecki et al., 2015). While semantic 3D city models can be used for visualization-based applications, they excel in non-visualization-based applications, where complex analyses can only be performed based on available semantic information provided by the 3D city models.

Therefore, when updates are made to a semantic 3D city model, such as through editing with software like the 3D City Database (3DCityDB), AutoCAD, or SketchUp, the resulting changes may reflect various deviations in the spatial and topological extent of city objects, their visual appearances, as well as the associated thematic data from various application domains. Typical examples include changes in the Coordinate Reference System (CRS) of a city model, spatial corrections of geometric objects, updated thematic attributes, and insertions or deletions of city objects or their parts. The ability to identify these diverse changes is crucial for capturing all progresses made to the city model, thereby laying the groundwork for further urban analyses.

1.2. Digital Changes in an Urban Digital Twin (UDT)

In recent years, digital twins have emerged as a major driving force behind many technological and economic progresses worldwide. The global market size for digital twins was estimated at over US$11 billion in 2023 and is projected to reach US$140 billion by 2030, with a compound annual growth rate of 42.6 % during the forecast period (Fortune Business Insights, 2023). Despite their rapid success, the concept of digital twins is not new. The birth of digital twins can be traced back to the early 2000s (Grieves & Vickers, 2016), but it was not until nearly a decade later that their application was first described in a major industry (Shafto et al., 2010). With the arrival of Industry 4.0, along with its digital advancements in Big Data and the Internet of Things (IoT), the concept of digital twins was 'rejuvenated' (Sharma et al., 2022), sparking further innovations, improvements, and discussions among a wide range of application territories seen today.

As digital twins are being employed in vastly different fields, many definitions of digital twins exist to date. In the context of smart cities and urban development, a digital twin of a city, or an Urban Digital Twin (UDT), is a comprehensive framework for organizing and harnessing various aspects of a city, ranging from physical components and logical structures to partaking actors and processes (Nguyen & Kolbe, 2024). Technical, legal, and administrative points of view co-exist in an urban digital twin. Urban digital twins are created for specific purposes. The goal is to gain essential insights into the state of the city and its development by observing and analysing the information available in its digital twin, thereby supporting both regular operations and critical urban planning and decision making. This thesis considers urban digital twins with semantic 3D city models serving as one of their key digital components.

Despite the many definitions of digital twins, the general consensus agrees that a digital twin must involve a physical entity, a corresponding digital representation, and a continuous feedback loop between the physical and digital entity. This means that changes in the real world must be reflected on the digital side, and vice versa. Such urban digital twins are illustrated in Figure 1.1 and described as follows:

1. **Physical World to Digital Entity**: An urban digital twin combines a vast amount of information gathered from a multitude of sources in the physical world, ranging from direct, closely-linked measurements (typically from in-situ IoT sensors) to remote sensing using various types of devices and platforms, as well as manually updated data. This information is then reflected in the virtual city model.

2. **Digital Entity to Physical World**: Urban digital twins, especially their virtual city models, have quickly become a crucial platform not only for storing, visualizing, and monitoring urban objects, but also for interpreting, simulating, and analysing urban environments in general. This process involves many types of transformations on the virtual city models, such as refinement, generalization, derivation, and enrichment, as shown at the bottom of Figure 1.1. These changes can be reflected back to the original physical counterpart for purposes such as automated manufacturing, manual construction or destruction, and control of components or systems. Moreover, they can be analysed to gain insights into changes that occurred in the real world, as illustrated at the top of Figure 1.1.

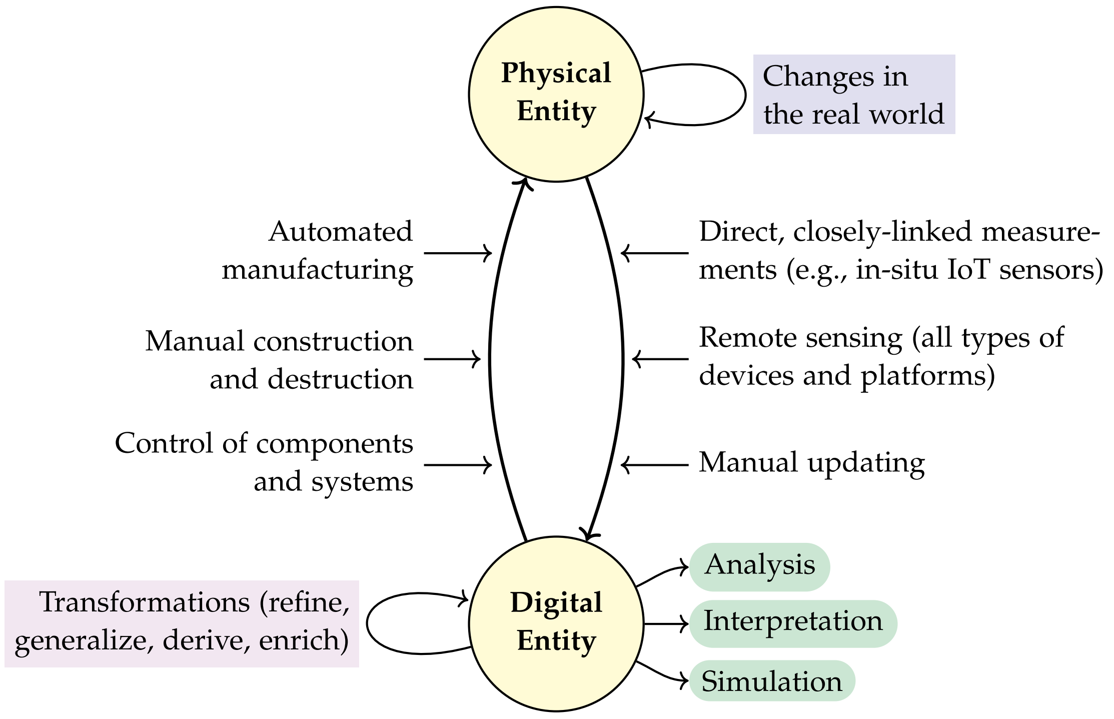

Physical Entity
Changes in the real world
Automated manufacturing
Manual construction and destruction
Control of components and systems
Direct, closely-linked measurements (e.g., in-situ IoT sensors)
Remote sensing (all types of devices and platforms)
Manual updating
Digital Entity
Transformations (refine, generalize, derive, enrich)
Analysis
Interpretation
Simulation

Figure 1.1.: An overview of an Urban Digital Twin (UDT). Various processes can be employed to reflect the changes in both the physical (top) and digital entity (bottom). Digital changes may correspond to actual changes in the real world or result from digital processes such as refinements, simulations, and scenario testing. By detecting and interpreting these changes, additional insights can be revealed, allowing for more in-depth urban analyses.

As a result, an urban digital twin acts as a comprehensive repository of both physical and digital entities. It preserves a cumulative collection of different versions of these entities over time, allowing for the predictive maintenance of the city in the real world and in-depth analyses through the examination of the feedback loop between the physical and digital entity. However, despite being one of the defining features of an urban digital twin, such systematic two-way synchronization poses significant challenges for implementation and scalability (Grieves & Vickers, 2016; Sharma et al., 2022), especially when the procedures that initially performed these changes were neither documented nor maintained.

A recent survey (Lei et al., 2023) conducted among international experts identified updating, including change detection and version management, as one of the most frequently cited technical challenges of urban digital twins, for which no complete solution has yet been achieved.

As a result, many current smart city deployments, especially those involving semantic 3D city modelling, often replace old datasets with newer ones. This approach not only wastes time and resources, but also discards any meaningful progresses that could have been captured between the datasets during the given time period. Therefore, an optimal and sustainable strategy would require the automatic, efficient, and accurate detection of changes, particularly in large-scale virtual semantic 3D city models.

1.3. Interpretation of Changes in Semantic 3D City Models

While the identification of changes in semantic 3D city models is essential, it is only one aspect of the problem. Another crucial task is to understand the significance and scope of these changes, as well as their impact on others.

1.3.1. Interrelations and Patterns among Changes

As illustrated in Figure 1.2, the relationships between digital and physical changes in an urban digital twin are presented along a timeline. The changes detected in the digital 3D city model may correspond to the real-world changes that occurred in the past, a result of the synchronization process from the physical to the digital entity. For example, a newly inserted building in the semantic 3D city model may indicate a recently constructed building in the physical world.

On the other hand, digital changes, like those from simulations, can be utilized to predict future changes in the real world. For example, prior to its construction, a planned new parking garage must first undergo rigorous simulation tests. Once approved, changes made in the simulation can be realized in the physical world.

However, these digital changes may not always correspond to real-world modifications. They could simply represent changes solely within the digital city model, such as those made during hypothetical scenario testing independent of the actual world.

Furthermore, changes, whether in the physical or digital world, are not merely individual discrepancies in the data. They are often interconnected, as they can be caused by a single common action in the real world, or one change can trigger a chain reaction of other changes. For instance, a single upward shift of an entire building may lead to a corresponding upward shift of all its boundary surfaces. This translated building may, in turn, coincide with the translation of all other buildings, indicating a systematic shift among all buildings in the city model. This single observation is often much more valuable to stakeholders, such as the manager of an urban digital twin, than a multitude of individual uninterpreted changes. Such patterns among the changes are hidden behind the data but can provide significant insights when uncovered, especially for a large number of deviations detected across the datasets.

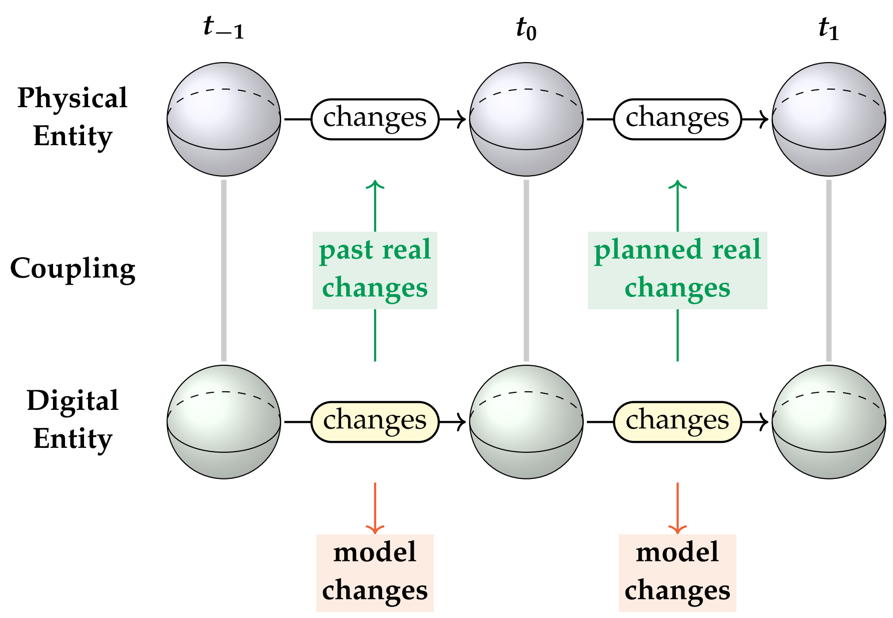


Figure 1.2.: An illustration of an Urban Digital Twin (UDT) over time. Identifying and understanding changes on the digital side can reveal insights into the past changes that occurred in the physical world. When utilized for simulation or scenario testing, these digital changes may also predict future changes in the physical entity. Additionally, changes on the digital side may refer to model changes only, independent of the physical reality. The primary objective of this research is to detect and interpret all these changes.

Therefore, in addition to identifying digital deviations between the temporal versions of a semantic 3D city model, further interpretations are required to reveal the semantic interrelations and patterns among these changes.

1.3.2. Human-centric Interpretation of Changes

The identified patterns within the aforementioned changes provide crucial insights into the interrelations and composition of changes in the semantic 3D city models. These interpretations can effectively condense a large number of detected changes into a substantially smaller, yet semantically richer set of changes.

However, as these interpretations must ultimately serve humans, and different groups of stakeholders perceive different types of changes vastly differently, these interpretations must be further categorized and refined to cater to the specific interests of each stakeholder.

For instance, among the detected changes and their interpretations, the city mayor, who is responsible for the decision making and policy development of a city, is predominantly interested, for example, in changes that may have an impact on the city's total living space. This includes instances such as the recent construction of a new residential building, the repurposing of a commercial building for residential use, or the renovation of building to add more storeys. On the other hand, an expert in the field of Geographic Information System (GIS), or a GIS specialist, who often maintains and updates virtual models of cities, is primarily interested in changes related to the overall quality of the data, the representation of geometric objects, and the visualization of these changes.

To enable this, there needs to be a method in place that can explicitly describe the relevance relations between different changes and stakeholders. Based on this description, the approach should then be able to efficiently determine whether a given change is relevant to a specific stakeholder, and vice versa, which changes a given stakeholder is interested in.

However, considering that stakeholders' interests in changes also vary over time, the proposed approach should be adaptable and flexible enough to accommodate these variations.

1.4. Problem Statement and Scope

This research addresses the fundamental problem of detecting and interpreting changes in semantic 3D city models, as well as in urban digital twins, where semantic 3D city models play a key role. The problem is described as follows:

Research Problem Statement

Given two temporal versions of a semantic 3D city model and a group of various stakeholders, the objectives are to:

1. Detect all changes that occurred between these documents, covering semantic, geometric, and topological contents, and

2. Deliver insightful interpretations of these changes, taking into account the different perspectives of stakeholders.

Given the expansive scope of urban digital twins as a research domain, which includes changes on both the physical and digital side, this thesis specifically focuses on the handling of changes within the digital components of an urban digital twin. The digital entity of an urban digital twin may involve not only semantic 3D city models, primarily encoded in CityGML, but also models from the built environment, such as BIM, given in Industry Foundation Classes (IFC). Given the complexity of this issue, this thesis only considers changes within semantic 3D city models that represent the digital components of an urban digital twin.

Furthermore, the CityGML data model and encoding standard version 2.0 is employed. However, as explained throughout this thesis, the proposed methods leverage the semantic and geometric information available in semantic 3D city models, which are also present in BIM and the newer CityGML version 3.0. Therefore, the methods proposed in this thesis can be adapted and extended to accommodate BIM and corresponding IFC models, as well as CityGML 3.0.

This thesis introduces methods capable of detecting all potential changes between two temporal versions of a CityGML document. These include modifications to the semantic, geometric, and topological contents. Changes in the visual appearance of city objects, which often involve updates in the appearance images, require corresponding methods from the field of computer graphics and computer vision, and are thus beyond the scope of this thesis.

When interpreting changes, this thesis does not aim to provide a comprehensive static model of all stakeholders. Instead, it focuses on providing an adaptable and extendable approach that allows for modelling the dynamic relevance relations between changes and stakeholders.

The results of this research can be utilized to enable complex semantic analyses on the city model and enhance the update of older datasets, ultimately contributing to the realization of the continuous bidirectional data flow necessary between the physical and digital entity of an urban digital twin.

2. Challenges of Comprehending Changes in Semantic 3D City Models

As highlighted in Chapter 1, the primary goal of this research is to provide comprehensive yet comprehensible interpretations of changes between two different temporal versions of a semantic 3D city model, particularly those represented according to the standard City Geography Markup Language (CityGML). However, this task is highly complex and is divided into several smaller steps, each presenting a set of its own challenges that need to be addressed. These challenges are summarized in this chapter, with more detailed discussions to be provided for each step in the following chapters of this thesis.

2.1. The Complexity of the CityGML Data Model

The City Geography Markup Language (CityGML) is an international standardized information model and exchange format, commissioned by the Open Geospatial Consortium (OGC), designed to model, store, and exchange most 3D urban objects (Gröger et al., 2012). In contrast to other 3D city models that are purely graphical, such as those with aerial mesh geometries, CityGML allows for the linking of graphical appearances and geometries with various semantic properties of city objects. The result is a comprehensive, general-purpose semantic 3D city model that has found extensive applications across a wide variety of domains (Kolbe et al., 2008; Schwab et al., 2020; Willenborg et al., 2017). However, due to its inherent characteristics and design, CityGML introduces several challenges that add complexity to the process of detecting and interpreting changes in CityGML documents.

2.1.1. The Complex Data Model of CityGML

The CityGML data model version 2.0 represents city objects using a large number of classes. These classes are organized into fourteen different thematic modules, such as *Building*, *Bridge*, and *Tunnel*, and are structured in a sophisticated class hierarchy with complex aggregation and inheritance relationships.

Typically, a non-abstract class exists as a subclass within a deep hierarchy with multiple levels of abstract superclasses above it. For instance, the class *Building*, representing all buildings in the city model, is a subclass of the abstract class *Abstract-Building*, which, in turn, is a subclass of *AbstractSite*. This abstract class is derived from *AbstractCityObject*, which is an extension of *AbstractFeature*. Finally, at the top of this class hierarchy lies the abstract superclass *AbstractGML*, representing all Geography Markup Language (GML) objects. Thus, there are in total six classes in this hierarchy that are employed by CityGML to model all buildings.

Therefore, when comparing two building instances, it is crucial to consider all information inherited from all superclasses within this hierarchy. The definitions of all such class hierarchies can be found in the Unified Modelling Language (UML) class diagrams provided by the official CityGML specifications (Gröger et al., 2012), as well as the model's XML Schema Definition (XSD) files.

Moreover, the CityGML data model allows for the modelling of city objects in five different Level of Details (LODs), spanning from LOD0 to LOD4, each with an increasing level of detail. For instance, a building's LOD0 representation displays its footprint, while its LOD1 representation extends this footprint vertically, forming a prismatic block. In LOD2, distinct boundary surfaces, such as roof, wall, and ground surfaces, are introduced. The LOD3 further incorporates openings such as doors and windows. Lastly, the most detailed LOD4 allows for the inclusion of indoor features, such as rooms and furniture (Kolbe, 2009).

The selection of a suitable LOD depends on the quality available and complexity required in each application. Notably, many geometric elements in CityGML, such as *lod[1-4]Solid* and *lod[1-4]MultiSurface*, exist across multiple LODs with varying degrees of detail. As a result, this LODs information should be considered when interpreting changes in LODs. For example, during the comparison of two CityGML documents, the consistent addition of a higher LOD to all buildings within the city model indicates an improvement in the quality of the building representations within the datasets.

2.1.2. The Graph Characteristics of CityGML

CityGML 2.0 is an application profile of the Geography Markup Language (GML), which, in turn, is a grammar of the Extensible Markup Language (XML). As such, CityGML inherits numerous methods and features from both XML and GML. This includes its independence from the order of elements and the use of the XML Linking Language (XLink).

Although CityGML documents are often provided in textual format, they do not distinguish between the order of sibling elements under the same parents. As a result, for two city models with identical content, a building defined at the beginning of the first may potentially correspond to a building defined at the end of the second file describing the same city model. The same also applies to sub-elements within each building, and so forth.

In XML, XML Linking Language (XLink) is a mechanism used to link a current XML element with another pre-existing one, thereby establishing a connection between the two elements without the need for redefinition (W3C, 2006). This concept is widely utilized in GML for the definitions of its geometries, and consequently, it is also extensively employed in CityGML for the definitions of various city objects.

For example, a building represented in LOD2 consists of several boundary roof, wall, and ground surfaces. Each of these surfaces is defined individually as a polygon. In addition to these, the building also includes another element: a solid that represents the 3D shape of the building, which is confined by these boundary surfaces. As a result, this solid is defined as a collection of XLink references, with each XLink pointing to an existing boundary surface.

Another application of XLinks in CityGML can be found in implicit geometries, where a prototypical shape is defined once and can be reused or referenced multiple times. This is particularly useful for representing city objects with similar shapes, such as trees and traffic lights.

CityGML also utilizes XLinks for city object groups composed of other city objects, such as buildings within a district or trees within a park. Each of these buildings or trees, often defined already elsewhere in the dataset, can be referred to by the city object group using XLinks. In addition, CityGML allows for a city object group to be nested within other larger city object groups (Gröger et al., 2012), leading to the chain use of XLinks.

The advantage of using XLinks is two-fold. Firstly, it allows for reusing elements, thereby minimizing redundancy. Secondly, it enables the explicit modelling of topological relationships between 3D geometric objects in GML-based documents. In the aforementioned example, the use of XLinks in defining the building's solid geometry indicates that all referenced boundary surfaces represent a side of the solid. Thus, they collectively form a closed 3D volume, with each side adjoining at their boundaries.

While XLink is commonly used to link existing objects in CityGML, it also serves a practical purpose, as explained throughout this thesis: reconnecting segmented parts of large CityGML documents. This is typically the case when an input CityGML document is so large that it needs to be divided into smaller segments for efficient reading and parsing. For instance, a CityGML document containing one million buildings is too large to be loaded entirely into main memory. Thus, it can be first divided into one million smaller pieces, each containing a single building. This division allows for the processing of arbitrarily large CityGML documents in manageable chunks, leading to more efficient memory consumption and the possibility for multi-threaded processing.

During this division, an XLink is inserted between the city model element and each of its building chunks. After processing, serving as anchors, these XLinks are resolved, effectively reconnecting all buildings back to the main city model.

While the use of XLinks offers numerous benefits, it also introduces an additional layer of semantic and structural complexity to GML and CityGML documents. This complexity arises from the fact that a single element may be referenced by multiple others, thereby transforming their structure from tree-like form to a network-like one. Figure 2.1 illustrates this graph structure of a CityGML document, which utilizes XLinks to define its solid from existing boundary surfaces, regardless of the order of sibling elements. As a result, despite their textual representation, GML and CityGML documents essentially exhibit graph-based structures. Consequently, CityGML documents cannot be accurately matched using solely traditional comparison tools that are designed for plain texts or tree structures.

2.1.3. Large Sizes of CityGML Documents

As mentioned in Chapter 1, semantic 3D city models are comprehensive repositories of various types of information, representing a wide variety of 3D city objects. Their scope can vary from a handful of building blocks to the entirety of a region's buildings. Therefore, CityGML documents can become very large in size, typically allocating gigabytes of storage for datasets that contain all buildings of an entire state or country.

The sheer number of city objects often results in a significant amount of main memory needed for loading and processing input CityGML documents. This also results in more time and computational resources required when matching objects that are in one-to-many or many-to-many relationships. For instance, when matching one million buildings in the older city model with another one million buildings from the newer city model, brute-force matching could result in one trillion comparisons of all possible pairs, rendering it unfit for large datasets.

Therefore, it is necessary to implement effective optimization strategies to minimize runtime complexity and memory consumption for large CityGML documents.

2.2. The Complexity of Matching CityGML Documents

In order to provide a comprehensive understanding of changes, it is essential to first detect the changes between two given CityGML documents. However, a number of challenges must be overcome, as described in the following sections.

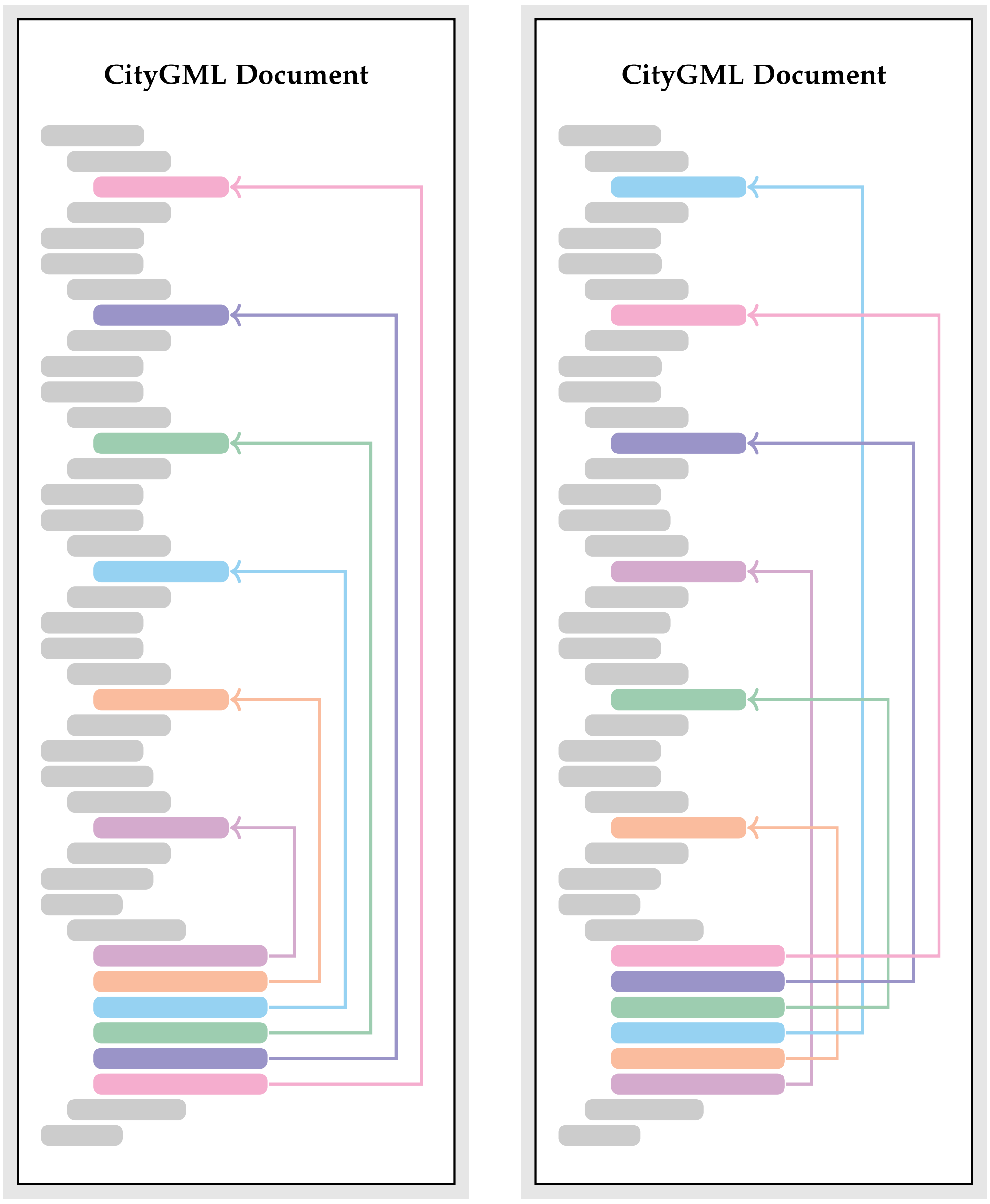


Figure 2.1.: An illustration of the graph structure of CityGML documents. In this example, a building's solid (bottom) is defined using XLinks that reference existing boundary surfaces (top). This may lead to cycles among elements. Moreover, the order of sibling elements under the same parent does not play any role in CityGML, as both left and right figure represent two of the many possible arrangements of the same document. Thus, CityGML documents exhibit a graph structure.

2.2.1. The Graph and Subgraph Isomorphism Problem

Due to the graph-based nature of CityGML, as explained previously in Section 2.1.2, the comparison of the content and structure of two CityGML objects is equivalent to matching the content and structure of their respective graph representations, provided a method for lossless mapping of CityGML objects onto graphs is used.

This corresponds to the graph and, more specifically, the subgraph isomorphism problem in graph theory. These problems involve determining whether two graphs are structurally identical, or whether the second graph is contained within the first. While the graph isomorphism problem is generally not known to be solvable in polynomial time nor to be NP-complete (McKay & Piperno, 2014; Skiena, 2008), the subgraph isomorphism problem is even more complex and known to be NP-complete (Ullmann, 1976). Thus, both the graph and subgraph isomorphism problem are difficult and require further optimization and heuristic strategies for large graph representations of CityGML documents.

In addition, the process of comparing two graph representations of CityGML documents extends beyond a simple decision problem. Merely providing answers such as 'true' or 'false', or 'identical' or 'not identical', is not sufficient. The process must also identify changes and accurately report their locations in the graphs when detected.

Moreover, graph isomorphism is only applicable when the two graphs have an equal number of vertices. In addition, both the graph and subgraph isomorphism evaluate whether two graphs are structurally identical. However, as GML and CityGML allow multiple syntactic ways to define an object, as will be explained in Section 2.2.3, multiple graph representations of the same CityGML object may exist. While these graphs represent the same content, their different structures result in them being consistently classified as 'not matched' by the standard graph isomorphism.

As a result, the matching process must employ a modified version of graph and subgraph isomorphism. This adaptation should leverage the rich semantic contents, such as the semantic labelling of nodes and relationships available in the graph representations of CityGML documents to enhance the runtime efficiency.

2.2.2. Identifier-independent Matching

In practice, CityGML objects are associated with a unique identifier that remains consistent throughout the object's lifespan and across various datasets. For instance, in Germany, many CityGML documents, especially those created by the state mapping agencies, reference objects based on their identifiers extracted from Amtliches Liegenschaftskatasterinformationssystem (ALKIS), the official real estate cadastre information system for Germany (AdV, 2008). In addition to identifiers, ALKIS also provides information about the shape, size, location, usage, and ownership of all registered real estate properties, including land parcels and buildings. The identifiers defined in ALKIS generally remain unchanged during the objects' entire lifespans and can be shared across multiple datasets that use the same land parcels or buildings. As a result, many urban analyses and processes leverage these unique identifiers to quickly retrieve associated objects.

However, this uniqueness and consistency of identifiers are not always guaranteed, as identifiers can be altered by a data manager or through a software change. Thus, the matching process should not rely solely on the identifiers of objects when searching for the best match of a reference city object.

2.2.3. Syntactic Ambiguities allowed by GML and CityGML

CityGML is a specialized application of GML3, which is based on the International Organization for Standardization (ISO) 19107 model (Herring, 2020) for defining feature geometries. Despite CityGML's utilization of only a subset of GML geometries, it adheres to the syntactic rules of GML and the geometric definitions of ISO 19107.

The GML standard allows for the definition of a wide variety of geometries through different syntactic methods. For example, a polygon in GML is a two-dimensional surface bounded by an exterior ring (representing the outer boundary) and any number of interior rings (representing holes inside of the surface). A ring is a closed sequence of control points that define a series of adjacent line segments. When the interior rings touch each other at their boundaries, they can be merged together to form a larger interior ring. Despite the differing number of interior rings, these two polygons are geometrically identical.

Similarly, a three-dimensional solid may also be defined in GML using different syntactic methods. It can be given by explicitly defining its boundary surfaces or by forming a collection of XLinks that reference and reuse the pre-existing boundary surfaces of a building.

There exist many other instances where the same objects can be defined using various syntactic rules. These include the cases where a point in 3D space can be provided either as a list of coordinates or a collection of three distinct sub-elements, each representing one of its coordinates. In another example, measurements with different values are still considered equal, if their units and values align, such as 1 m and 1,000 mm.

These syntactic ambiguities provide users, such as GIS specialists, with the flexibility and freedom to define geometries in their preferred syntactic styles. However, these ambiguities also add a significant layer of complexity to the comparison of CityGML objects, especially geometric objects, as all the different syntactic methods used to define an object must be taken into account when matching objects.

2.2.4. Uncertainties in Detecting Geometric Changes

In addition to the syntactic ambiguities discussed previously, the matching process must also account for geometric uncertainties. These include small acceptable error tolerances when comparing lengths, angles, areas, and volumes. For instance, two points are considered identical if their distance is smaller than the acceptable length error tolerance. Two sets of line segments are considered equal if all their vertices are located within a certain proximity of each other. Similarly, two polygons are regarded as equal if the angle between their normal vectors is less than an angle error tolerance, and they share at least 90 % of their area. Lastly, two solids are considered equal, if they share at least 90 % of their volume. These percentage values can be adjusted.

However, the guidelines for matching geometric objects explained above do not account for the possibility of geometric transformations, such as shifts by a translation vector, rotations by a rotation vector or matrix, changes in vertical and horizontal size, or any combination thereof. Figure 2.2 gives an example of such geometric transformations for 2D surfaces. Therefore, the matching process must be capable of matching geometries based on their spatial properties and given error tolerances, while also accounting for potential occurrence of combinations of geometric transformations.

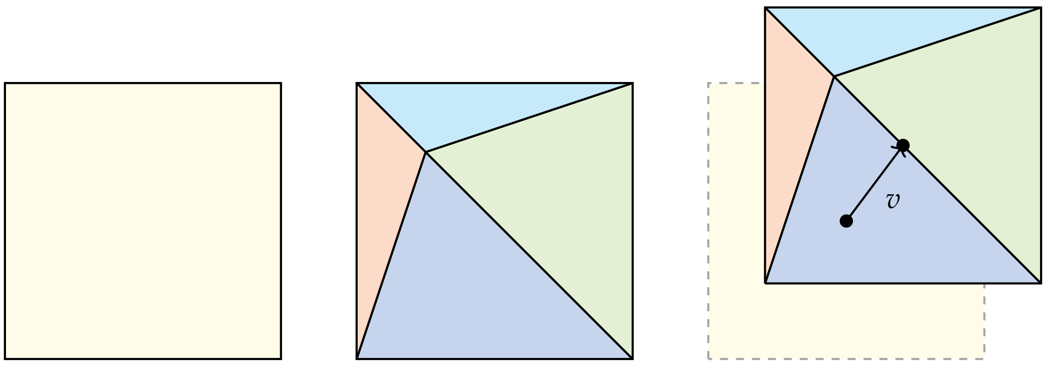


Figure 2.2.: An example of geometric changes in semantic 3D city models. On the left, a 2D surface is shown. In the middle, the same surface has been triangulated, while retaining its original size, location, and orientation. On the right, the triangulated shape has been translated by a vector v. Despite these changes, all surfaces are considered equivalent. The first and second surface are equivalent due to their identical geometric extents, differing only in their syntactic representations. The second and third surface are equivalent for sufficiently small values of $\|v\|$, with error tolerances taken into account. This comparison becomes even more complex when also considering size changes, the 3D orientations of 2D surfaces, and 3D geometries.

2.3. The Challenging Task of Interpreting Detected Changes

In previous steps, all base changes are detected between different temporal versions of a CityGML document. These changes are then subject to further analysis, providing a deeper understanding and comprehensive evaluation of the changes from the perspective of diverse stakeholders. As with preceding phases, this step introduces its own set of challenges that need to be addressed.

2.3.1. Large Sets of Low-level Detected Changes

During the matching process, changes between the content and structure of two CityGML documents are identified. These changes are presented as edit operations or base changes, directly attached to the source elements where the changes occurred. This allows for efficient retrieval of relevant information during the interpretation process.

At the lowest level, these changes include basic modifications such as the insertion, removal, and updating of properties, as well as the insertion and removal of objects. Even for CityGML documents of smaller sizes, the change detection process may produce a significant number of such edit operations. In large CityGML datasets, this number could reach millions of changes. Although these changes and their connected objects could provide valuable insights for subsequent analyses, their sheer number at this initial stage renders them difficult for humans to comprehend, especially those without specialized knowledge of the underlying data structures and software tools being used. As a result, these changes must be further analysed, interpreted, and semantically enhanced before they can be presented to stakeholders.

2.3.2. Hidden and Intercorrelated Patterns among Changes

A change is not simply a deviation in the data; it has a meaning, context, and impact on other elements. In fact, the majority of changes detected in CityGML documents are often part of at least one larger constellation.

For instance, on its own, a single change in the location of a building's boundary surface may not provide any substantial meaning. However, if all boundary surfaces of all buildings in the city model have been moved by the same offset, a pattern emerges that suggests a systematic translation of the entire city model. In large city models, this one single pattern can represent millions of otherwise incomprehensible geometric changes.

In this thesis, patterns among changes are viewed as semantic aggregations, combining changes of lower semantic levels into changes of higher semantic levels. As the semantic level increases, the number of (interpreted) changes reduces. A pattern can therefore be thought of as a condensed semantic representation of all its member changes.

However, such patterns are often hidden behind the data and pose a challenge for interpreting due to the following reasons:

1. **Large Quantity**: A vast number of object types exist in the underlying CityGML data model, leading to a correspondingly large number of potential patterns among them.
2. **Interdependent Patterns**: An outcome of a pattern may be required and utilized by another. For instance, the pattern of systematic translation of all buildings in the city model is only valid if all buildings have been translated by the same offset. A building, in turn, is considered translated, if all of its boundary surfaces have been shifted by the same amount.

Therefore, to capture such hidden and complex patterns among changes, the interpretation process must be guided by well-defined rules. These rules dictate which lower-level changes must be aggregated into higher-level changes. Many current smart city deployments employ a database query on an ad-hoc basis for each pattern. This approach not only requires expert knowledge of the database in use, its structure, and its query language, but it also leads to repeated interpretation of many patterns if these patterns are interdependent on each other (forming a 'pattern' of patterns).

To reduce redundancy due to the interdependence among change patterns, the interpretation process should allow for the centralized definition of pattern rules, with each type of change employed at most once. Corresponding criteria and multiplicities, required from the lower-level changes to form the next higher-level changes, should also be evaluated.

2.3.3. Multiple Perspectives when Interpreting Changes

In contrast to the preceding change detection, the interpretation process aims to provide insights into changes that are relevant to specific stakeholders, a process that is inherently subjective in nature.

For instance, a city model manager is involved in the creation, maintenance, and analysis of semantic 3D city models. They are interested in changes in the data quality of the city model, such as improved numeric precision and increased LODs. Additionally, they are also interested in the changes of the city model itself, particularly when there is a systematic vertical shift of all buildings, which may indicate a change in the elevation or terrain used in the city model.

On the other hand, a city mayor, who is involved in the decision and policy making of the city, is interested in changes that have an impact, for example, on the city's total living space. These changes include the construction of new buildings or demolition of existing buildings, updates in the number of storeys in buildings, and variations in the size of ground surfaces.

Therefore, besides the capability to detect hidden patterns among the detected changes, the interpretation process must also be able to provide to specific groups of stakeholders only the information they find most relevant, and vice versa. Figure 2.3 illustrates an example of such interpretations in both directions between the stakeholders and changes. This requires an adaptable and extendable strategy that can model and capture the complex interrelations between changes and stakeholders in one central place, such as in a layered network that comprises all types of changes and groups of stakeholders.

2.3.4. Stakeholders' Varying Interests in Changes

Another challenge in interpreting changes with respect to stakeholders is that a stakeholder may have vastly different interests in different types of changes. Furthermore, a stakeholder's interest in a specific type of changes may fluctuate or even vanish over time. On the other hand, a single type of changes may be relevant to multiple groups of stakeholders at the same time, each with different levels of relevance.

As a result, the model that represents the relevance relations between changes and stakeholders must be flexible enough to accommodate adjustments in the relevance values, or even the deletion and addition of relevance relations between changes and stakeholders.

2.4. Use Case Example: Introduction of Stakeholders

CityGML serves as a versatile information model, and any changes detected in CityGML documents provide a wide spectrum of information for a variety of stakeholders. However, not all changes are of equal importance to stakeholders. It is therefore crucial to distinguish between three cases: (1) stakeholders are interested only in changes in the real world, (2) stakeholders are exclusively interested in changes in the city models, and (3) stakeholders are interested in changes in the city models to conclude changes in the real world.

The application of the methods proposed throughout this study, especially the interpretation process, is illustrated based on the following three example actor roles: a data broker in the private sector, a city model manager working at a mapping agency, and a city mayor representing the city planning department. They are described as follows:

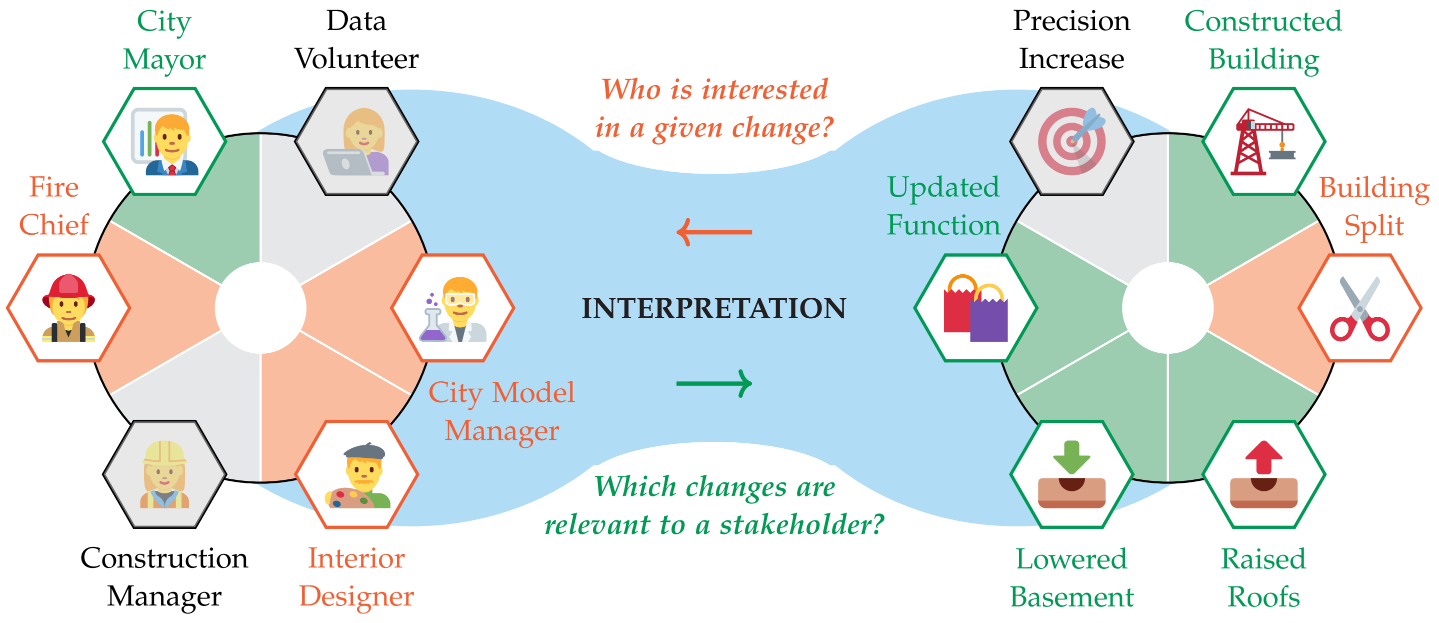


Figure 2.3.: An illustration of the dynamic two-way process of interpreting changes from the perspectives of various stakeholders. From left to right (green): For a given stakeholder, the process determines relevant changes. From right to left (orange): For a given change, the process determines interested stakeholders. For instance, the change indicating the division of an old building into two smaller adjoining ones, as depicted in orange on the right, may be of interest to city model managers, interior designers, and firefighters, as shown in orange on the left. Conversely, the city mayor, displayed in green on the left, may be interested in changes that have an impact on the city's available living space, such as newly constructed buildings, buildings repurposed from commercial to residential use, or changes like lowered basements and raised roofs. This dual approach ensures a comprehensive understanding of the complex interrelations between changes and stakeholders. The icons depicting the stakeholders and changes shown in this figure were provided by Twitter, Inc. (2019) and other contributors under the Creative Commons Attribution 4.0 International (CC BY 4.0) licence.

1. **Data Broker**: Typically part of the private sector or larger corporations, a data broker collects data from a variety of sources, organizes this information, and then sells highly specialized data tailored to specific buyers. Such information can be employed for various applications, such as for market analysis, research studies, and decision making. In the context of urban digital twins and semantic 3D city modelling, a data broker is concerned with all aspects of the available data, including semantic and geometric content, and thus requires as much detail as possible. Therefore, in the context of change detection and interpretation in CityGML documents, a data broker is interested in a wide variety of changes, ranging from updated thematic properties to modified geometries. Therefore, a data broker is predominantly interested in changes within the city model.

2. **City Model Manager**: A city model manager is responsible for the maintenance, enhancement, and quality control of the 3D city models or urban digital twins of a city. A city model manager may belong to the private or public sector, such as in a mapping company or mapping agency. They are primarily interested in the changes made to the digital city models. Their interests include an improvement in the data accuracy, model refinement and completion (such as increased LODs), and spatial and geometric changes. As for changes in the real world, a city model manager is only interested in verifying whether these changes have been accurately reflected in the city models.

3. **City Mayor**: As a key decision-maker, a city mayor holds significant influence over the city's overall development. Operating within the public sector, specifically in the city administrative and planning department, a mayor's focus lies in the actual condition and well-being of the city, rather than the status of its models. Thus, a city mayor is only interested in changes in the city models that reflect actual changes within the real city, such as large-scale or significant changes that can directly affect the living conditions and experiences of city residents. These changes include those that affect the living space of citizens, such as the enlargement of existing buildings, the construction of new building, the addition of storeys, changes in building energy demand to account for resource allocation and utility infrastructure planning, the repurposing of buildings (such as transitioning from residential to commercial use), or the preservation of historical buildings that, unlike others, prohibits any structural changes to the buildings.

Chapter 7 demonstrates how all methods proposed in this research can be applied to provide valuable insights into the changes of a city using its real-world datasets over the years. These interpretation results are further evaluated and presented to the stakeholders introduced above, with their various perspectives taken into account.

2.5. Research Questions and Objectives

The research questions and objectives of this thesis, based on the aforementioned challenges, are as follows:

A. **Graph Representations of CityGML Documents**: Given the graph-based nature of GML and CityGML, CityGML documents are mapped onto graphs. These graph representations serve as the basis for all subsequent processes. The research questions during this phase include:

   RQA1. **Representation Comparison**: What distinct advantages does the use of graphs in this study offer compared to other representations of CityGML, such as relational and object-oriented representations?

   RQA2. **Related Graph Representations**: What existing graph representations are there for GML and CityGML, and why is there a need to develop a new method for mapping CityGML onto graphs?

   RQA3. **Graph Data Model for CityGML**: How can a graph data model be designed to accurately capture all substantial information available in the original CityGML documents? What are the prerequisites or limitations of this model?

   RQA4. **Mapping Methods and Evaluation**: How can CityGML objects from all fourteen thematic modules and five different levels of detail be mapped onto graphs? Which evaluation criteria are required to ensure that the resulting graph representations fully mirror all types of content and structure information from the original CityGML documents?

   RQA5. **XLink Resolution**: Where are the XLink elements from the original CityGML documents stored in the graphs, and how can they be resolved to restore the connectivity within the graphs?

   RQA6. **Compatibility with Graph Database**: What strengths and limitations does the employed graph database provide, and how can they be leveraged or mitigated to implement the mapping methods?

   RQA7. **Reconstruction of Graphs to CityGML Objects**: How can graph representations be reconstructed back into their original CityGML representation? What additional information must be enriched to the graphs to enable this reconstruction?

B. **Change Detection in CityGML Documents**: With the availability of a lossless graph mapping of CityGML documents, the comparison of two temporal versions of a CityGML document can be performed based on their graph representations. The research questions in this phase are as follows:

RQB1. **Related Comparison Methods**: What existing comparison methods are available for XML, GML, and CityGML documents, and why is there a need to develop a new method for comparing graph representations of CityGML documents?

RQB2. **Graph and Subgraph Isomorphism**: What is the nature of the graph and subgraph isomorphism problem, and why are they considered challenging for graph representations of CityGML documents? What optimization and heuristic adjustments can be incorporated to apply graph and subgraph isomorphism to the comparison of these graphs?

RQB3. **Advantages and Challenges of CityGML Graphs**: What benefits does the use of the semantic graph representations of CityGML documents provide to enhance their matching process? What are the limitations and challenges associated with using such graph representations?

RQB4. **Matching Methods**: How can the graph representations of CityGML documents be compared? How can it be ensured that all content and structure of the graphs are considered during matching? This includes:

a) Numeric values, measurements with units, and date-time values,

b) Generic attributes and generic attribute sets, and

c) Complex geometric objects, such as 0D points, 1D line segments, 2D polygons, and 3D solids.

RQB5. **Syntactic Ambiguities**: What strategies should be employed to handle the possible syntactic variations allowed by the GML and CityGML encoding standard to define the same objects, especially geometries?

RQB6. **Geometric Uncertainties**: How can small acceptable error tolerances for lengths, angles, areas, and volumes be accommodated while comparing numeric and geometric contents?

RQB7. **Geometric Transformations**: Since geometric objects can be moved, resized, or undergo any combination thereof, how can these transformations be detected in the graphs, with error tolerances taken into account?

RQB8. **Finding Best Match**: What strategies should be used to find the best match for any reference CityGML object, regardless of whether it is a text property or a geometric object, in one-to-many or many-to-many relationships? How can this be scaled to bigger datasets with a large number of potential matches?

RQB9. **Representation of Changes**: How can the detected changes be stored and represented in the same graph representations of CityGML documents? How to ensure that all relevant information and context can be quickly retrieved?

C. **Change Interpretation in CityGML Documents**: Utilizing the detected changes and their context in the graphs, hidden patterns are identified to derive meaningful interpretations. These interpretations are then evaluated based on the interests of various stakeholders. The research questions in this phase are as follows:

RQC1. **Existing Rule-based Systems**: What are the common rule-based systems used to define and model Event Condition Action (ECA) rules? What are the strengths and limitations of these systems in terms of describing the pattern rules among changes in the context of this study?

RQC2. **Hierarchical Modelling of Changes**: How can the changes detected in the previous step be modelled in a class hierarchy? What are these types of changes, and how can they be utilized for the definition of rules to detect change patterns?

RQC3. **Rule Definition for Change Patterns**: How can the rules for detecting semantic and aggregative patterns among changes be explicitly defined in a consistent manner? How can redundancies in interdependent rules, where one rule relies on the outcome of another, be eliminated? What can be incorporated to ensure that these rules can be easily adapted and extended to specific use cases?

RQC4. **Matching Change Patterns**: What strategy should be employed to match patterns among detected changes in the graphs based on the predefined rules given above? This involves further research questions such as:

a) How to avoid repeated processing of the same changes when evaluating pattern rules?

b) What approach should be employed to deal with rules, where the number of changes of a certain type required for the creation of the next interpreted change is only known in runtime? For instance, the pattern of a moved building requires that all its boundary surfaces have also been moved by the same offset. However, this number of boundary surfaces varies for each building.

c) Which sources of information can be leveraged to differentiate and process different types of changes that belong to different classes of objects using their graph representations?

d) How to calculate the scope of detected changes that indicates whether they belong to a global, clustered, or local pattern?

e) How to handle a pattern rule that relies on the outcomes of multiple other rules?

RQC5. **Managing Temporary Data**: Where can the temporary data employed during the interpretation process be stored in the graphs? This data includes the type of the next change to be created, the number of changes per type required by the current pattern, and the number of changes per type collected so far.

RQC6. **Change-Stakeholder Model**: How can the identified change patterns and their interpretations be evaluated based on the different perspectives of stakeholders? This involves further research questions such as:

a) What are the most relevant stakeholders and change types serving as examples in the context of this study?

b) How can the complex and hidden interrelationships between different types of changes and various group of stakeholders be explicitly modelled and described within a centralized network?

c) How can the model account for the varying interests of stakeholders in a change type over time?

d) Where in the model can the reasoning or the real-world actions that could have caused these changes be represented?

e) How can the model differentiate stakeholders and their specific roles in the city, especially when a stakeholder, such as an organization, can be assigned with multiple roles, with each role having distinct interests in different changes?

RQC7. **Graph-based Change-Stakeholder Analysis**: What analyses can be conducted based on the aforementioned change-stakeholder model? This involves further research questions such as:

a) How can it be quickly determined whether a given detected change is of interest to any stakeholder?

b) How can it be quickly determined which changes are of interest to a given stakeholder?

c) When multiple stakeholders share a common interest in a change, how can their differing levels of interest be evaluated against each other? And vice versa, when multiple changes hold relevance to the same stakeholder, how can their different relevance values be evaluated against each other?

D. **Optimization for Large CityGML Documents**: The aforementioned processes are further optimized to allow for the handling of large CityGML documents. The research questions in this phase are as follows:

    RQD1. **Memory Reduction**: How can the memory consumption of all processes discussed above be reduced to accommodate massive CityGML documents and a large number of changes?

    RQD2. **Database Indexes**: What types of thematic and spatial indexes are required, and how can they be initiated, populated, and utilized in this research?

    RQD3. **Transaction Management**: What characteristics do the transactions in the employed graph database exhibit, and how can they be optimized?

    RQD4. **Concurrency Control**: Are the transactions in the employed graph database suitable for execution in a multi-threaded environment? What mechanisms are used to prevent or avoid deadlocks when multiple concurrent threads attempt to modify a shared resource?

These research questions and objectives shall be addressed in the subsequent chapters of this thesis. Their overview can be found in Table 2.1.

Table 2.1.: An overview of all research questions (RQ) and their corresponding sections in this thesis.

RQ	**Section**	**RQ**	**Section**	**RQ**	**Section**	**RQ**	**Section**
RQA1	3.1	RQB1	4.1	RQC1	5.1	RQD1	6.1
RQA2	3.1	RQB2	4.1, 4.5	RQC2	5.2	RQD2	6.2, 6.3
RQA3	3.2	RQB3	4.2	RQC3	5.3	RQD3	6.4
RQA4	3.3, 3.5	RQB4	4.3, 4.5	RQC4	5.4	RQD4	6.4
RQA5	3.4	RQB5	4.5	RQC5	5.4		
RQA6	3.1	RQB6	4.5	RQC6	5.5		
RQA7	3.6	RQB7	4.5	RQC7	5.5		
		RQB8	4.5				
		RQB9	4.6				

2.6. Outline of the Thesis

Chapters 3 to 5 establish the core workflow of this thesis. An overview of this workflow is provided in Figure 2.4.

Chapter 3 introduces the use of semantic graphs in this study, illustrating their ability to accurately represent the content and structure of CityGML documents. These graph representations serve as the basis for all subsequent processes.

Chapter 4 discusses the graph and subgraph isomorphism problem, offering heuristic graph matching methods that can leverage the semantic and geometric nature of the employed graph representations of CityGML documents. These proposed methods account for the syntactic ambiguities in GML and CityGML, while being able to quickly determine the best match for each reference CityGML object.

Chapter 5 addresses the challenges of understanding the detected changes from the perspectives of stakeholders. A rule network is proposed to define all rules for detecting patterns among changes in one place, effectively eliminating the redundancies caused by interdependent rules. Then, a change-stakeholder network is introduced, offering a graph-based approach to explicitly describe the hidden interrelationships between changes and stakeholders. Path-tracing techniques within the network are presented that are capable of quickly determining the relevance of a given change to any stakeholders, and conversely, identifying changes relevant to a specific stakeholder.

Chapter 6 approaches the scalability challenges posed by massive CityGML documents and provides various strategies to minimize the memory consumption and enhance parallelism across all processes introduced in this thesis.

Chapter 7 applies the proposed processes to real-world datasets, evaluates, and interprets their results. It provides additional details about the implementation and offers guidelines and examples on how to employ these processes and further analyse the results.

Finally, Chapter 8 summarizes the work of this thesis and discusses its applicability, extendability, and contributions.

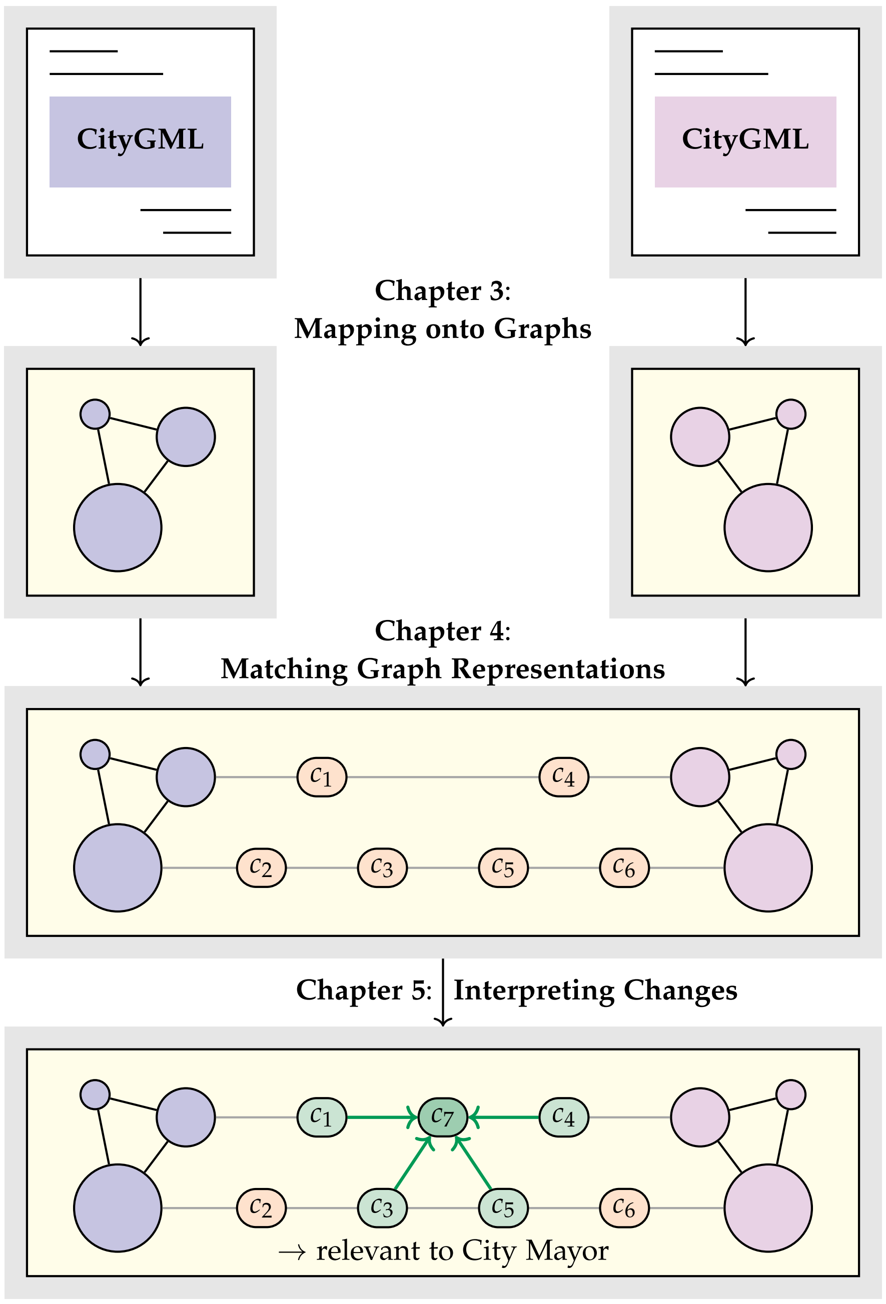


Figure 2.4.: An overview of the workflow presented in this thesis. Firstly, two temporal versions of a CityGML document are mapped onto graphs (Chapter 3). Then, these graphs are compared to identify changes (Chapter 4). Lastly, change patterns are matched to provide relevant interpretations to stakeholders (Chapter 5). All processes take place in the same graph database.

3. Graph Representation of Semantic 3D City Models

The comparison of two City Geography Markup Language (CityGML) documents requires knowledge about not only the content but also the structure of their elements, which cannot be provided using the text format of these inputs. Despite being encoded as text files, the CityGML data model exhibit network-like structures. Therefore, to enable the matching of CityGML documents, they must first be transformed into their equivalent graph representations. The process of mapping these documents is explained in this chapter.

The content of this chapter substantially expands upon the author's earlier publications, which are detailed as follows:

1. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2017, October). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database [12th International 3D GeoInfo Conference 2017, University of Melbourne, Melbourne, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 99–106, Vol. IV-4/W5). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-iv-4-w5-99-2017.

2. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2018). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database. In gis.Science (pp. 85–100, Vol. 3). Wichmann Verlag. https://gispoint.de/artikelarchiv/gis/2018/gisscience-ausgabe-32018.html.

3.1. Foundations and Related Work

This section establishes the necessary groundwork and discusses the literature relevant to the development of the concepts and methods presented in this chapter. Research Questions RQA1, RQA2, and RQA6 (Representation Comparison, Related Graph Representations, and Compatibility with Graph Database, respectively) are addressed in this section.

3.1.1. The City Geography Markup Language (CityGML)

CityGML is an open-source information model for representing, storing, and exchanging semantic 3D city models (Gröger et al., 2012). It has been an international standard of the Open Geospatial Consortium (OGC) since 2008, with version 1.0 released in 2008 and version 2.0 in 2012. To date, CityGML is the most commonly used exchange format for semantic 3D city models worldwide and has become an important basis for the development and application of urban digital twins. In 2021, the OGC approved CityGML version 3.0 as the next evolution of the data model.

This thesis primarily focuses on CityGML version 2.0, which is currently the most widely adopted version with the largest number of available datasets worldwide. However, the generic approach for mapping CityGML documents onto graphs, as will be explained later in this chapter, is also applicable to version 3.0. Henceforth, any reference to CityGML in this thesis shall be understood as referring to CityGML version 2.0, unless a version is explicitly specified.

CityGML is an application schema of GML version 3.1.1 (Cox et al., 2004), which is an Extensible Markup Language (XML) grammar defined by the OGC in accordance with ISO/TC 211 standards for the purpose of describing geographical features. In contrast to conventional 3D city models, which primarily focus on graphical or geometrical representations, CityGML enables the storage of semantic, topological, geometrical, and appearance information of city objects in one place. The following sections provide a brief overview of some selected important aspects of CityGML that are most relevant to the scope of this thesis.

Semantic Modelling

One of the most prominent features and biggest strengths of CityGML, compared to other data models, is its fundamental modelling paradigm that enriches every element with a **semantic context**. Each element is defined by its unique meaning, role, and type, allowing for direct differentiation from elements of other types. These semantic types are systematically organized into classes within a well-defined hierarchical structure. This detailed schema enables non-redundant and comprehensive definitions of objects. This is achieved by leveraging common object-oriented principles such as inheritance, polymorphism, and encapsulation. Moreover, each object class is enriched with both predefined thematic properties, such as the measured height and number of storeys of a building, and other generic attributes that can be defined on an ad-hoc basis.

Another significant semantic feature extensively utilized in CityGML is the use of **aggregations**. An object defined within an aggregation relationship can contain smaller components or be a part of a larger element. For instance, the 3D shape of a building part is an aggregation of all its boundary surfaces, while the building part itself, along with other building parts, belongs to a larger building. These relationships not only provide a comprehensive understanding of the city model but also aid in interpreting their changes, as will be explained later in Chapter 5.

Multi-scale Modelling

CityGML allows for the modelling of city objects with regard to both their geometric and thematic contents using five different Level of Details (LODs), ranging from LOD0 (the lowest) to LOD4 (the most detailed). The LOD0 representation of a building is its footprint or roof edge polygon, while LOD1 is a prismatic block with flat roofs. In LOD2, roof structures are added, and boundary surfaces are differentiated between roof, wall, and ground surfaces. LOD3 adds further details to the architectural structures, such as balconies and openings. Finally, LOD4 adds interior structures, such as rooms, stairs, and furniture. Multiple LODs can exist simultaneously for a single object.

Modularization

The CityGML data model is thematically organized into a *core module* and thirteen *extension modules*. The core module defines the basic concepts and components used by the extension modules, while each of the extension modules covers a specific area of semantic 3D city objects. These areas include *Appearance*, *Bridge*, *Building*, *CityFurniture*, *CityObjectGroup*, *Generics*, *LandUse*, *Relief*, *Transportation*, *Tunnel*, *Vegetation*, *WaterBody*, and *TexturedSurface* (which is now deprecated) (Gröger et al., 2012). This structure allows for a highly flexible and modular representation of most important types of city objects. A Unified Modelling Language (UML) package diagram illustrating this modular structure is given in Figure 3.1.

Coherent Spatio-Semantic Modelling

CityGML integrates two distinct yet interrelated types of hierarchies: *semantic* and *geometric*. At the semantic level, CityGML represents real-world city objects as features, each with its own attributes and relationships to other features. For example, a building consists of several boundary roof, wall, and ground surfaces. The relationships between these features thus form an aggregation, with boundary surfaces being parts of the building. On the other hand, at the spatial level, objects may be assigned with geometries that represent their locations and shapes. A building, for instance, is represented by a 3D solid, while its boundary surfaces are represented as polygons forming the sides of the solid. CityGML combines both the semantic and geometric hierarchy by linking semantic objects with their corresponding geometries (Gröger

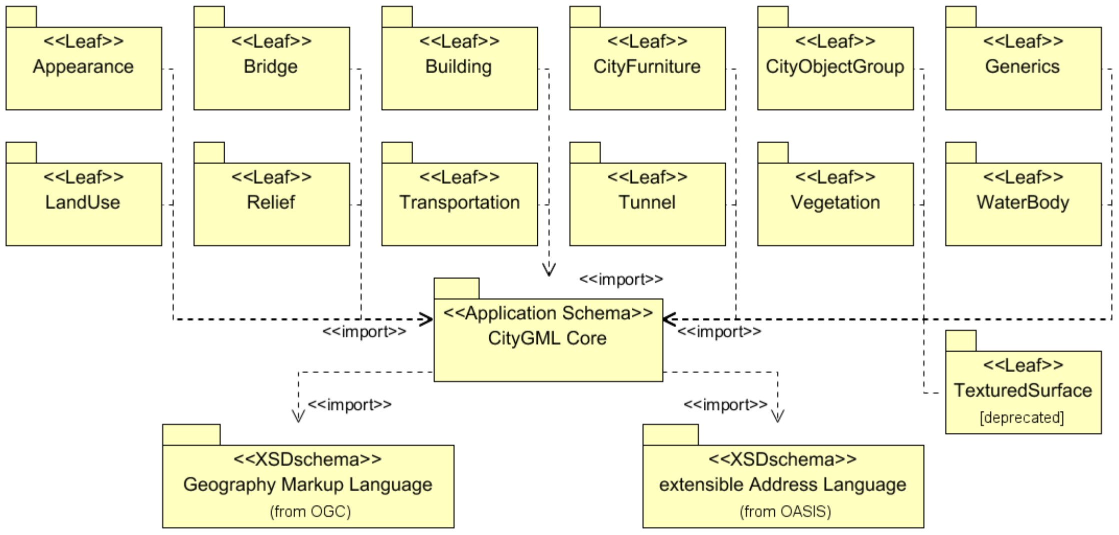


Figure 3.1.: An overview of the modular structure of CityGML thematic modules and their schema dependencies, represented as a UML package diagram. All extension modules, denoted by «*Leaf*», depend on the core module, denoted by «*Application Schema*». The core module, in turn, depends on the schema definition of GML, denoted by «*XSDschema*». These dependencies are indicated by the «*import*» relations. This diagram was taken from (Gröger et al., 2012).

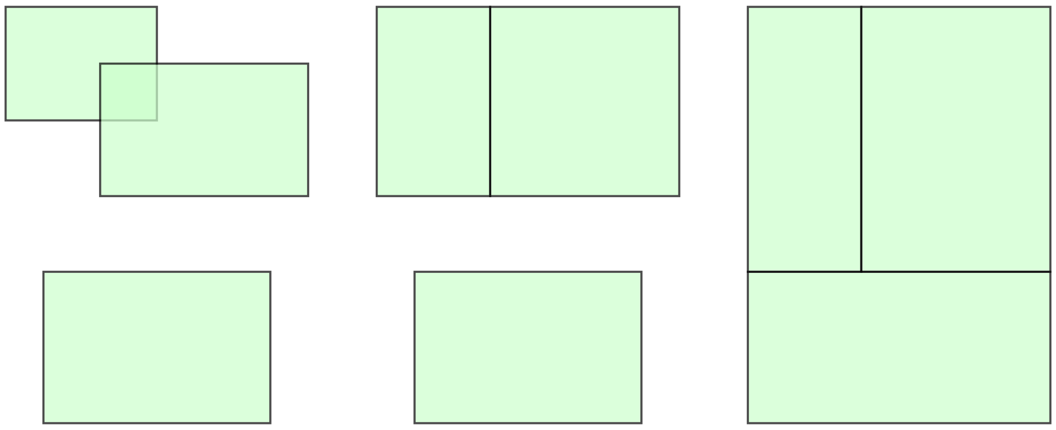
et al., 2012; Stadler & Kolbe, 2007). This coherent spatio-semantic modelling enables seamless navigation between both hierarchies, allowing for more complex analyses.

Geometric-topological Modelling

Geometries in CityGML are defined using a subset of the geometry model provided by the Geography Markup Language (GML). The GML model consists of geometric primitives and combined geometries that are composed of primitives.

A geometric primitive is defined for each dimension: *Point* in 0D, *_Curve* in 1D, *_Surface* in 2D, and *_Solid* in 3D. GML employs the Boundary Representation (B-Rep) (Mäntylä, 1988) to represent its geometries. For instance, a 3D solid is bounded by 2D surfaces that form a closed volume, and a 2D surface is bounded by 1D curves that form a closed area. In CityGML, curves are limited to being straight lines only, meaning only the class *LineString* is used for 1D primitives. In addition, surfaces in CityGML are represented by the class *Polygon*, where the boundary and interiors of each polygon must be located on the same plane.

Geometric primitives can be combined to form *aggregates*, *complexes*, or *composites*, each with its own unique topological characteristics. An aggregate, such as a *MultiPoint*, *MultiCurve*, *MultiSurface*, or *MultiSolid*, is a collection of corresponding geometric primitives without any topological constraints. This means that the components can be disjoint, touching, or even overlapping. In contrast, the components of a geometric complex must not overlap and can only be disjoint or touch at their boundaries. A composite is a complex that requires its components to additionally be of the same dimension and connected along their boundaries. Composites are thus available in one-dimensional space or higher and are represented by *CompositeCurve*, *CompositeSurface*, or *CompositeSolid*. These are illustrated in Figure 3.2.



(a) *MultiSurface* (b) *GeometricComplex* (c) *CompositeSurface*

Figure 3.2.: The topological differences between geometric aggregates (left), complexes (middle), and composites (right) in 2D in CityGML. Adapted from (Gröger et al., 2012).

CityGML also allows for the explicit modelling of topological relationships between geometries that share a common boundary. These shared boundaries can be defined once and then be referenced using the XML Linking Language (XLink) (W3C, 2006), a concept provided by XML for interlinking objects. For instance, two adjacent rooms of a building may be separated by a shared wall, which can first be defined in the first room and then referenced by the second room using its identifier. This not only reduces redundancy but also establishes the explicit topological relationships between neighbouring geometries.

Listing 3.1 shows an example of a building model in CityGML. This is a simplified excerpt from the open-source dataset *FZK-Haus* (KIT IAI, 2017) presented in LOD2. The 3D shape of the building is represented by an element *Solid*, which contains a

CompositeSurface that consists of seven contiguous boundary surfaces. These include two roof surfaces, four wall surfaces, and one ground surface, which have been defined separately outside of this solid in the CityGML dataset. Thus, the solid does not redefine the contents of these boundary surfaces. Instead, it references the already existing boundary surfaces using XLinks. A 3D visualization of this *FZK-Haus* model can be found in Figure 3.3.

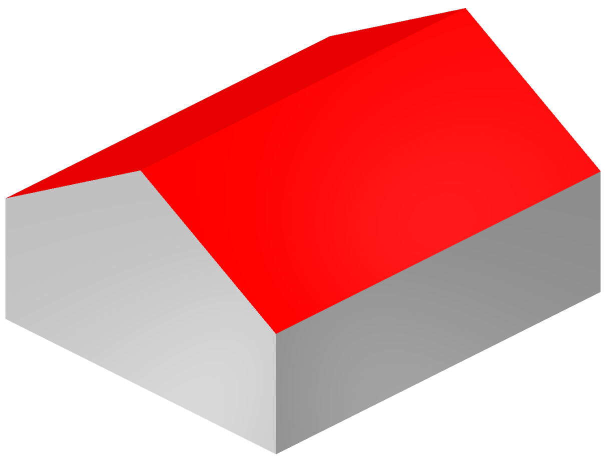


Figure 3.3.: An illustration of the CityGML building model *FZK-Haus* in LOD2. As outlined in Listing 3.1, the building has two roof surfaces, four wall surfaces, and one ground surface. Therefore, the building's solid contains seven XLinks referencing these surfaces.

As previously mentioned in Section 2.1.2, XLinks are not only employed to describe the explicit topological relationships among geometric objects, but also to define city objects with implicit geometries, such as trees and traffic lights, which share the same prototypical shape. Additionally, XLinks can be found in city object groups.

As will be shown later in Section 6.1, XLinks play a pivotal role in reassembling a massive CityGML dataset that has been divided into smaller pieces, each marked with a unique identifier. These identifiers are crucial for the XLink resolution process to accurately relocate each segment.

```
...
<core:cityObjectMember>
  <bldg:Building gml:id="FZK_HAUS_LOD2">
    <bldg:lod2Solid>
      <gml:Solid>
        <gml:exterior>
          <gml:CompositeSurface>
            <!-- XLink references to the surfaces -->
            <gml:surfaceMember xlink:href="#Roof_Surface_1_North"/>
            <gml:surfaceMember xlink:href="#Roof_Surface_2_South"/>
            <gml:surfaceMember xlink:href="#Outer_Wall_Surface_1_West"/>
            <gml:surfaceMember xlink:href="#Outer_Wall_Surface_2_South"/>
            <gml:surfaceMember xlink:href="#Outer_Wall_Surface_3_East"/>
            <gml:surfaceMember xlink:href="#Outer_Wall_Surface_4_North"/>
            <gml:surfaceMember xlink:href="#Ground_Surface"/>
          </gml:CompositeSurface>
        </gml:exterior>
      </gml:Solid>
    </bldg:lod2Solid>
    <!-- Content of Roof Surface 1 (North) -->
    <bldg:boundedBy>
      <bldg:RoofSurface gml:id="...">
        <bldg:lod2MultiSurface>
          <gml:MultiSurface>
            <gml:surfaceMember>
              <gml:Polygon gml:id="Roof_Surface_1_North">
                ...
              </gml:Polygon>
            </gml:surfaceMember>
          </gml:MultiSurface>
        </bldg:lod2MultiSurface>
      </bldg:RoofSurface>
    </bldg:boundedBy>
    ... <!-- Content of other boundary surfaces -->
  </bldg:Building>
</core:cityObjectMember>
    ...
```

Listing 3.1: An excerpt of the CityGML building model *FZK-Haus* in LOD2.

3.1.2. Object-oriented Representations of CityGML Models

The CityGML data model employs object-oriented principles to organize city objects into classes, making extensive use of concepts such as inheritance, polymorphism, and encapsulation. These definitions are specified in the XML Schema Definition (XSD) of CityGML, to which all CityGML documents must adhere.

As a result, the object-oriented representations of CityGML objects are the most accurate and closest form to the source model in terms of both structure and content. However, these representations are stored entirely in main memory, leading to high memory consumption in large city models. Therefore, in many real-world applications and use cases, these in-memory object-oriented representations are often used as a basis for further transformation to other forms or representations, such as tables or graphs, as will be discussed later in this chapter. This approach ensures minimal information loss during conversion from the original CityGML documents to the target representation.

This conversion process can be implemented in several popular programming languages, such as Java, TypeScript, and Python. Among these, Java is selected for the implementation of this thesis due to its strong support for object-oriented concepts and high performance in handling multi-threaded operations, especially when dealing with massive input datasets.

Unmarshalling XML Documents

In computer science, marshalling refers to the process of converting the state of an object from its in-memory representation into a data format that can be stored or transmitted (Grochowski et al., 2020). The reverse process is called unmarshalling. While many programming languages regard marshalling (or unmarshalling) as a specific type of serialization (or deserialization) (Python, 2023; Ryan et al., 1999), in some contexts, these terms are used interchangeably (Grochowski et al., 2020).

Therefore, by unmarshalling CityGML documents, their object-oriented representation can be produced. Many programming languages provide their own frameworks to marshal or unmarshal XML files, such as *XMLSerializer* in C# and Jakarta XML Binding (JAXB) in Java.

The software library JAXB allows for the automatic mapping of Java classes from a set of given XSD files. This capability enables three key operations:

1. **Unmarshalling** XML documents into corresponding Java objects,
2. **Processing and updating** the Java objects, and
3. **Marshalling** Java representations into XML documents.

Table 3.1 shows a list of some example XML schema built-in data types and their corresponding types in Java (Kawaguchi et al., 2017).

XML Schema Data Type	**Java Data Type**
xsd:string	*java.lang.String*
xsd:integer	*java.math.BigInteger*
xsd:int	*int*
xsd:long	*long*
xsd:short	*short*
xsd:decimal	*java.math.BigDecimal*
xsd:float	*float*
xsd:double	*double*
xsd:boolean	*boolean*
xsd:byte	*byte*

Table 3.1.: Mapping between some XML schema data types and Java data types (Kawaguchi et al., 2017).

Unmarshalling CityGML Documents

In this thesis, to unmarshal CityGML documents into their corresponding object-oriented representations, the software tool *citygml4j*[1] is used. *citygml4j* is an open-source Java library and Application Programming Interface (API) for reading, processing, and writing CityGML datasets. At its core, the software employs JAXB to automatically generate Java classes for CityGML objects based on their corresponding XSD files. Some of the key features provided by *citygml4j* include:

1. Support for all major CityGML versions 1.0, 2.0, and 3.0 according to their conceptual model standards (Gröger et al., 2008, 2012; Kolbe et al., 2021),

2. Ability to divide large CityGML documents into smaller chunks,

3. Capability to marshal to and unmarshal from CityGML documents,

4. Option to transform, process, and update city objects, and

5. An API for extending support for CityGML Application Domain Extensions (ADEs) in applications.

[1]https://github.com/citygml4j/citygml4j

Figure 3.4 gives an overview of the contents of an in-memory building object in CityGML, as parsed by *citygml4j*. This mapping corresponds to the conceptual modelling of the building module of CityGML (Gröger et al., 2012). Although some presented properties are declared as *private* in their defining classes and can only be accessed in subclasses via *getter* methods, for simplicity purposes, these properties are shown instead of their corresponding *getter* methods.

The major advantage of the object-oriented representations lies in their precision in depicting the contents and relationships of CityGML objects. As will be explained later in this chapter, both the relational and graph representations of CityGML models are often constructed based on these object-oriented representations.

However, object-oriented representations are held entirely in main memory, meaning that (1) their contents will be lost after execution, and (2) a large main memory capacity is necessary for handling large CityGML datasets.

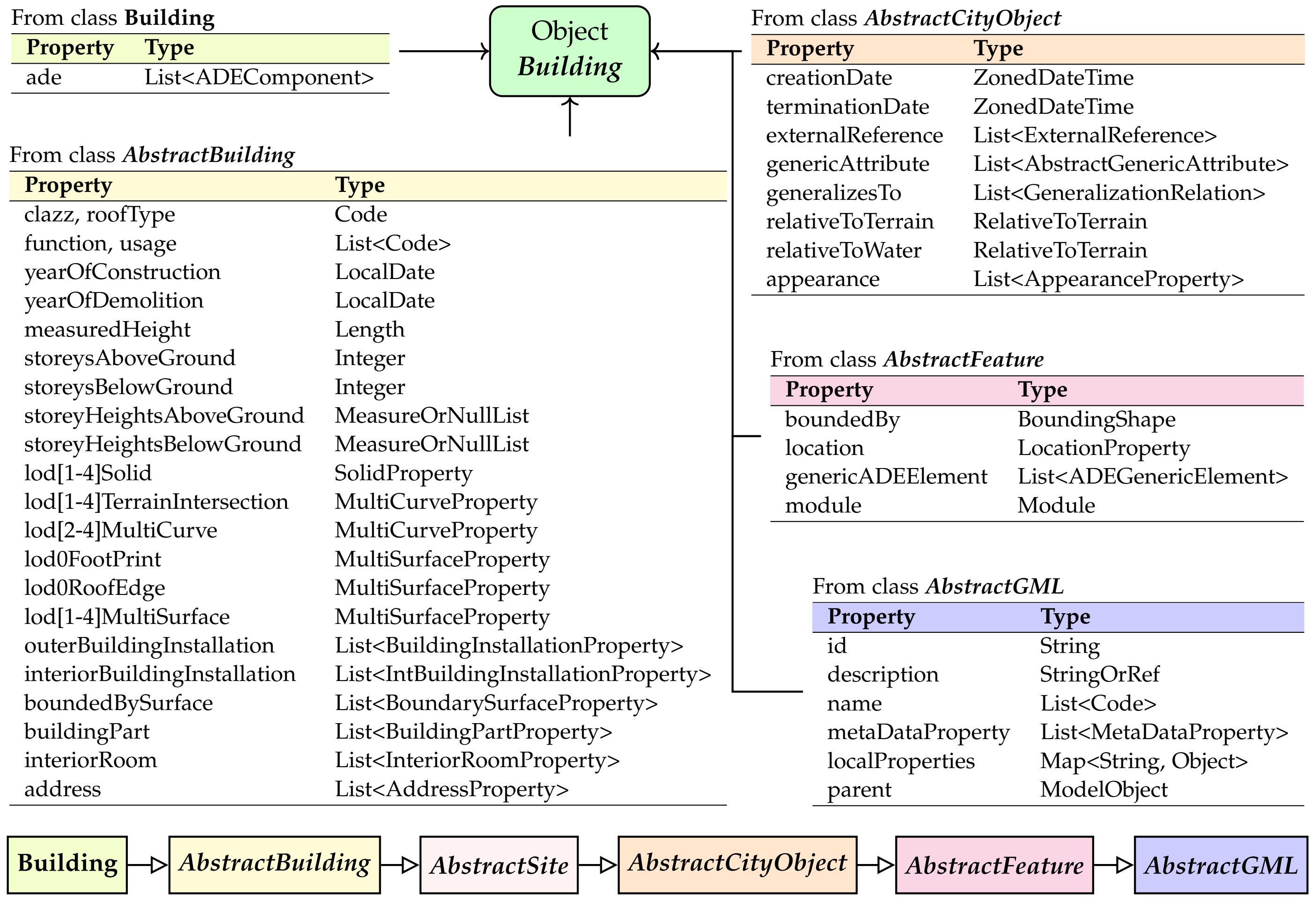
While alternative representations, such as those using a relational or a graph database, may capture less information and require additional efforts for mapping, they offer the advantage of persistent storage of city objects. This not only allows for data reuse in subsequent applications but also reduces memory consumption significantly.

3.1.3. Relational Representations of CityGML Models

Since the release of CityGML, there has been a rapid increase in both the number and scale of virtual semantic 3D city models. This has led to a growing need for efficient storage, management, and analysis of large CityGML documents, which is often achieved through the use of databases.

Relational databases have long been a dominating workhorse in the data world and are widely used in various application fields. A relational model represents all data as a collection of relations (or tables), each consisting of a set of tuples (or rows) and attributes (or columns) (Codd, 1970). Since its introduction (Chamberlin & Boyce, 1976), Structured Query Language (SQL) has become the standard query language in modern Relational Database Management Systems (RDBMSs) (Davoudian et al., 2018). Many studies have therefore explored the use of an RDBMS for storing and managing XML and GML documents in general, and CityGML documents in particular.

To convert XML documents into tables, most relational databases define a mapping between the corresponding XML document schemas, such as Document Type Definition (DTD) and XSD, and the database schemas. Extensible Stylesheet Language Transformations (XSLT) may be used in advance to transform input XML documents to a structure specifically required by the mapping. There exist two common types of mapping: table-based mapping, and object-relational mapping (Bourret, 2005). The table-based mapping represents XML documents using a single table or a set of tables,



From class **Building**

Property	Type
ade	List<ADEComponent>

From class *AbstractBuilding*

Property	Type
clazz, roofType	Code
function, usage	List<Code>
yearOfConstruction	LocalDate
yearOfDemolition	LocalDate
measuredHeight	Length
storeysAboveGround	Integer
storeysBelowGround	Integer
storeyHeightsAboveGround	MeasureOrNullList
storeyHeightsBelowGround	MeasureOrNullList
lod[1-4]Solid	SolidProperty
lod[1-4]TerrainIntersection	MultiCurveProperty
lod[2-4]MultiCurve	MultiCurveProperty
lod0FootPrint	MultiSurfaceProperty
lod0RoofEdge	MultiSurfaceProperty
lod[1-4]MultiSurface	MultiSurfaceProperty
outerBuildingInstallation	List<BuildingInstallationProperty>
interiorBuildingInstallation	List<IntBuildingInstallationProperty>
boundedBySurface	List<BoundarySurfaceProperty>
buildingPart	List<BuildingPartProperty>
interiorRoom	List<InteriorRoomProperty>
address	List<AddressProperty>

From class *AbstractCityObject*

Property	Type
creationDate	ZonedDateTime
terminationDate	ZonedDateTime
externalReference	List<ExternalReference>
genericAttribute	List<AbstractGenericAttribute>
generalizesTo	List<GeneralizationRelation>
relativeToTerrain	RelativeToTerrain
relativeToWater	RelativeToTerrain
appearance	List<AppearanceProperty>

From class *AbstractFeature*

Property	Type
boundedBy	BoundingShape
location	LocationProperty
genericADEElement	List<ADEGenericElement>
module	Module

From class *AbstractGML*

Property	Type
id	String
description	StringOrRef
name	List<Code>
metaDataProperty	List<MetaDataProperty>
localProperties	Map<String, Object>
parent	ModelObject

Figure 3.4.: The contents of a building object in CityGML parsed by *citygml4j*. The properties are grouped by their respective defining classes. The hierarchical structure of these classes is shown at the bottom.

where the XML elements must be given in a specific structure required by the mapping. On the other hand, the object-relational mapping represents complex XML elements as class instances, while simple elements or attributes are represented as scalar properties of these instances. Classes are then mapped to tables, and scalar properties to columns. Complex properties used as references to other objects are stored as pairs of primary and foreign keys in respective tables. Some popular relational databases that support XML include Oracle Database (Murthy et al., 2005), IBM Db2 (Nicola & van der Linden, 2005), PostgreSQL (Obe & Hsu, 2017), and Microsoft SQL Server (Rys, 2005).

GML is an XML grammar that can capture a wide range of geographic information, including geometry, topology, features, coordinate reference systems, units of measurement, and time (Cox et al., 2004). Databases that support the storage and management of geographic information are known as geographic databases (or geo-databases), which are a subset of spatially-enhanced databases (or spatial databases). Many modern XML-enabled relational databases are also capable of handling GML documents, either natively or using extensions. For instance, the extension PostGIS for PostgreSQL allows for storing and managing simple GML data within the database.

As an application schema of GML for modelling and storing most common 3D urban objects (Gröger et al., 2012), CityGML documents can also be represented in a relational database. For example, the database cjdb (Powałka et al., 2024), a compact relational data model designed for PostgreSQL, allows for the storage of CityGML models by utilizing CityJSON (Ledoux et al., 2019), a JavaScript Object Notation (JSON)-based encoding of a subset of CityGML. The most prominent representative in this domain, however, is the 3D City Database (3DCityDB), an open-source high-performance 3D geo-database solution for importing, managing, analysing, visualizing, and exporting semantic 3D city models in CityGML (Stadler et al., 2009; Yao et al., 2018). The software is equipped with an import and export tool capable of handling very large CityGML datasets. Additional functionalities are available, such as for converting relational representation of city objects to common data formats used for visualization purposes, and for managing use-case specific ADEs in CityGML (Yao & Kolbe, 2017). The entire software package makes use of a single relational interface on top of a spatially-enhanced Oracle or a PostgreSQL/PostGIS relational database instance.

3.1.4. The Concept of Graphs and the Graph Data Structure

The earliest recorded use of graphs dates back to 1736, when the Swiss mathematician Leonhard Euler utilized them to solve the now-classical problem of the 'Seven Bridges of Königsberg' (Euler, 1736, 1995). Euler's solution to this problem is widely recognized as the first true proof in the theory of networks (Newman, 2003), laying the foundations for the development of topology and graph theory (Shields, 2012).

Graphs as a Mathematical Concept

In discrete mathematics, particularly graph theory, a graph $G = (V, E)$ is a mathematical structure that consists of a set V of vertices (or nodes) and a set E of edges. An edge that connects two vertices is described as being incident to those vertices. The vertices connected by an edge are referred to as adjacent. Similarly, two edges that share a common vertex are also referred to as adjacent.

Edges can be directed or undirected. A directed edge, denoted as $e = (v_1, v_2)$, connects the start vertex v_1 to the end vertex v_2. On the other hand, an undirected edge, denoted as $e = \{v_1, v_2\}$, does not distinguish between start and end vertices. Thus, an undirected edge can be represented as a pair of two oppositely directed edges: $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_1)$. A directed graph contains only directed edges, while an undirected graph contains only undirected edges. In this thesis, the graph representations of CityGML models are **directed**, since CityGML is an object-oriented data model and the relations between CityGML objects are directed.

The definitions above do not permit a graph to have multiple edges (or multi-edges) between the same pair of vertices. Such graphs are called simple graphs. However, in some studies, the edge set may be extended to be a multiset, allowing multiple edges between the same pair of vertices. These graphs are called multigraphs. On the other hand, the aforementioned definitions allow the existence of edges that connect a vertex to itself, known as loops. Loops and multi-edges are rarely employed in practice (West, 2000). Thus, the graph data model for CityGML presented in this thesis are a **simple, directed graph** that excludes multi-edges (with identical contents) and loops.

An undirected graph is considered connected if there exists a sequence of adjacent undirected edges connecting every pair of its vertices. Similarly, a directed graph is strongly connected if there exists a sequence of adjacent directed edges connecting every pair of vertices. A directed graph is weakly connected if it can be transformed into a connected undirected graph by treating all directed edges as undirected.

The graph mapping algorithm for CityGML documents introduced in this chapter ensures that the resulting graph representations of CityGML documents are **weakly connected**, as all elements of a CityGML model can be reached from a single root element, the city model. Furthermore, since the use of XLinks can cause circular references or cycles that prevent processes from terminating, the methods for mapping CityGML documents onto graphs must eliminate directed cycles in their results.[2] Thus, the graph representations of CityGML documents employed in this thesis are **Directed Acyclic Graphs (DAGs)**.

[2]The processes proposed in this thesis for mapping CityGML elements onto graphs and reconstructing graphs into CityGML objects support directed cycles. However, for the matching and interpretation process to terminate, the graphs must be acyclic, as observed in all CityGML 2.0 datasets tested so far.

Graphs as an Abstract Data Type (ADT)

In the field of computer science, a graph data structure is an Abstract Data Type (ADT) that implements the mathematical concepts of graphs introduced above. In a graph data structure, vertices are stored in a finite set, while edges are represented as either an ordered or unordered pair of corresponding incident vertices, depending on whether the underlying graph is directed or undirected, respectively. These edges often employ references to existing vertices, such as their unique identifiers or indices within the vertex set.

A graph data structure typically provides a range of common operations, including graph construction and manipulation (such as inserting and removing vertices and edges), connectivity control (such as verifying the adjacency between two vertices or identifying all vertices adjacent to a vertex), and attribute manipulation (such as retrieving and updating properties stored in vertices and edges) (Goodrich & Tamassia, 2014).

Two of the most commonly used representations of graphs are adjacency matrices and adjacency lists. In an adjacency matrix, the value of each entry reflects the connectivity of the corresponding vertices denoted by the row and column index. In an adjacency list, all adjacent vertices of a specific vertex are stored in a linked list assigned for that vertex (Horowitz & Sahni, 1983). Adjacency matrices are more suited for dense graphs (where almost every pair of vertices is connected), while adjacency lists are more suited for sparse graphs (where the majority of possible edges are unset) (Goodrich & Tamassia, 2014).

As shown later in this chapter, the graph representations of CityGML datasets can be considered as sparse graphs, as most elements are clustered within their respective top-level features, such as buildings. Therefore, an adjacency list can be used to capture the structure and contents of the graph representations of CityGML models. However, other types of representations can be employed. This choice of representation for a graph is typically handled internally by the underlying graph database.

3.1.5. The Graph-based Nature of CityGML

In the early 1980s, the increasing popularity of the object-oriented programming paradigm led to several issues observed in the fields of Computer-aided Design (CAD), Computer-aided Manufacturing (CAM), and Geographic Information System (GIS) when using relational databases. The relational data model has limited capabilities for modelling complex objects (Davoudian et al., 2018), and an *object-relational impedance mismatch* may arise (Ireland et al., 2009). This mismatch occurs when data is stored in a relational database but used by an object-oriented programming language, due to the fundamental differences in how the data is represented between the two logic models. For example, objects in an object-oriented programming language can reference each other using object references, forming a directed graph. In contrast, a relational data model organizes data in tuples and relations, which are linked together by *JOIN* operations based on relational algebra (Codd, 1970). Since (natural or inner) *JOIN* operations are symmetric, these links are bidirectional, forming an undirected graph. Such conflicts involve many common object-oriented concepts, such as encapsulation, accessibility, inheritance, polymorphism, and associations between classes and data types that are more complex than those defined in SQL (Keller, 1997), as in the case of the data models in XML, GML, and specifically CityGML. The 3D City Database (3DCityDB) provided a database schema that effectively addressed these challenges, while maintaining a relatively compact database structure and adhering to the naming conventions of the CityGML data model. However, the design process for this schema was highly manual.

Relational databases have proven to be highly effective in storing and managing large-scale semantic 3D city models given in CityGML. However, the objectives of this thesis extend beyond simply representing and storing CityGML datasets. It also aims to further detect and interpret changes in the data, which requires frequent access to the interrelationships between city objects. Due to the complex hierarchical structure of CityGML elements and their multi-level deep relationships, a relational data model would require a significant number of *JOIN* operations among many tables, rendering it impractical for this purpose.

In this context, non-relational databases, also known as NoSQL databases, are often employed as an alternative to their relational counterparts. A wide range of non-relational databases exist to date, each with its own data models and applications, including key-value store, tuple store, document-oriented store, object-oriented database, and graph database. Non-relational databases do not follow the general principles that are enforced by relational databases. For example, non-relational databases are schema-less and thus more flexible in their ability to adopt structured, semi-structured, and unstructured data models (Davoudian et al., 2018). This eliminates the need for prior investments in the database schemas, which are typically required in relational databases. In addition, instead of relying on SQL, non-relational databases often employ their own native query languages, omitting costly *JOIN* operations across tables. Many non-relational databases trade off some of their Atomicity, Consistency, Isolation, Durability (ACID) properties in exchange for improved scalability, availability, and latency, particularly in web applications. However, some non-relational databases, such as the graph database Neo4j, retain full ACID compliance.

While XML documents in general can be stored in a document-oriented database (Bourret, 2005), which represents entire documents in a single instance, graph databases are a more suitable choice for representing and managing CityGML documents due to the graph-based nature of the CityGML data model. A graph database employs the graph data structure as its fundamental concept to store information in the form of vertices (or nodes) and edges (or relationships), each having its own properties. The use of edges enables the explicit modelling and efficient querying of deep relationships within the data.

Graph databases are particularly well-suited for handling highly interconnected data (Davoudian et al., 2018). When compared to a relational database for counting the number of nodes at a specific depth reachable from a start node, the graph database Neo4j has been shown to deliver results up to ten times faster (Vicknair et al., 2010). The nodes and edges used in the experiment were arranged in a DAG, with payloads resembling the semi-structured data found in XML or JSON documents. Recent studies have also shown that while relational databases excel at operations such as grouping, sorting, and aggregation, graph databases outperform them in tasks such as joining properties across multiple entities, pattern matching, and path traversal (Cheng et al., 2019).

CityGML elements are defined in a graph-like structure, where the elements are represented as vertices and their interrelations as directed edges. The relationships between CityGML elements are complex, ranging from inheritance and polymorphism to aggregation and composition. In addition, these edges can form (undirected) cycles if XLinks are used to reference existing elements. All of these characteristics can be effectively captured in a single graph. Figure 3.5 shows a graph representation of the *FZK-Haus* building model, as outlined previously in Listing 3.1. In this figure, the polygon geometries (blue) are referenced by both the corresponding boundary surfaces (as the polygons are part of these surfaces) and by the element *CompositeSurface* (through the use of XLinks).

3.1.6. Existing Graph Representations for GML and CityGML

Early adaptations of graphs for CityGML include the use of a graph-based schema for storing, analysing and managing city objects (Falkowski & Ebert, 2009). The study employed a directed graph called *TGraph*, where vertices and edges are typed, ordered, and attributed. The model schema integrated many information aspects of CityGML, including geometry, topology, semantic, and appearance.

Later, the potential of storing the graph representations of complex data models like CityGML in a database was explored (Agoub et al., 2016). The difficulties of storing and managing well-defined objects, attributes, and relations using an RDBMS were discussed, and a lightweight method for mapping and storing objects of various OGC standards in a graph database, such as Neo4j and ArangoDB, was provided.

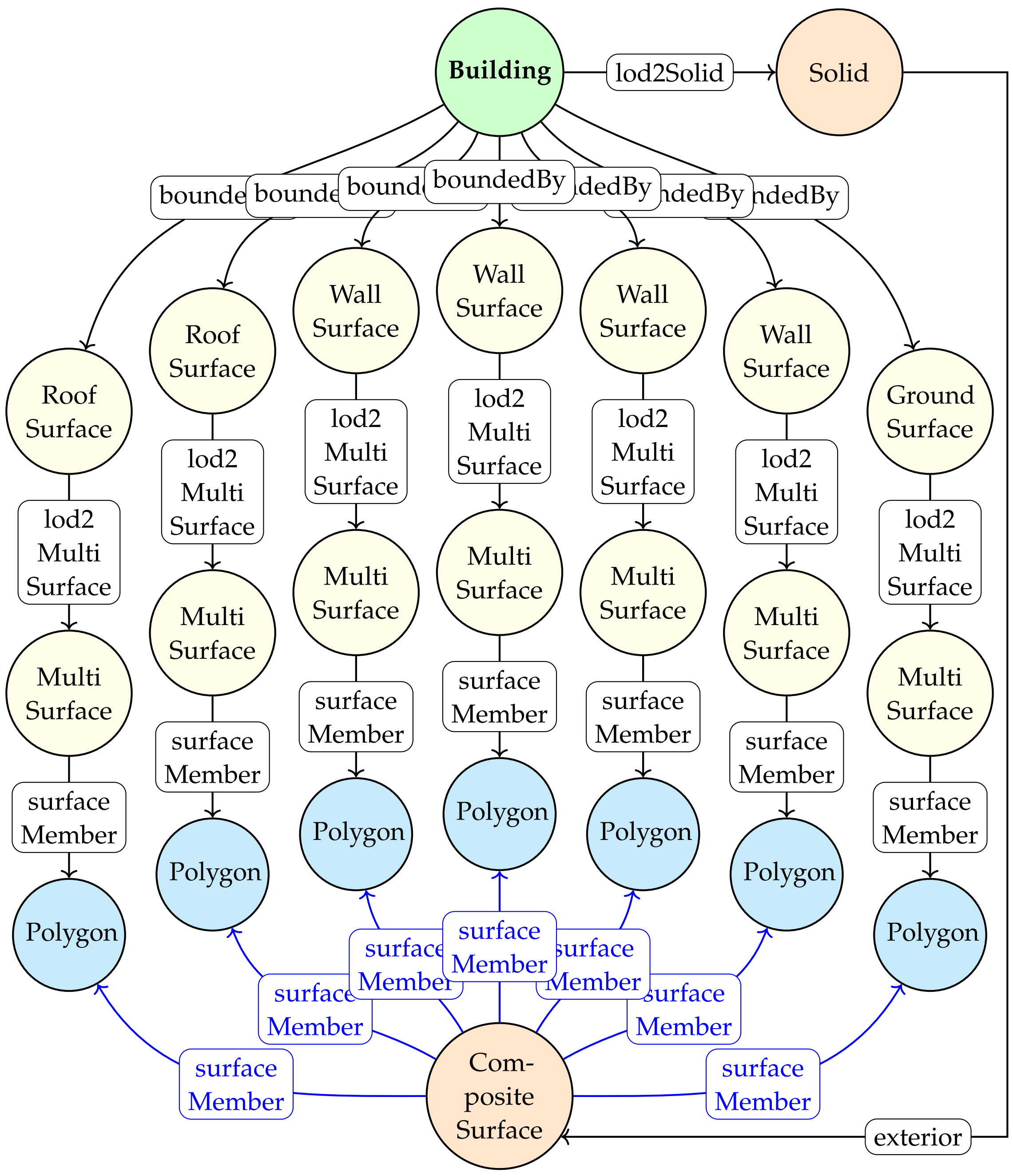


Figure 3.5.: A graph representation of the CityGML building model *FZK-Haus* given in Listing 3.1 and illustrated in Figure 3.3. The edges *surfaceMember* (blue) are produced by resolving all XLinks in the CityGML document.

On the other hand, the versatility and extendability of graphs were showcased in a study (Yao, 2020) aiming at enhancing the interoperability of CityGML ADEs with established spatial relational databases, such as the 3DCityDB (Yao et al., 2018). Graphs were used to represent both the XSD files of these ADEs and their associated transformation rules. The transformed graphs could then be converted to corresponding relational database schemas, allowing for the automatic integration of these ADEs.

A recent research (Ding et al., 2024) proposed a knowledge graph (Hogan et al., 2021) in Resource Description Framework (RDF) structure to represent CityGML datasets. This graph was built upon the 3DCityDB (Yao et al., 2018), utilizing an ontology designed for CityGML[3]. This approach enabled integration of external data, such as from OpenStreetMap (OSM), which could be queried using the OGC standard query language GeoSPARQL. The use of a knowledge graph and its associated query language simplified the query process, eliminating the need for complex SQL queries. However, the knowledge graph's structure may not fully align with the CityGML data model, as it was derived from the relational 3DCityDB. Moreover, the ontology was manually tuned and required manual extension, such as to incorporate properties present in the CityGML documents but absent in the ontology. Furthermore, the ontology and its associated mapping rules were designed for CityGML version 2.0 and would need to be revised to accommodate the newer version 3.0. An automated approach was later proposed (Vinasco-Alvarez et al., 2024) to transform the conceptual UML model of CityGML version 3.0 into Web Ontology Language (OWL). However, the resulting ontologies would require more complex rules to handle changes in cities.

In another study, graphs were used to represent IndoorGML documents, an OGC standard for indoor spatial information, to enhance indoor navigation efficiency (Jang et al., 2023). These labelled and attributed graphs were stored in the graph database Neo4j, providing an accurate representation of IndoorGML features and their interrelationships. The graph structure allowed for the application of graph algorithms, such as the Dijkstra's algorithm for determining the shortest paths from one node to all other nodes in the graph (Dijkstra, 1959). This could be applied to identify the optimal route between two rooms inside a building, a task that would have been difficult using a relational database. However, the authors did not directly convert input IndoorGML documents into graphs. Instead, IndoorGML documents were first converted to key-value pairs, similar to those in the JSON format. However, this structure had inherent limitations and could not capture all information available in XML and GML documents, inevitably leading to information loss without any control mechanism in place. Additionally, the employed rules for mapping nodes and relationships were manually defined for each object type, which could present challenges for adaptability and extendability.

[3]https://cui.unige.ch/isi/ke/ontologies

Semantic graphs, or knowledge graphs, have been proposed not only to represent, but also to store, manage, and compare CityGML documents. Recent studies (Nguyen et al., 2017; Nguyen, 2017; Nguyen et al., 2018) addressed the challenges of detecting changes between different temporal versions of the same CityGML document. This comparison was performed not directly between the CityGML text files but rather between their graph representations. The implementation for the mapping of CityGML objects onto graphs, stored in the graph database Neo4j as provided by these studies, was one of the first open-source implementations that could directly map arbitrarily large CityGML documents onto graphs. However, these mapping methods were implemented based on a set of mapping rules manually defined for specific object types of CityGML version 2.0, primarily covering only the core and building module.

Graphs are also employed in other application fields such as for the automatic conversion of the Industry Foundation Classes (IFC) exchange format in Building Information Modelling (BIM) to CityGML (Stouffs et al., 2018). This leveraged the expressiveness of graphs, essentially using them for transformation of data from or to the CityGML format. In another example, 'surface-line' graphs are utilized to represent 3D building models from various input exchange formats, such as OBJ, COLLADA, and CityGML (Mao & Li, 2019). These graph representations are then divided into subgraphs based on the adjacency properties of the building geometries, enabling detection of key semantic elements. This allows for sustainability analyses, particularly energy simulations.

The majority of the studies mentioned above employed graph representations to achieve various objectives, including the storage, management, analysis, visualization, transformation, and comparison of GML or CityGML datasets. They accomplished this by utilizing an abstraction or a subset of the underlying data model, such as CityGML, and manually defining graph mapping rules. This thesis, however, presents a universal approach that can (1) accurately map any objects available in both CityGML versions 2.0 and 3.0 across **all LODs and thematic modules**, ensuring minimal information loss, (2) **reconstruct** the created graph representations back into their original CityGML objects when required, and (3) accomplish these tasks **without any manual rules** or the need for conversion to an intermediate structure such as relational tables or the key-value pairs.

3.1.7. The Graph Database Neo4j

In this thesis, the graph database Neo4j is employed as a central tool for the persistent storage and database management of graph representations of CityGML datasets. Neo4j is a native graph database management system developed by Neo4j, Inc. It is one of the most popular and widely used graph databases worldwide (Fernandes & Bernardino, 2018). The core components of Neo4j are implemented in Java, but the database can be accessed from applications written in many other programming languages, such as TypeScript and Python, through the use of *drivers*. Neo4j is available in several products and licences. This thesis employs the open-source Neo4j Community Edition (Neo4j, Inc., 2023), which is licensed and distributed under GPL v3 (GNU, 2023).

In Neo4j, vertices and edges of a graph are stored as **nodes** and **relationships**, respectively. In this thesis, the term 'vertex' is used to refer to the mathematical concept in graph theory, while the term 'node' is used to refer to the corresponding entities in a graph database or network. Similarly, the term 'edge' refers to the mathematical concept, while the term 'relationship' refers to the corresponding entities in a graph database or network.

In Neo4j, nodes are identified by their **labels**, while relationships are distinguished by their **types**. Both labels and types serve the same purpose of categorizing nodes and relationships and can thus be used interchangeably. However, to maintain consistency with Neo4j terminology, this thesis employs the term 'labels' when referring to nodes and 'types' when referring to relationships.

Node labels and relationship types are used to index nodes and relationships, respectively, thus enabling more efficient querying. While a node can have any number of labels, including none, each relationship must be assigned with exactly one relationship type. Although relationships are defined as directed connections between start and end nodes, they can be traversed in both directions during queries in Neo4j.

Neo4j is a property graph (Vukotic et al., 2014), meaning each graph entity can be further described using properties. Numeric and character-based properties, such as integers, real numbers, and strings, are stored in nodes and relationships as a collection of key-value pairs. For example, a node representing a building can be labelled as *Building* and assigned with values such as its identifier and modification date. Figure 3.6 illustrates an example of how such a building and its bounding shape can be represented as nodes and relationships in Neo4j.

Although it is possible to store properties in relationships in Neo4j, in this thesis, most useful non-structural data imported from CityGML documents are stored exclusively in nodes. The majority of properties stored in relationships are auxiliary and reserved internally for further processing.

Neo4j is a *schema-less* database, meaning it does not impose any specific requirements on the contents and structure of nodes and their relationships. Instead, the data model is implicitly derived from the data stored within the database, rather than being explicitly defined as a component of the database itself (Vukotic et al., 2014). This allows for efficient and flexible handling of various CityGML objects, eliminating the need to account for all potential combinations that could arise from their data schema. However, this also means that Neo4j is better suited for storing specific *instances* of

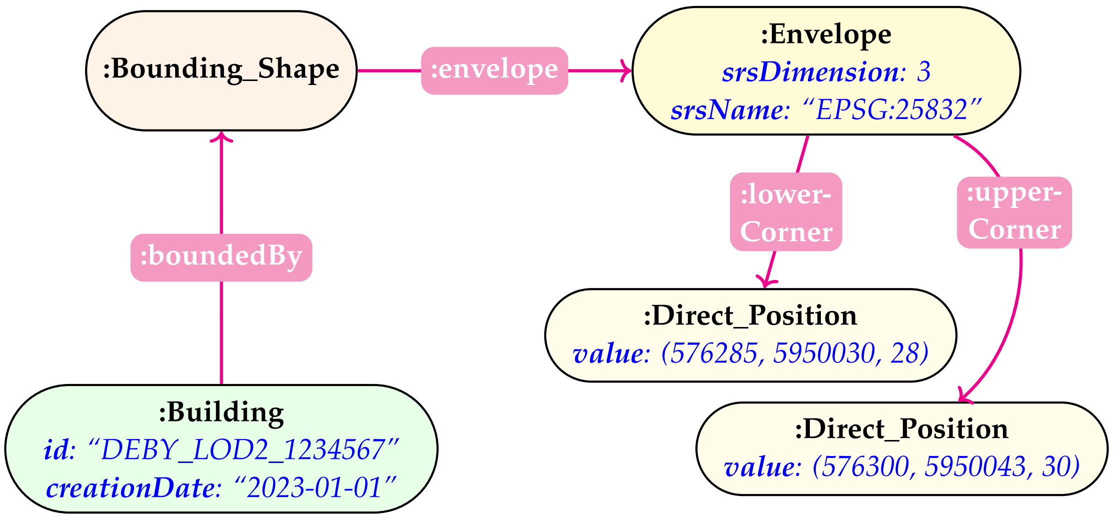


Figure 3.6.: An example of a building and its bounding shape represented as nodes and relationships in Neo4j. Node labels and relationship types are denoted by a preceding colon ':'. Node properties are shown in blue. This bounding shape will be automatically calculated and mapped onto graphs for each city object with a spatial extent, such as a building or a building part.

object classes, rather than the object classes themselves. The database lacks support for object-oriented concepts such as inheritance, polymorphism, and encapsulation. As a result, mapping CityGML objects onto graphs in Neo4j, despite their semantic and structural similarities, requires additional handling to ensure minimal information loss. These mapping strategies are explained in Section 3.3.

The concepts and methods proposed in this thesis have been tested and evaluated using Neo4j. However, they also applicable to other graph databases with similar data models and structures as explained in this section.

3.2. Graph Data Model for CityGML

Section 3.1.6 discussed several graph models used to represent CityGML documents. However, many of them were tailored to specific purposes, such as navigation or transformation. They often employed intermediate formats like JSON during conversion, leading to substantial information loss. This study proposes a graph data model compatible with both the CityGML data model and the graph database Neo4j, while minimizing information loss. This section addresses Research Question RQA3 (Graph Data Model for CityGML).

3.2.1. Requirements on Input CityGML Documents

While the methods presented in this chapter exhibit high robustness and generality, certain prerequisites regarding the input CityGML datasets must be met to ensure both the accuracy of the resulting graphs and the optimal performance of the mapping process. These prerequisites include:

1. **Adherence to the CityGML Information Model**: All input CityGML documents must strictly adhere to the underlying CityGML information model and its encoding schemas. This adherence allows for all elements to be correctly recognized and mapped onto their respective graph representations, minimizing any unnecessary deviations between the original CityGML documents and their graph-based counterparts.

2. **Exclusion of Circular Referencing**: This study only considers semantic interrelationships in CityGML that are defined with an explicit direction pointing from an object 'downwards' to its member *part-of* components. This establishes a consistent 'semantic order' throughout the dataset and, consequently, in its corresponding graph representation. Therefore, the use of XLinks in CityGML documents must also follow this coherent order, strictly allowing only parent objects to reference their respective components. Otherwise, circular referencing may occur, potentially causing the mapping process to fail to terminate or produce incorrect results. Therefore, any circular references must be excluded from the input CityGML datasets before mapping.

3.2.2. Modelling Nodes

Each CityGML element, based on its structural complexity, can be represented as a subgraph, a single graph node, or a node property. For example, a building, which contains many sub-elements, is represented as a subgraph. In contrast, simpler elements such as geometric points can be stored as individual nodes, with their coordinates stored as node properties. These different forms of representation are outlined as follows:

1. **Node Properties**: This applies to CityGML attributes or simple element with a single value that can be expressed in plain text.

2. **Single Node with Properties**: This applies to simple CityGML elements that possess more than one property or attribute value, which are stored as properties of the node.

3. **Subgraph**: This applies to complex CityGML objects with multiple sub-elements.

Additionally, labels can be assigned to nodes to group or distinguish them. These labels may represent the types or groupings of corresponding CityGML objects, indicate whether the elements belong to the old or new input dataset, or any other categories necessary for the process. To reflect the hierarchical structure of CityGML object types, two strategies for assigning node labels exist:

1. **Full Hierarchy**: This strategy stores the entire hierarchy of an object, spanning from the lowest subclass to the highest superclass, with each label representing a class. For instance, using this approach, a building node would require six labels: *Building*, *AbstractBuilding*, *AbstractSite*, *AbstractCityObject*, *AbstractFeature*, and *AbstractGML*. This hierarchy is shown previously in Figure 3.4. This approach allows for more compact but powerful semantic queries, as nodes can be queried based on any class within the class hierarchy. However, this requires a significant amount of additional disk space, as labelled nodes are indexed and a large number of classes exist in the class hierarchy of CityGML. The query time may thus be drastically slower in many databases, especially in Neo4j.

2. **Single Class**: In this strategy, only the lowest subclass (i.e., the defining class) of the corresponding CityGML object is assigned as a label to the representing node. For example, a building node would have exactly one label for its type: *Building*. The major advantage of this approach is the much smaller disk space required for indexing, as only a small number of classes are stored as labels. This accelerates query runtime significantly. However, queries that operate on more general or abstract classes require additional handling.

As (1) some superclasses, such as *AbstractGML*, may cover a large number of objects in CityGML, (2) indexing has a major impact on query runtime in Neo4j, and (3) type checking (i.e., determining whether a class is a subclass of another class) is well-supported in most object-oriented programming languages, the latter approach **Single Class** is employed in this thesis. Thus, the majority of nodes in the graph representations produced in this thesis contain two labels: one indicating its type and another specifying its originating CityGML document.

3.2.3. Modelling Relationships

In Neo4j, nodes are interconnected through relationships. Unlike nodes, each relationship must have exactly one relationship type. This type is used to represent the semantic connection between two nodes, which in turn represent two CityGML elements. The majority of these relationship types are derived from the CityGML data model itself. For example, the connection *boundedBy* between a building and its bounding shape, and the connection *boundedBySurface* between a building and its boundary surfaces, as shown in Figure 3.4.

The relationships are directed, starting from parent nodes and pointing towards their child nodes. This reflects the semantic order given in the CityGML data model. As a result, all contents of a building can be retrieved by traversing ‘downwards’ along the directed relationships starting from a node representing this building. Despite being directed, relationships in Neo4j can be traversed in both directions.

While traversing in one direction ‘downwards’, there are two types of multiplicities for relationships: **one-to-one** and **one-to-many**. These are the direct results of the one-to-one and one-to-many (aggregation) relationships in the CityGML data model. For instance, a building has exactly one bounding shape but multiple boundary roof, wall, and ground surfaces.

Many-to-many relationships can also be represented in relationships that can be traversed bidirectionally. For instance, a wall may belong to two adjacent rooms that share it (commonly achieved via the use of XLink), while each room can be bounded by multiple walls and other surfaces.

Like nodes, relationships in Neo4j can also be assigned with properties. However, most of the substantial content in the CityGML data model and its *citygml4j* objects, such as identifiers and thematic properties, is stored within CityGML elements rather than their connections. Thus, this is also reflected in the corresponding graph representations. If properties exist in a relationship, they are stored similarly to node properties. In this thesis, relationship properties primarily serve internal, auxiliary functions, such as those used during the interpretation process, as will be further explained in Chapter 5.

3.3. Methods for Mapping CityGML Objects onto Graphs

As mentioned in Section 3.1.1, the CityGML data model allows for the storage and exchange of various types of important information in city objects, including their semantic, geometric, and topological properties (Gröger et al., 2012). Addressing the first half of Research Question RQA4 (Mapping Methods and Evaluation), the mapping algorithms proposed in this thesis are designed to preserve these types of information[4] in the graph representations of CityGML documents. At the same time, they aim to maintain the structure of the resulting graphs as **aligned with the CityGML data model** as feasible. This approach allows users, both familiar and unfamiliar with CityGML,

[4]In CityGML, appearances can contain any surface-based theme. This includes not only visual or observable (geo-referenced) textures, but also other information sources such as infrared or solar radiation. Processing and matching such visual information involve research across multiple domains and are therefore beyond the scope of this research.

to navigate its graph representations effectively. Moreover, this approach eliminates information loss in all tested datasets, as will be shown in Section 3.5, thereby enabling converting graphs back to their respective CityGML objects.

The mapping methods used to construct graph representations of CityGML documents are shown in Algorithms 1 and 2. The first method *map(source)* initiates necessary parameters and invokes its auxiliary method *map(source, visited)*.

Algorithm 1: Graph mapping method *map(source)*

```
Input  : In-memory object-oriented representation source of a CityGML object
Output: Graph representation node of that CityGML object
1 visited ← new key-value map with objects as keys and mapped nodes as values
2 return map(source, visited)
```

The algorithms mentioned above employ several techniques and strategies, which are explained in the following sections.

3.3.1. Recursive Mapping

Most of the mapping process is executed within the second method *map(source, visited)* shown in Algorithm 2. This method is recursive (refer to Lines 13 and 26) and operates in a top-down manner. It starts with a given object, followed by its sub-elements, then the sub-elements of those sub-elements, and so forth. The process terminates when there are no more elements left for mapping. This is summarized as follows:

1. For each input object, an empty node is created in the graph, and the object's type is stored as the label of this node.
2. Each 'simple' attribute of the object is stored as a node property.
3. Each 'complex' attribute of the object is mapped as a node by recursively applying this method. The resulting nodes are then connected together to form a subgraph.

Section 3.3.6 explains when object attributes can be considered 'simple' or 'complex'.

3.3.2. Avoiding Circular References

Object-oriented representations of CityGML elements, such as those produced using *citygml4j*, may contain references that form cycles among objects. This may be caused by the use of special properties such as references to *parent* objects. These could lead to endless loops in the subsequent matching and interpretation process. Therefore, such circular references are excluded from mapping.

Algorithm 2: Recursive graph mapping method *map(source, visited)*

```
Input  : In-memory object-oriented representation source of a CityGML object
         Key-value map visited storing visited objects and corresponding nodes
Output: Graph representation node of that CityGML object
if visited.hasKey(source) then return visited.getValue(source)
node $\leftarrow$ a new empty graph node
visited.add(source, node)
class $\leftarrow$ source.getClass()
node.addLabel(class)
if class is array then
    node.addProperty("arrayMemberType", source.getArrayMemberType())
    node.addProperty("arraySize", source.getArraySize())
    for index $\leftarrow$ 0 to node.getArraySize() $-$ 1 by 1 do
        if isSimple(source[i]) then
            node.addProperty("arrayMember_" + index, source[i])
        else
            child $\leftarrow$ map(source[i], visited)
            rel $\leftarrow$ create a relationship ARRAY_MEMBER from node to child
            rel.addProperty("arrayMemberIndex", index)
        end
    end
else
    while class $\neq$ null do
        foreach non-static property prop declared in class do
            value $\leftarrow$ source.getProperty(prop)
            if isSimple(value) then
                node.addProperty(prop, value)
                node.addProperty(prop + "Type", value.getClass())
            else
                child $\leftarrow$ map(value, visited)
                create a relationship with type prop from node to child
            end
        end
        class $\leftarrow$ class.getSuperClass()
    end
end
return node
```

In addition, the use of XLinks can also lead to circular interlinking, particularly when an object references its ancestors. To avoid such cases, this study only considers CityGML documents that employ XLinks in a strictly 'downwards' semantic direction, as mentioned in Section 3.2.1. Such cycles can be identified during mapping by tracking visited elements, similar to the strategy described in Section 3.3.3.

3.3.3. Preventing Repeated Mapping

The CityGML data model is defined with a complex web of interconnections between its elements. As a result, the in-memory representations of these elements contain a large number of pointers or references to other objects. Despite the exclusion of directed cycles, these objects are prone to repeated mapping without an effective repetition prevention mechanism. This is illustrated in the following example of a building with several boundary roof, wall, and ground surfaces. The 3D shape of the building is represented by a solid geometry that includes these boundary surfaces as its sides. This solid reuses the already defined boundary surfaces through the use of XLinks. However, due to the recursive, top-down nature of the proposed mapping methods, which process an object along with all its sub-elements, each boundary surface of the building would be mapped onto graphs twice, even though the object's structure does not contain any directed cycles.

To prevent duplicates of the same objects in the resulting graphs and to ensure the process terminates correctly, a key-value store, denoted as *visited* in Algorithm 2, is employed. This store is essentially a map over tuples of type *(Object, Node)*, where the objects serve as keys and their corresponding graph representations serve as values. The keys are unique, as each object is identified by their allocated address in main memory. As previously mentioned, a graph representation of an object may be a single node or a subgraph containing many nodes and relationships, depending on the complexity of the given object. In the former case, the single generated node is stored as the value, along with its corresponding object, in a key-value pair. In the latter case, the source node of the generated subgraph is selected as the value of the key-value pair. A (universal) source node of a directed graph can be considered as a starting point, from which all other nodes within the graph can be reached.

During mapping of each object, the process first determines whether the object has been previously visited and mapped by searching for a corresponding key in the key-value map (refer to Line 1). The mapping continues only if the key does not exist, indicating that the object has not been visited before. Otherwise, the previously mapped node or subgraph assigned to this object is returned instead. This ensures that no object is mapped more than once, and references between objects can be accurately represented.

3.3.4. Mapping Arrays

In CityGML, many elements are given in a one-to-many or many-to-many relationships. These relationships are stored using various data structures for collections, such as (dynamic) arrays, linked lists, and key-value maps. In Java, arrays are particularly common, as they are also used internally by many other data structures. For instance, array lists or some key-value maps store their data in a built-in array. Therefore, the ability to map arrays of objects onto graphs is essential (refer to Line 6), as it also allows for mapping more complex data structures that utilize arrays.

An array with simple elements is represented as a collection of key-value pairs, following Neo4j's graph data model discussed in Section 3.1.7. For complex member types, the array is mapped as a collection of nodes, with each node representing a member object, and the structure is recursively mapped until all contents are captured.

The auxiliary node properties *arrayMemberType*, *arraySize*, *arrayMember_i*, as well as the auxiliary relationship property *arrayMemberIndex* are needed for reconstructing graph representations back to their original corresponding arrays. This reconstruction process will be explained in Section 3.6.

3.3.5. Extracting Sub-elements

All contents available within an object must be mapped onto graphs. These include properties declared in the object's defining class, as well as those inherited from the object's superclasses within the class hierarchy. For instance, as shown in Figure 3.4, a building object contains properties declared in its defining class *Building*, along with those inherited from its superclasses: *AbstractBuilding*, *AbstractSite*, *AbstractCityObject*, *AbstractFeature*, and *AbstractGML*.

However, due to data encapsulation in object-oriented modelling, an object can only inherit non-private properties declared in its superclasses. In such cases, an alternative is to utilize the read-only *getter* methods to extract the same information. However, this approach depends on the existence and visibility of such methods. Typically, *getter* methods are defined with non-*private* access modifiers.

Algorithm 2 performs a bottom-up search for all sub-elements of a given object (refer to Line 20). It first extracts all properties declared in the object's defining class, which is the lowest within the class hierarchy. These properties must be non-static, since static properties are bound to their respective classes and exist only once for all instances. The process then moves 'upwards' to the next superclass in the hierarchy, continuing extracting properties until it reaches the highest superclass (refer to Lines 19 and 30).

In statically typed languages such as Java, extracting properties from objects can be achieved using reflective programming, also known as **reflection** (McCluskey, 1998).

In object-oriented modelling, two types of inheritance exist: single and multiple inheritance. Single inheritance restricts a subclass to inherit from at most one superclass, while multiple inheritance enables a subclass to inherit from more than one superclass. Due to the substantial increase in complexity and ambiguity of multiple inheritance, this research exclusively applies single inheritance to all models. This approach aligns with the CityGML data model and is enforced by the majority of object-oriented programming languages, including Java. The use of single inheritance ensures that all superclasses of a given object can be sequentially iterated along a single path.

3.3.6. Evaluating Sub-elements' Complexity

Adhering to the graph data model presented in Section 3.2, Algorithm 2 initiates its recursive calls based on the complexity of the input objects and their properties. If an object property is 'simple', it suffices to store it as a node property. However, for sub-elements that are sufficiently complex, the method will be recursively invoked again (refer to Lines 10 and 22). Whether an object is considered 'simple' in this context, is determined by the method *isSimple(source)*. A flowchart illustrating this method can be found in Figure 3.7.

This method evaluates the complexity of objects through three consecutive checks:

1. It first verifies whether the source object is of a primitive type,
2. If not, it determines whether the object contains exactly one value of a primitive type, and
3. If none of the above applies, it tests whether the source object can be efficiently stored and compared using a single string representation.

The first check is applied to primitive data types, such as booleans (*true*, *false*), natural and floating numbers, and texts. These types account for the majority of the actual information payload in the CityGML datasets. The second check prevents the creation of redundant nodes for an object that has only one primitive value. In this case, a single node property is created instead of two additional nodes. The third check is designed to handle more complex objects that can still be stored and compared efficiently using their corresponding text representation.

For instance, the modification date and number of storeys of a building are of primitive type and thus considered simple. As a result, they are stored as properties within the building node. On the other hand, measurements are considered complex and are stored as nodes. This is because each measurement contains a value and a unit. To facilitate the comparison of measurements during the matching process, as will be discussed in Chapter 4, the value and unit of a measurement are stored separately

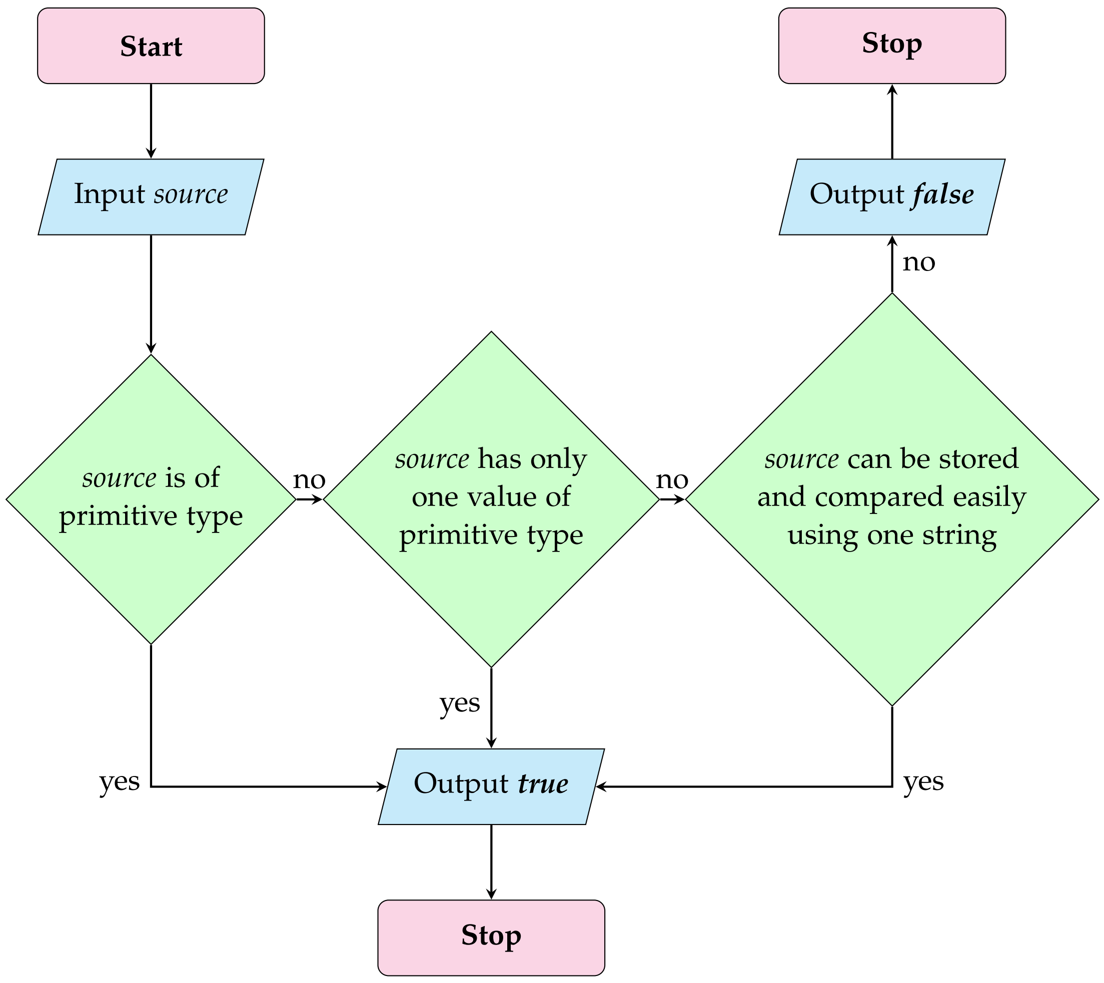

Start
Input *source*
source is of primitive type
no
source has only one value of primitive type
no
source can be stored and compared easily using one string
no
Output ***false***
Stop
yes
yes
yes
Output ***true***
Stop

Figure 3.7.: A flowchart visualization of the method *isSimple(source)*. The method returns *true* if the *source* object can be represented as a single node property. Otherwise, additional nodes or subgraphs are created to represent this *source* object. The primary objective of this method is to provide a unified approach for representing CityGML objects of arbitrary complexities, thereby increasing the runtime efficiency of the matching process introduced later in Chapter 4.

as properties within a node. This allows for efficient comparison of measured values with unit conversion and measurement errors taken into account, such as in the case of 1.001 m and 100.101 cm, which are considered equal for an error tolerance of 1 mm.

3.4. Resolving XLinks

As an XML-based encoding, CityGML utilizes XLinks (W3C, 2006) to define new connections among city objects. This not only allows for reusing existing objects, but also enables the definition of more complex semantic, geometric, and topological relationships between CityGML objects. Moreover, as will be explained in Section 6.1, the XLink concept is also utilized to divide a large CityGML document into smaller, more manageable pieces.

However, by solely applying the mapping methods outlined in Algorithms 1 and 2 to CityGML documents containing XLinks, the resulting graph representations become fragmented. For instance, the graph representation of a building's solid geometry becomes disconnected from those representing its boundary surfaces. This is because the solid is defined as a collection of XLink references to these boundary surfaces, but XLink elements do not contain any further sub-elements other than a text field. This results in the termination of the mapping of the current elements at their XLink elements, leaving the graphs disjointed.

Given the significance of XLink connections as a source of semantic information, they must be present as explicit relationships in the underlying graph representation of CityGML documents. The method for resolving XLinks is outlined in Algorithm 3, addressing Research Question RQA5 (XLink Resolution).

Algorithm 3: Method for resolving XLinks *resolveXLinks(graph)*

```
Input  : Graph representation graph of a CityGML model containing XLinks
Output: Graph representation graph updated with resolved connections
1 hrefNodes ← all nodes of graph that have property href
2 idNodes ← all nodes of graph that have property id
3 foreach node hrefNode in hrefNodes do
4     id ← hrefNode.getProperty("href").remove("#")
5     idNode ← a node in idNodes uniquely identified by id
6     rel ← create a relationship object from hrefNode to idNode
7     hrefNode.removeProperty("href")
8     rel.addProperty("wasXLink", true)
9 end
```

The process for resolving XLinks begins by searching for all nodes in the given graph that contain the property *href*. This property, as prescribed in XLink, is used to reference other objects via their identifiers. An example of an *href* is '#ReferencedID', where the prefix '#' is employed in XLink to distinguishes *href* properties from identifiers. For each node with the property *href*, its value is extracted. The referenced identifier is obtained by removing the prefix '#' from this value. The method then searches the graph for a node that uniquely matches this referenced identifier. Finally, a relationship is established in the direction from the *href* node to this referenced node, serving as an **explicit representation** of the XLink connection. An illustration of this process is shown in Figure 3.8. Several strategies and methods employed in this process are explained in the following sections.

3.4.1. Separation of Graphs for each City Model

In this thesis, the graph representations of both the old and new CityGML dataset are stored in the same graph database. As a result, all nodes and relationships of both graphs are organized under the same sets of database indexes. This means that (1) for each *href* value, there may exist two nodes with the same matching identifier originating from each dataset, and (2) vice versa, when searching for *href* nodes, those of both resolved and unresolved XLinks may be returned. Without a distinction between nodes originating from each dataset, unrelated nodes across the two datasets may be connected with each other, leading to incorrect representations of XLinks.

Therefore, to address this issue, the following strategies are employed:

1. The graph representations of both the old and new dataset must be distinct, sharing no common nodes. Moreover, a mechanism to differentiate nodes based on their originating dataset is required. This can be achieved by utilizing an additional label indicating the originating dataset for all nodes during mapping.

2. The method for resolving XLinks outlined in Algorithm 3 must be invoked twice, one for each CityGML dataset after it has been mapped onto graphs.

3.4.2. Strategies on Finding Nodes for Interlinking

Database indexes can be employed to efficiently locate all nodes containing an *href* property or an identifier (refer to Lines 1 and 2), as well as to search for nodes with a specific identifier (refer to Line 5). In Neo4j, indexing can be applied to nodes with specific labels and property names. While property names are simply either 'href' or 'id', a large number of labels exist, as different types of nodes can be assigned with these properties. There are two strategies to index and obtain these labels:

1. **Hierarchical Approach**: As explained previously in Section 3.2.2, the class name of each object is assigned as label of their corresponding node representation. This label can be extracted and evaluated within its class hierarchy. According to the data models of GML and CityGML, the use of the properties *href* and *id* is defined in specific superclasses. This behaviour is inherited by various subclasses,

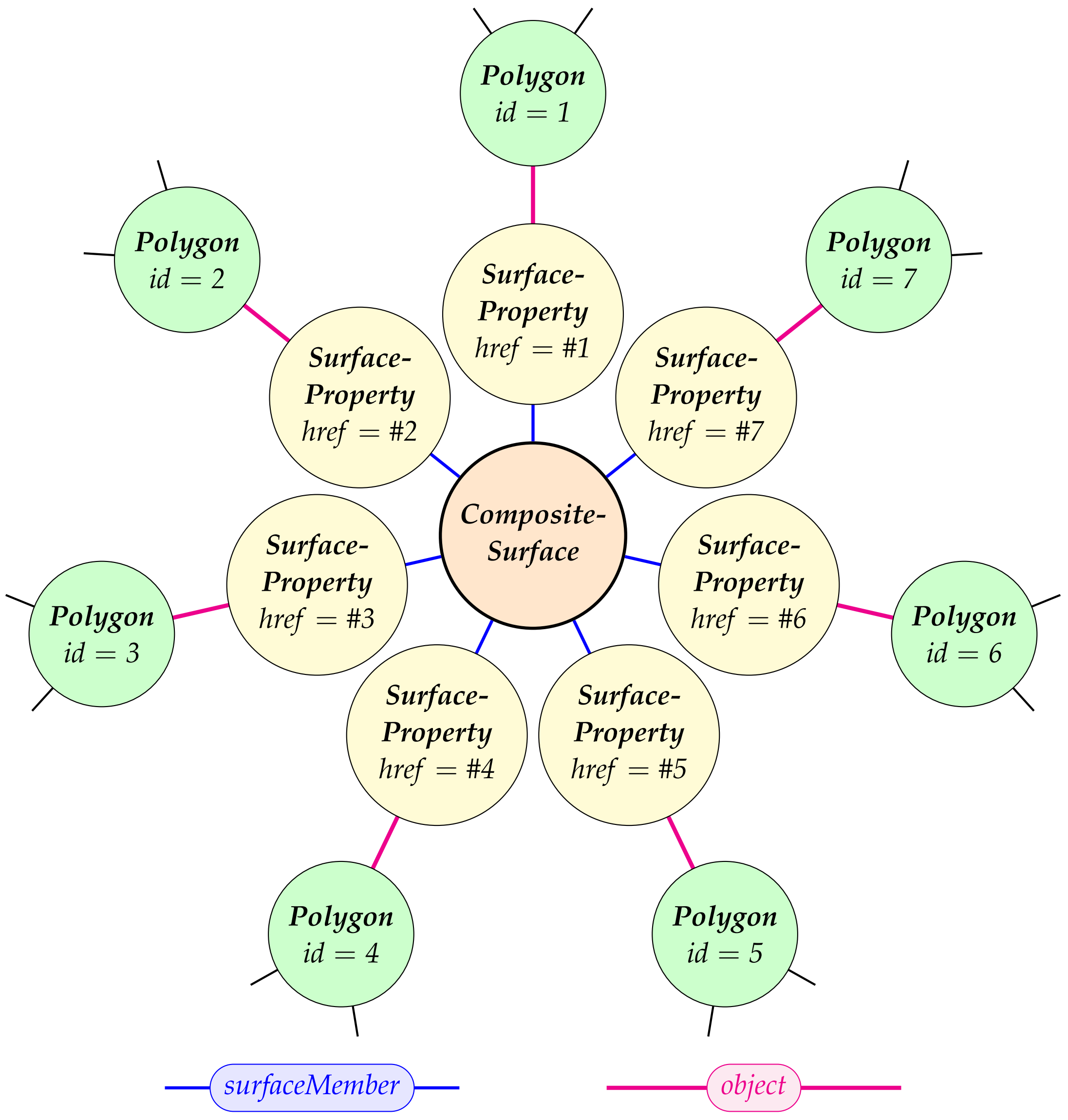

Polygon id = 1
Polygon id = 2
Polygon id = 3
Polygon id = 4
Polygon id = 5
Polygon id = 6
Polygon id = 7
Surface-Property href = #1
Surface-Property href = #2
Surface-Property href = #3
Surface-Property href = #4
Surface-Property href = #5
Surface-Property href = #6
Surface-Property href = #7
Composite-Surface
surfaceMember
object

Figure 3.8.: An illustration of the resolution of XLinks in the graph representation of a solid's *CompositeSurface* in CityGML. For every matched pair of *href* and *id* node (yellow and green, respectively), an explicit relationship (red) is created. Without these connections, the resulting graph would become disjointed. The node *CompositeSurface* in the centre (orange) represents the exterior of a building's solid. The corresponding CityGML model of this building is given in Listing 3.1. For visual clarity, relationships are shown without direction. A separate visualization of the graph representation of this building is shown in Figure 3.4 (without *SurfaceProperty* nodes).

whose instances are mapped and present in the graphs. For example, in *citygml4j*, the definition of the property *id* of all CityGML objects can be traced back to one single superclass *AbstractGML* (see the content of this class in the example of a building object shown in Figure 3.4). In the case of *href*, the superclass *Association-ByRepOrRef* is the defining class. Thus, in this case, nodes can be indexed and subsequently obtained, if the class represented by their labels are a subclass of the superclasses mentioned above.

2. **Cumulative Approach**: During mapping, when the properties *href* or *id* are encountered, the labels of their corresponding container nodes are stored in a list. After the mapping process is complete, these stored labels are extracted and used for defining indexes.

The hierarchical approach has the advantage of recognizing the superclasses and their subclasses in advance, allowing for the definition of indexes prior to mapping. During the mapping process, when the input object is an instance of the specified superclasses, its corresponding node is indexed. However, this approach requires strict adherence of the node labels to the hierarchical structure of GML, CityGML, and *citygml4j*, which must be known in advance.

On the other hand, the cumulative approach is not limited by this constraint. It dynamically collects labels of nodes that have the properties *href* or *id* while mapping, making it more adaptable and applicable to any data model. Moreover, no type-checking of classes is required in this case. However, this means that index definitions can only be performed after the core mapping process is complete, once all node labels for indexing are known.

Since the graph database Neo4j allows index definitions both before and after the node creation, the more flexible, **cumulative approach** is employed in this thesis.

3.4.3. Connecting *href* and Referenced Nodes

For each pair consisting of an *href* node and its corresponding referenced node, an explicit relationship is established (refer to Line 6), pointing from the *href* node to the referenced node. The type of this relationship is determined based on the semantic context of the *href* node and its link to the referenced node in the underlying encoding. In *citygml4j*, all objects containing the property *href* are instances of the class *Association-ByRepOrRef*. Therefore, all relationships representing XLink connections are created with type *object*, as dictated by the class *AssociationByRep*, which is a superclass of *AssociationByRepOrRef*. As a result, once these relationships are in place, there will be **no structural distinction** between a graph representation of an object defined using an XLink and one defined without it.

Although not mandatory, the following supplementary steps can be performed after the XLink connections have been resolved:

1. **Removal of the Node Property *href***: The property *href* becomes redundant and can be removed (refer to Line 7). This ensures that the subsequent matching process does not register any unnecessary changes on this type of properties, since the matching process does not differentiate between objects defined with or without an XLink. This removal of the property *href* also enables the distinction between resolved and unresolved XLinks in the graphs.
2. **Addition of the Relationship Property *wasXLink***: To indicate that an XLink connection has been replaced by an explicit relationship, an additional property *wasXLink* with value *true* can be added to the relationship (refer to Line 8). This is particularly useful for tasks such as reconstructing graph representations back into their original CityGML objects, as will be described in Section 3.6.

The method outlined in Algorithm 3 operates under the assumption that the input CityGML datasets are valid and adhere to the encoding standards of XML, GML, and CityGML. For instance, while there may be multiple *href* values pointing to the same identifier, each referenced identifier must be found in exactly one node per dataset.

3.5. Evaluating Information Preservation in Generated Graphs

The graphs created using the methods proposed in this chapter serve as a basis for subsequent processes, including but not limited to the change detection and interpretation process in Chapters 4 and 5. Therefore, it is crucial to ensure that the generated graphs can (1) accurately capture all substantial thematic details available in the input datasets, and (2) closely mirror the semantic structure of the original CityGML documents, thereby minimizing or preventing any discernible loss of information. This section addresses the last half of Research Question RQA4 (Mapping Methods and Evaluation).

3.5.1. Assessing Mapped Thematic Content

To evaluate how accurately a graph representation of a CityGML document can capture its thematic content, several indicators are involved:

1. **Coverage**: Each CityGML element has a corresponding representation in the graphs. This representation can be a property of a node or a relationship, an individual node or relationship, or even a subgraph consisting of multiple nodes and relationships. The node labels and relationship types must align with the types of their original CityGML elements. The coverage is determined by examining:

   a) **Type Coverage**: Whether all types of CityGML objects have been mapped at least once to corresponding graph entities, and
   b) **Instance Coverage**: For each type of CityGML objects, whether all instances have been mapped to corresponding graph entities.

   A graph representation is considered to have a high type and instance coverage if it exhibits both a high number of mapped CityGML types and occurrences per type in relation to the total number of CityGML types (determined by the CityGML data model) and occurrences per type (determined by the datasets), respectively. The aim is to ensure that the generated graphs achieve 100 % type and instance coverage.

2. **Data Replication**: Most substantial thematic data of CityGML objects is given as attributes or text contents, such as the properties of buildings, and names and values of generic attributes. It is crucial not only for the generated graphs to include these sources of information but also to accurately preserve their data. Simple plain texts, such as the function or description of a building, must be stored in the corresponding graphs exactly as presented. More complex data, such as point coordinates, can be represented differently from those in the original datasets, but the content itself must remain the same. Therefore, the objective is to attain a 100 % data replication rate, indicating that all thematic data available in the original CityGML datasets is accurately replicated in the generated graph representations.

3.5.2. Assessing the Semantic Structure of Mapped Graphs

The semantic structure of the generated graph representations of CityGML documents is evaluated based on the following indicators:

1. **Relationship Coverage**: For any two elements in the original CityGML documents that are in a parent-child relationship, there must also be a corresponding path between the node representations of these CityGML elements in the generated graph. The length of this path is not strictly limited to one, as additional (auxiliary) graph nodes may be created in between. However, this length must remain consistent in the case of multiple children of the same type under the same parent. The graph must also reflect the number of children per type for each CityGML element in the original CityGML documents by ensuring the same number of paths between the parent node and its ascendant nodes of that type. Like the order of the children under the same parent element in CityGML, the order of their graph representations do not play any role. The objective is to maximize the

coverage of these graph paths representing the parent-child relationships in the original datasets to 100 %.

2. **XLink Replacement**: The XLink connections in the CityGML documents must be represented as explicit relationships between existing nodes in the generated graphs. This replacement should apply consistently across all elements, eliminating structural deviations between CityGML elements defined with and without the use of XLinks. The aim is to ensure that the generated graphs can achieve a replacement rate of 100 % for all valid XLinks. In addition, the evaluation must ensure that no directed cycles occur during the replacement of XLinks, as required in Section 3.2.1.

3.5.3. Evaluation Results of all 14 CityGML Modules and 5 LODs

A graph is considered an accurate representation of a CityGML document if it can entirely capture both the source's thematic and structural content. Using the strategies described above, this section evaluates the preservation of thematic and structural information within the generated graph representations of CityGML documents.

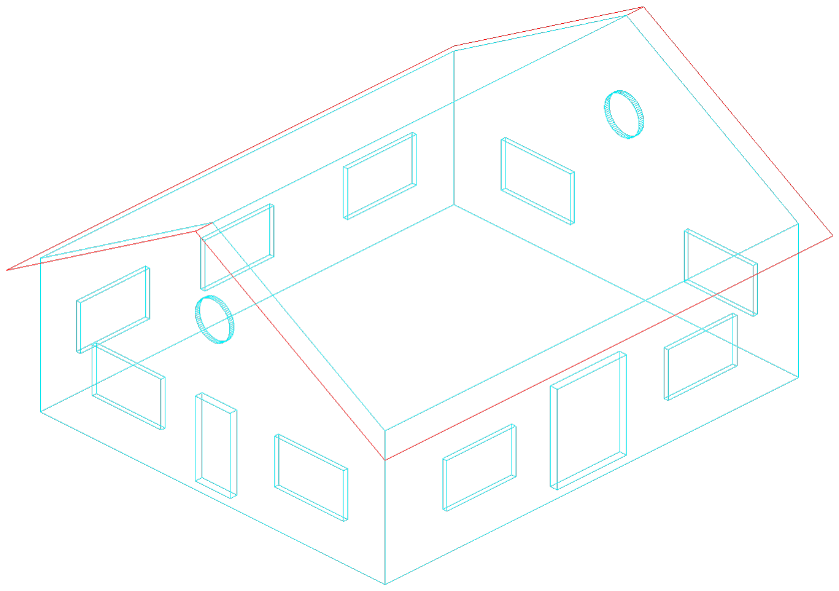
The evaluation applies the proposed methods outlined in Algorithms 1 to 3 to two types of datasets: the *FZK-Haus* datasets (KIT IAI, 2017) in all five LODs from 0 to 4, and the *Railway-Scene* dataset (Häfele & Nagel, 2015) in LOD3 that employs all fourteen thematic modules of CityGML. These six datasets ensure that the proposed methods can reliably handle a wide variety of CityGML elements.

A visualization of the *FZK-Haus* dataset in LOD2-4 can be found in Figures 3.3, 3.9a, and 3.9b, respectively. Figure 3.10 shows a combined 3D rendering of various CityGML objects from the *Railway-Scene* dataset.

A visualization in Neo4j Browser of the graph representation of the *FZK-Haus* dataset in LOD2 and its single building are shown in Figures 3.11 and 3.12, respectively. The results of the XLink resolution process, as outlined in Algorithm 3, is illustrated in Figure 3.13.

Table 3.2 provides a comprehensive overview of the evaluation results regarding the preservation of both semantic and structural information in the generated graph representations of all six CityGML documents. This table is an excerpt, showing only the first and last two pages of the full version in Table A.1.

To evaluate the preservation of thematic content, all element and attribute names from the original CityGML documents are collected, as shown in the table rows. These names incorporate prefixes, such as *bldg* in *bldg:Building* or *xlink* in *xlink:href*, corresponding to the namespace prefixes used in CityGML encoding, as well as other XML-based standards.


(a) FZK-Haus in LOD3 (interior view)

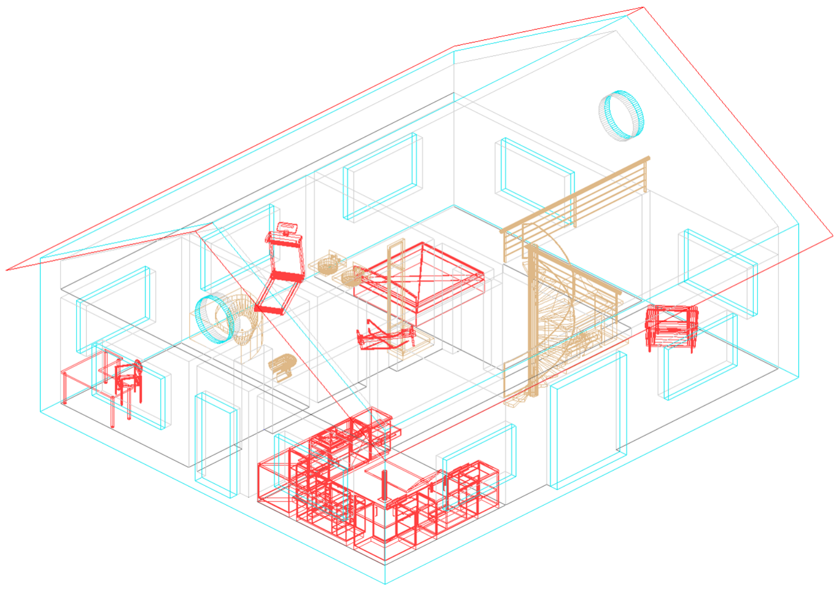

(b) FZK-Haus in LOD4 (interior view)

Figure 3.9.: Interior visualization of the *FZK-Haus* datasets (KIT IAI, 2017) in LOD3 and LOD4 using the KITModelViewer (KIT IAI, 2024).

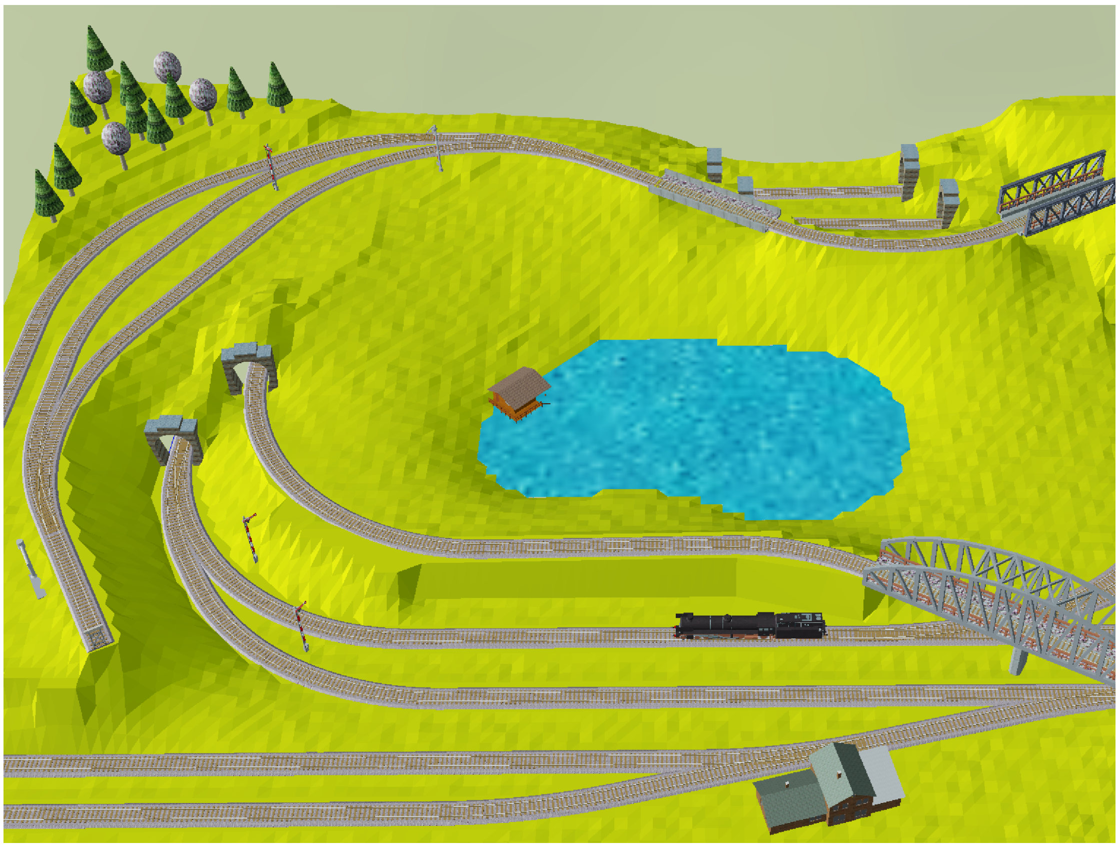


Figure 3.10.: Visualization of a segment from the *Railway-Scene* dataset (Häfele & Nagel, 2015) using the 3DCityDB Web Map Client (Yao et al., 2018). The dataset, presented in LOD3, incorporates various city objects from all fourteen thematic modules in CityGML (Gröger et al., 2012): *Core*, *Appearance*, *Building*, *Bridge*, *Relief*, *CityFurniture*, *Generics*, *CityObjectGroup*, *LandUse*, *TexturedSurface* (deprecated), *Transportation*, *Tunnel*, *Vegetation*, and *Water-Body*.

The fourteen thematic CityGML modules and their default prefixes are: *Core* (*core*), *Appearance* (*app*), *Building* (*bldg*), *Bridge* (*brid*), *Relief* (*dem*), *CityFurniture* (*frn*), *Generics* (*gen*), *CityObjectGroup* (*grp*), *LandUse* (*luse*), *TexturedSurface* (*tex*) (deprecated), *Trans-portation* (*tran*), *Tunnel* (*tun*), *Vegetation* (*veg*), and *WaterBody* (*wtr*) (Gröger et al., 2012). The XML-based standards and their default prefixes are: GML (*gml*) (Cox et al., 2004), Extensible Address Language (xAL) (*xAL*) (CIQ TC, 2002), and XLink (*xlink*) (W3C, 2006).

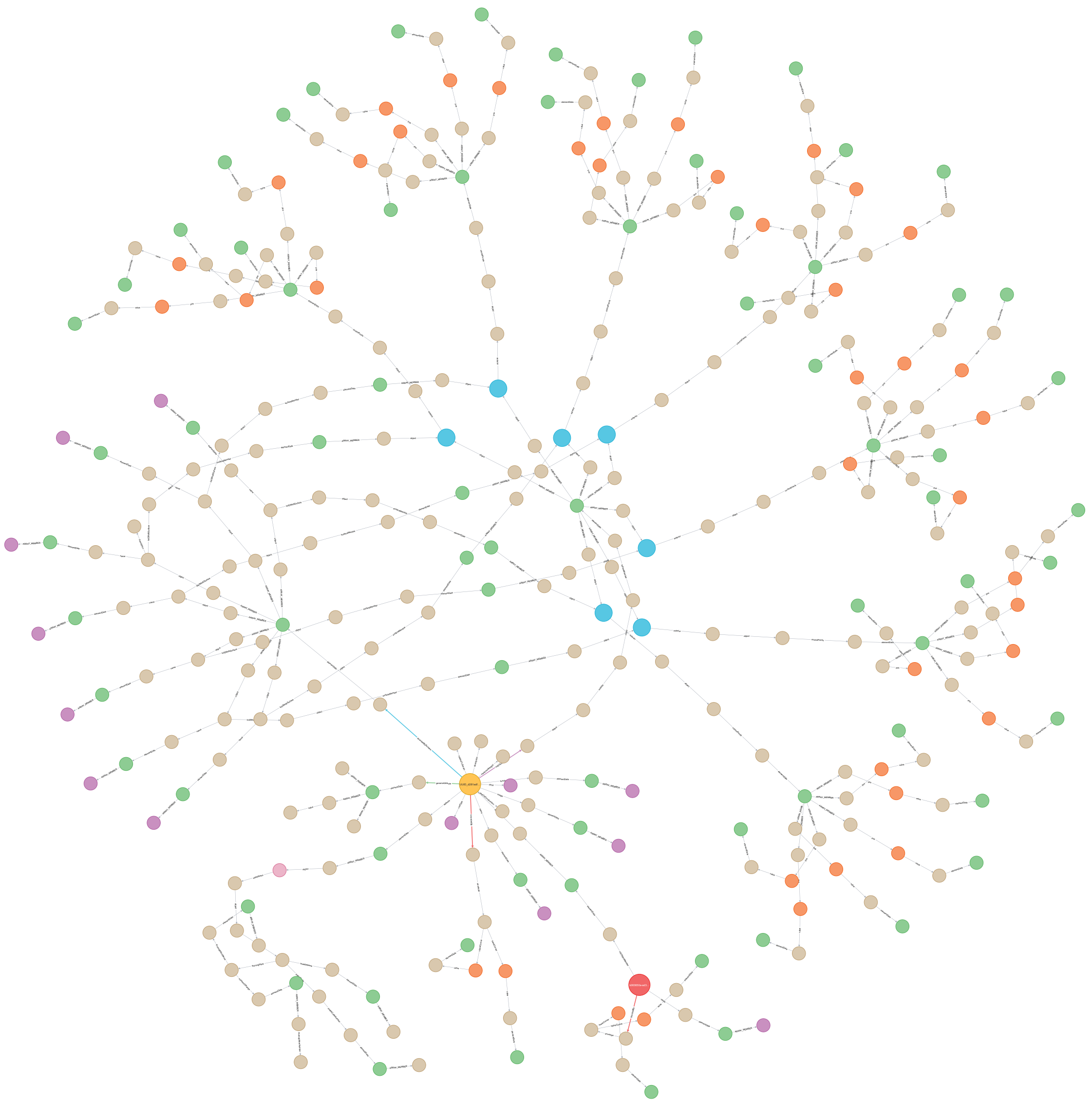


Figure 3.11.: Visualization in Neo4j Browser of the graph representation of the *FZK-Haus* dataset (KIT IAI, 2017) in LOD2. The big red node at the bottom represents the city model element, while the big orange node represents the building. Arrays, such as collections of objects (when shown as inner nodes) or simple point coordinates (when shown as sink nodes), are depicted in green. Polygons are shown in blue, points in orange, and code values (such as for the CityGML attributes *function* and *usage*) in purple.

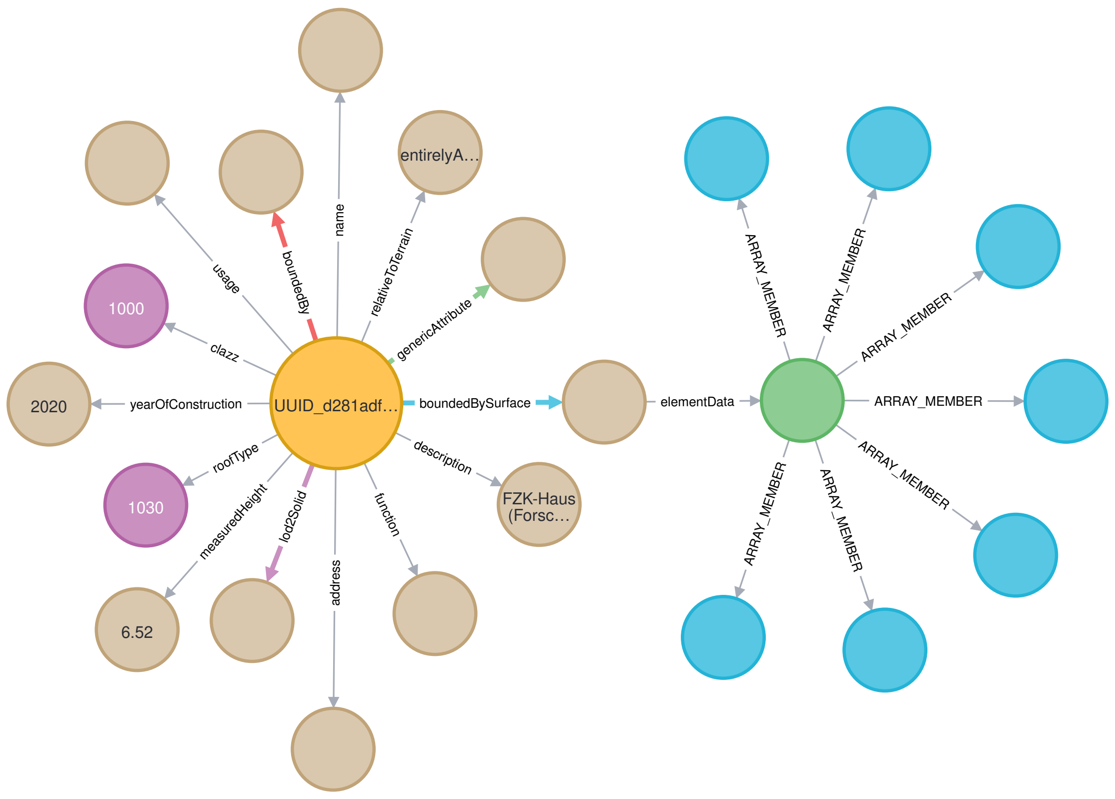
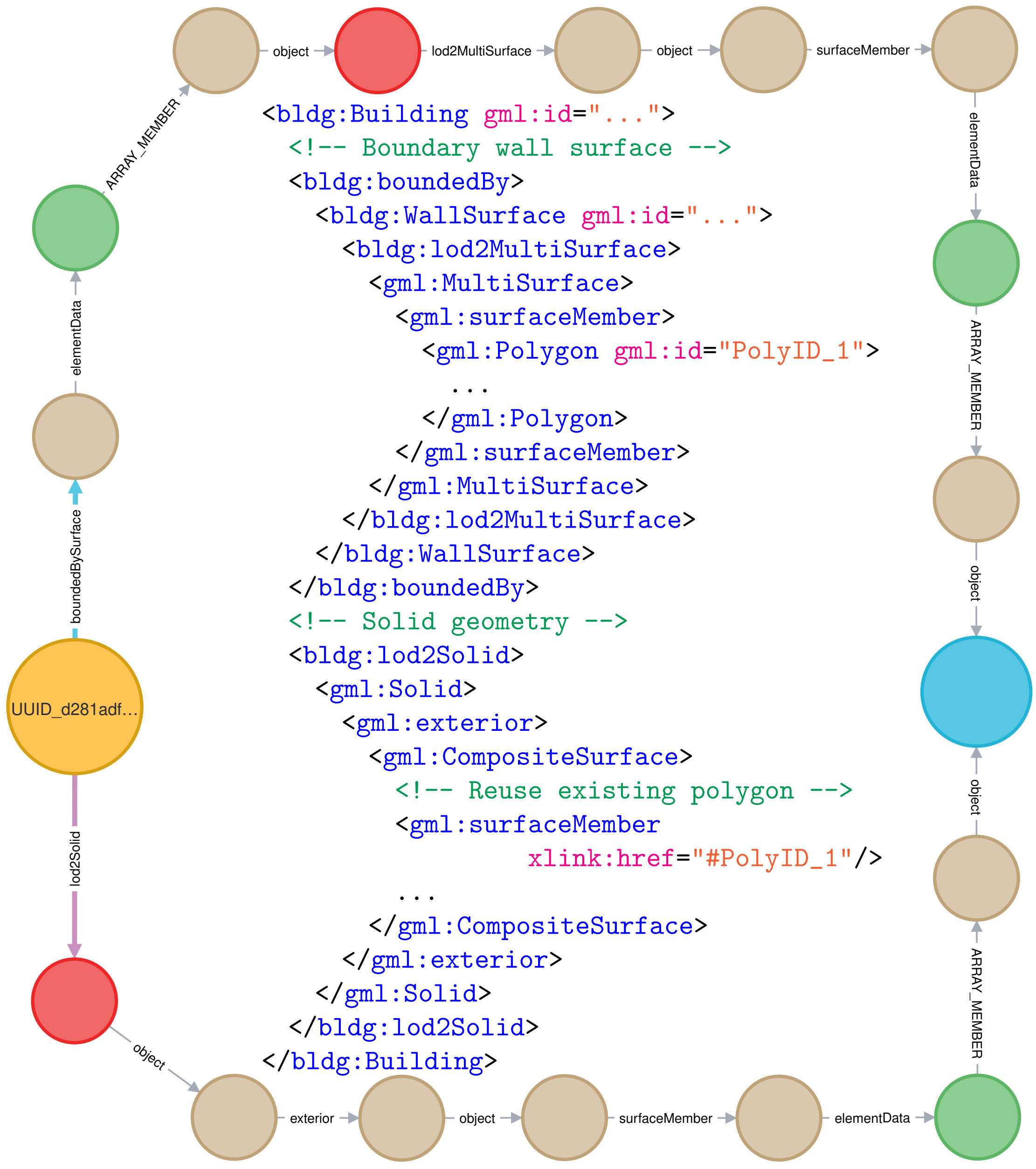


Figure 3.12.: Visualization in Neo4j Browser of the graph representation of the building (big left node) from the *FZK-Haus* dataset (KIT IAI, 2017) in LOD2. For visual clarity, only paths of lengths between one and three from the building node are shown. This building has seven boundary surfaces, as indicated by seven blue nodes on the right. Their array node is depicted in green.

To evaluate the preservation of structural content, each element and attribute is assigned a level (or depth) indicating its position in the document hierarchy. The level starts from 0 for the root element and increases by 1 for each nested element. For instance, the element *core:CityModel* is at level 0, as it is the root element of every CityGML element. The element *bldg:Building* is at level 2, as it is a child of the element *core:cityObjectMember*, which is a child of the root element. Thus, these levels provide the information on the semantic order and reachability properties of these elements and their corresponding graphs. The maximum levels of the *FZK-Haus* datasets in LOD0-4 are 9, 11, 11, 13, and 15, respectively. The maximum level of the *Railway-Scene* dataset is 13. Table 3.2 displays the elements sorted in ascending order of their levels.



```xml
<bldg:Building gml:id="...">
  <!-- Boundary wall surface -->
  <bldg:boundedBy>
    <bldg:WallSurface gml:id="...">
      <bldg:lod2MultiSurface>
        <gml:MultiSurface>
          <gml:surfaceMember>
            <gml:Polygon gml:id="PolyID_1">
              ...
            </gml:Polygon>
          </gml:surfaceMember>
        </gml:MultiSurface>
      </bldg:lod2MultiSurface>
    </bldg:WallSurface>
  </bldg:boundedBy>
  <!-- Solid geometry -->
  <bldg:lod2Solid>
    <gml:Solid>
      <gml:exterior>
        <gml:CompositeSurface>
          <!-- Reuse existing polygon -->
          <gml:surfaceMember
                    xlink:href="#PolyID_1"/>
          ...
        </gml:CompositeSurface>
      </gml:exterior>
    </gml:Solid>
  </bldg:lod2Solid>
</bldg:Building>
```

Figure 3.13.: Visualization in Neo4j Browser of the graph representation of a polygon (right blue) from the *FZK-Haus* dataset (KIT IAI, 2017) in LOD2 after all XLinks have been resolved, as described in Algorithm 3. An excerpt of this CityGML document is shown in the middle. This polygon is defined both as a boundary wall surface (top red) and an exterior surface of a solid geometry (bottom red) of the building (big left).

To evaluate whether the thematic attributes of the original CityGML datasets can be accurately replicated in the generated graphs, all these attributes and node properties in the graphs are first divided into groups. This grouping is based on their levels in the CityGML documents and depths in the graphs, which are their distances from the city model node. Then, a comparison between these text-based values with the same name is performed within these groups.

Table 3.2 shows that some elements may appear at different levels, as allowed by the underlying encoding standards. For example, the element *gml:Polygon* is found at levels 10 and 12 in the same *FZK-Haus* LOD4 dataset, since GML polygons can be used to define many complex geometric objects in CityGML. These levels directly influence the placement of the corresponding subgraphs within the main graphs representing each CityGML document.

The differentiation in levels of otherwise identical elements enables the structural evaluation of the generated graph representations when compared to the original CityGML datasets. However, the level of an element does not affect the structure of its own graph representation. The mapping methods will generate the same graph structure for the same city object type, regardless of the levels of their instances.

The total number of occurrences of each CityGML element and attribute per dataset and level is shown on the right-hand side of Table 3.2. The table cells are blue if (1) all the occurrences have been mapped to graphs, and (2) the order of their levels is preserved in the graphs. For instance, the *FZK-Haus* datasets have one building each, while the *Railway-Scene* dataset has three buildings. All of these table cells are blue, which means that there exists a node representation for each building, and all these nodes are the descendants of the node representing *core:CityModel*.

While some CityGML elements are mapped onto nodes or subgraphs, others are converted into relationships. This is determined by the CityGML encoding schemas, the binding of CityGML classes in *citygml4j*, and the mapping methods proposed in this study.

Table 3.2 (middle column) describes the mapping patterns for each CityGML element. These descriptions use the following symbols:

1. Node : Create either a single node or a source node of a subgraph
2. Rel : Create an outgoing relationship from the current node
3. Prop : Insert a property to the current node
4. *L* : Use the name from the left column in 'CamelCase' without namespace prefix
5. *l* : Use the name from the left column in 'camelCase' without namespace prefix

To distinguish between elements and attributes that share the same name but belong to different namespaces, the (shortened) package names of their corresponding Java classes in *citygml4j* can be used. For instance, the elements *bldg:GroundSurface* and *tun:GroundSurface* have identical names without the namespace prefixes. Thus, their corresponding nodes in the graph can be labelled as *building.GroundSurface* and *tunnel.GroundSurface*, where *building* and *tunnel* are the (shortened) names of their packages. However, these package names are omitted in the following descriptions for simplicity, if the element and attribute names are unique and not shared by any other elements or attributes presented in the example. For instance:

1. ***core:CityModel*** Node *L*
   Create a node with the label *CityModel*.

2. ***core:cityObjectMember*** Rel *l*
   Create a relationship with the type *cityObjectMember*.

3. ***core:creationDate*** Prop *l*
   Insert a property with the name *creationDate* to the current node.

4. ***bldg:measuredHeight*** Prop *l* + Node ***Length***
   Create a relationship with the type *measuredHeight* and a node with the label *Length*. The new relationship connects the current node with the new node.

5. ***xlink:href*** Rel ***object*** $\longrightarrow$ Node **found by *href***
   Create a relationship with the type *object* and point it to an existing node that has the same identifier as the value of *href* (without '#').

Such descriptions in Table 3.2 illustrate how different types of CityGML elements are mapped to nodes, relationships, and properties in the graph. However, these descriptions do not serve as static rules for the mapping methods proposed in Algorithms 1 to 3, but rather describe their outcomes. These methods can automatically produce a graph representation from any CityGML element, **without requiring any manual mapping rules**.

All non-zero cells of Table 3.2 are blue. This indicates that the generated graph representations of all CityGML datasets have achieved the value of 100 % for the type, instance, and relationship coverage, as well as the XLink replacement, leading to the total preservation of thematic and structural data of the original CityGML documents. Similar results have been consistently observed across all datasets tested thus far.

Table 3.2.: Assessing the preservation of thematic and structural content in the generated graph representations of the CityGML datasets *FZK-Haus* (KIT IAI, 2017) and *Railway-Scene* (Häfele & Nagel, 2015). The total number of occurrences of each CityGML element and attribute per dataset is shown in the cells located in the respective rows and columns. Cells with full coverage are shown in blue. This table is an excerpt, showing only the first and last two pages of the full version in Table A.1.

Lvl.	**Name of CityGML Element or Attribute**	**Corresponding Node Label, Rel Type, or Prop Name**	**FZK-Haus in LOD** 0	1	2	3	4	**Rail-way**
CityGML elements in ascending XML levels:								
00	*core:CityModel*	Node L	1	1	1	1	1	1
01	*app:appearanceMember**	Rel l + Node L	0	0	0	0	2	151
01	*core:cityObjectMember**	Rel l	1	1	1	1	1	52
01	*gml:boundedBy*	Rel l	1	1	1	1	1	0
01	*gml:name*[t]	Rel l	1	1	1	1	1	0
02	*app:Appearance*	Node L	0	0	0	0	2	151
02	*bldg:Building*	Node L	1	1	1	1	1	3
02	*brid:Bridge*	Node L	0	0	0	0	0	4
02	*dem:ReliefFeature*	Node L	0	0	0	0	0	1
02	*frn:CityFurniture*	Node L	0	0	0	0	0	11
02	*gen:GenericCityObject*	Node L	0	0	0	0	0	2
02	*gml:Envelope*	Node L	1	1	1	1	1	0
02	*grp:CityObjectGroup*	Node L	0	0	0	0	0	1
02	*tran:Railway*	Node L	0	0	0	0	0	10
02	*tun:Tunnel*	Node L	0	0	0	0	0	4
02	*veg:SolitaryVegetationObject*	Node L	0	0	0	0	0	15
02	*wtr:WaterBody*	Node L	0	0	0	0	0	1
03	*app:surfaceDataMember**	Rel l	0	0	0	0	2	151

[t] Contains texts * Multi-instances Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	**Name of CityGML Element or Attribute**	**Corresponding Node Label, Rel Type, or Prop Name**	**FZK-Haus in LOD** 0	1	2	3	4	**Rail-way**
03	*app:theme*^{t}	Prop l	0	0	0	0	0	151
03	*bldg:address*	Rel l	1	1	1	1	1	0
03	*bldg:boundedBy*	Rel l	0	0	7	7	7	40
03	*bldg:class*^{t}	Rel l + Node *Code*	1	1	1	1	1	0
03	*bldg:function*^{t,*}	Rel l + Node *Code*	1	1	1	1	1	1
03	*bldg:interiorRoom**	Rel l	0	0	0	0	7	0
03	*bldg:lod0FootPrint*	Rel l	1	0	0	0	0	0
03	*bldg:lod0RoofEdge*	Rel l	1	0	0	0	0	0
03	*bldg:lod1Solid*	Rel l	0	1	0	0	0	0
03	*bldg:lod2Solid*	Rel l	0	0	1	0	0	0
03	*bldg:lod3Solid*	Rel l	0	0	0	1	0	0
03	*bldg:lod4Solid*	Rel l	0	0	0	0	1	0
03	*bldg:measuredHeight*^{t}	Rel l + Node *Length*	1	1	1	1	1	0
03	*bldg:outerBuildingInstallation**	Rel l	0	0	0	0	0	56
03	*bldg:roofType*^{t}	Rel l + Node *Code*	1	1	1	1	1	0
03	*bldg:storeysAboveGround*^{t}	Prop l	1	1	1	1	1	0
03	*bldg:storeysBelowGround*^{t}	Prop l	1	1	1	1	1	0
03	*bldg:usage*^{t,*}	Rel l + Node *Code*	1	1	1	1	1	0
03	*bldg:yearOfConstruction*^{t}	Rel l + Node *LocalDate*	1	1	1	1	1	0
03	*brid:class*^{t}	Rel l + Node *Code*	0	0	0	0	0	2
03	*brid:function*^{t,*}	Rel l + Node *Code*	0	0	0	0	0	2
03	*brid:lod3MultiSurface*	Rel l	0	0	0	0	0	4
03	*brid:outerBridgeConstruction**	Rel l	0	0	0	0	0	3

...

^{t} Contains texts * Multi-instances Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	**Name of CityGML Element or Attribute**	**Corresponding Node Label, Rel Type, or Prop Name**	FZK-Haus in LOD 0	1	2	3	4	**Rail-way**
		. . .						
09	*gml:lowerCorner*^{t}	Rel *l* + Node *DirectPosition*	0	0	0	0	0	43
09	*gml:name*^{t,*}	Rel *l* + Node *Code*	0	0	0	0	15	0
09	*gml:posList*^{t}	Rel *l* + Node *DirectPositionList*	2	0	0	0	0	38,819
09	*gml:surfaceMember**	Rel *l*	0	0	0	205	14,511	167
09	*gml:upperCorner*^{t}	Rel *l* + Node *DirectPosition*	0	0	0	0	0	43
09	*xAL:PostalCodeNumber*^{t,*}	Rel *l* + Node *L*	1	1	1	1	1	0
09	*xAL:ThoroughfareName*^{t,*}	Rel *l* + Node *L*	1	1	1	1	1	0
09	*xAL:ThoroughfareNumber*^{t,*}	Rel *numberOrRange* + Node *L*	1	1	1	1	1	0
10	*gml:CompositeSurface*	Node *L*	0	0	0	0	16	0
10	*gml:LinearRing*	Node *L*	0	6	7	5	5	26,770
10	*gml:MultiSurface*	Node *L*	0	0	0	0	19	0
10	*gml:OrientableSurface*	Node *L*	0	0	0	0	70	0
10	*gml:Polygon*	Node *L*	0	0	0	205	14,425	167
11	*gml:baseSurface*^{t}	Rel *l*	0	0	0	0	70	0
11	*gml:exterior*	Rel *l*	0	0	0	205	14,425	167
11	*gml:interior**	Rel *l*	0	0	0	13	36	24
11	*gml:pos*^{t,*}	Rel *l* + Node *DirectPosition*	0	0	37	33	33	0
11	*gml:posList*^{t}	Rel *l* + Node *DirectPositionList*	0	6	0	0	0	26,770
11	*gml:surfaceMember**	Rel *l*	0	0	0	0	232	0
12	*gml:LinearRing*	Node *L*	0	0	0	218	14,461	191
12	*gml:OrientableSurface*	Node *L*	0	0	0	0	16	0
12	*gml:Polygon*	Node *L*	0	0	0	0	216	0
13	*gml:baseSurface*^{t}	Rel *l*	0	0	0	0	16	0

^{t} Contains texts * Multi-instances Left value in *l* 'camelCase' or *L* 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
13	*gml:exterior*	Rel *l*	0	0	0	0	216	0
13	*gml:interior**	Rel *l*	0	0	0	0	11	0
13	*gml:pos*^{t,*}	Rel *l* + Node *DirectPosition*	0	0	0	1,364	70,420	0
13	*gml:posList*^{t}	Rel *l* + Node *DirectPositionList*	0	0	0	0	0	191
14	*gml:LinearRing*	Node *L*	0	0	0	0	227	0
15	*gml:pos*^{t,*}	Rel *l* + Node *DirectPosition*	0	0	0	0	1,302	0
CityGML attributes:								
	codeSpace	Prop *l*	4	4	4	4	4	0
	gml:id	Prop *l*	1	1	22	458	29,383	110,047
	name	Prop *l*	3	3	3	3	3	0
	orientation	Rel *l* + Node *Sign*	0	0	0	0	86	0
	ring	Prop *l*	0	0	0	0	0	14,106
	srsDimension	Prop *l*	5	9	3	3	3	66,025
	srsName	Prop *l*	1	1	1	1	1	230
	Type	Prop *L*	2	2	2	2	2	0
	uom	Prop *l*	2	2	2	2	2	0
	uri	Prop *l*	0	0	0	0	0	13,933
	xlink:href	Rel *object* $\rightarrow$ Node found by *href*	0	0	7	20	106	26
Total preservation of thematic and structural content			☑	☑	☑	☑	☑	☑

CityGML module prefixes: *Core* (*core*), *Appearance* (*app*), *Building* (*bldg*), *Bridge* (*brid*), *Relief* (*dem*), *CityFurniture* (*frn*), *Generics* (*gen*), *CityObjectGroup* (*grp*), *LandUse* (*luse*), *TexturedSurface* (*tex*) (deprecated), *Transportation* (*tran*), *Tunnel* (*tun*), *Vegetation* (*veg*), and *WaterBody* (*wtr*) (Gröger et al., 2012). XML specification prefixes: GML (*gml*) (Cox et al., 2004), xAL (*xAL*) (CIQ TC, 2002), and XLink (*xlink*) (W3C, 2006).

Node Represented as a node   Rel Represented as a relationship   Prop Stored as a node property

l Value of the left column in 'camelCase' (without namespace)   *L* Value of the left column in 'CamelCase' (without namespace)

^{t} Element contains text content * Multiple instances may exist

3.6. Reconstruction of CityGML Objects from Graphs

This chapter mainly focuses on the creation of graphs that can accurately represent CityGML objects. However, many applications, including the change detection and interpretation process introduced in later chapters, require the reconstruction of these graphs back into their original in-memory objects. This is particularly useful for tasks such as matching two surfaces that were defined using different syntactic styles. Such tasks require the handling of geometric extents and point coordinates in 3D space, which can be achieved more efficiently using their object-oriented in-memory representations and associated object functions, rather than repeatedly parsing their graph contents from Neo4j, which officially only supports spatial operations of points (Neo4j, 2023). As a result, there is no need to represent these geometries in any additional structures, such as the Well-known Text (WKT) and Well-known Binary (WKB) representations (Herring, 2011) commonly used in Spatially-enhanced Relational Database Management Systems (SRDBMSs). Ultimately, in conjunction with the mapping methods, this reconstruction process could be scaled up to facilitate the import and export of entire CityGML documents to and from a graph database. This section addresses Research Question RQA7 (Reconstruction of Graphs to CityGML Objects).

The methods for reconstructing graphs back into their original in-memory CityGML objects are detailed in Algorithms 4 and 5. These procedures can be thought of as the reverse counterparts of the methods for mapping CityGML objects onto graphs, as shown earlier in Algorithms 1 and 2. The first method *toObject(node)* sets up the parameters required for invoking the second method *toObject(node, visited)*, which recursively performs most of the reconstruction process.

Algorithm 4: Graph to CityGML conversion *toObject(node)*

```
  Input  : A node generated from the mapping process
  Output: The original CityGML object represented by node and its subgraph
1 visited ← new key-value map with nodes as keys and converted objects as values
2 return toObject(node, visited)
```

3.6.1. Enriching Graph Representations of CityGML Objects

The methods *map(source)*, *map(source, visited)*, and *resolveXLinks(graph)* detailed in Algorithms 1 to 3 are designed with a focus on high flexibility and universality, making them suitable for mapping a wide range of objects, including but not limited to CityGML objects, onto their respective graph representations. The advantages are two-fold: (1) the methods do not require any additional manual mapping rules for any specific types

Algorithm 5: Recursive graph to CityGML conversion *toObject(node, visited)*

```
Input  : A node generated from the mapping process
         Key-value map visited storing nodes and converted objects
Output: The original CityGML object object represented by node and its subgraph
1  if visited.hasKey(node) then return visited.getValue(node)
2  if node represents an array then
3      arrayMemberType ← node.getProperty("arrayMemberType")
4      arraySize ← node.getProperty("arraySize")
5      object ← a new empty array of arrayMemberType and arraySize
6      visited.add(node, object)
7      if node does not have any outgoing ARRAY_MEMBER relationships then
           // All array members are simple and stored as properties
8          foreach property prop of node do
9              index ← extract index value from the name of prop
10             object[index] ← the value of prop cast to arrayMemberType
11         end
12     else
           // All array members are complex and stored as child nodes
13         foreach relationship ARRAY_MEMBER rel outgoing from node do
14             endNode ← rel.getEndNode()
15             index ← rel.getProperty("arrayMemberIndex")
16             object[index] ← toObject(endNode, visited) cast to arrayMemberType
17         end
18     end
19 else
20     object ← a new empty instance of class node.getLabel()
21     visited.add(node, object)
22     foreach property prop available in object.getClass() do
23         if node.hasProperty(prop) then
24             propType ← node.getProperty(prop + "Type")
25             object.get(prop) ← the value of prop cast to propType
26         else
27             endNode ← node.getOutgoingRelationship(prop).getEndNode()
28             object.get(prop) ← toObject(endNode, visited)
29         end
30     end
31 end
32 return object
```

of objects, and (2) the absence of such manual mapping rules greatly simplifies the process of reconstructing the graph representations back to their original objects. This is because, with each additional manual rule introduced during the mapping process, a corresponding reverse rule must also be implemented in the reconstruction process. The mapping methods described in previous sections, as well as the reconstruction methods proposed in this section, eliminate the need for any of such manual rules.

However, since the graph database Neo4j does not impose any schemas on its stored content and employs its own data types for properties that may not align with the data types used in the original objects, such type information is lost or changed during mapping. Although this is typically not an issue for many processes that utilize the content of these graph representations, the type information is crucial for reconstructing these graphs back to their original in-memory objects. Therefore, to facilitate this reconstruction process, graph representations of CityGML objects are supplemented with additional metadata. This applies to:

1. **Array Node Representation**: If the current node represents an array, it is enriched with additional properties such as the array member type *arrayMemberType* and the array size *arraySize*.
   a) Each simple character-based or numeric array member is represented as a node property named *arrayMember_i*, where *i* corresponds to the index in the original in-memory array object.
   b) Each complex array member is represented as a subgraph and is connected with the array node through a relationship of type *ARRAY_MEMBER*. This relationship contains an additional property *arrayMemberIndex* indicating the index of the current array member.

   Despite the order of sibling elements do not play any role, the order of array members is preserved explicitly as properties to ensure the structure of the reconstructed objects closely resembles that of the original CityGML objects. This is particularly useful when reconstructing an array of 3D coordinates to points. These additional graph entities are created in Lines 7, 8, 11, 14, and 15 of Algorithm 2 and utilized in Lines 3, 4, 9, 13, and 15 of Algorithm 5.

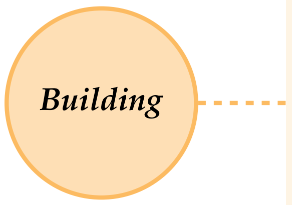
2. **Non-array Node Representation**: For each character-based or numeric property of a non-array node, an additional property is created to store the type of its corresponding property. For example, a building node is not an array and has a property *creationDate*. Thus, an additional property *creationDateType* is created to explicitly store the type of the property *creationDate*, which is *java.time.ZonedDateTime* in Java. Such additional properties are created in Line 24 of Algorithm 2 and employed in Line 24 of Algorithm 5.

3. **XLink Relationship Representation**: If the current node has an incoming relationship representing an XLink connection, a property *wasXLink* is added to this relationship. Although the generated graph representations of CityGML documents do not distinguish between elements defined with or without XLinks, this information is preserved as explicit properties to ensure the structure of the resulting reconstructed objects closely matches that of the original CityGML documents. This is implemented in Line 8 of Algorithm 3. Contrary to other supplementary metadata, the property *wasXLink* is not employed during the process of reconstructing graphs into CityGML objects. This is because in-memory CityGML objects, like their graph representations, replace the implicit XLink connections with explicit references or relationships, thereby eliminating the distinction between cases with or without the use of XLinks. Instead, the property *wasXLink* is primarily used for exporting the reconstructed in-memory CityGML objects back into their original CityGML documents, which can contain XLinks. However, this is done only when the structure of the exported CityGML documents needs to closely mirror that of the original.

This metadata, which is not included in the original CityGML documents, should not be confused with the metadata already available according to the GML and CityGML information model. For instance, the definition and description of the local Coordinate Reference System (CRS) contained in the element *gml:metaDataProperty* are part of the existing metadata. If such elements are present in the input CityGML documents, they will be mapped onto graphs like every other element.

Figure 3.14 provides an illustration of the additional metadata properties of the building node generated from the *FZK-Haus* CityGML dataset (KIT IAI, 2017). On the other hand, Figure 3.15 demonstrates how a subgraph representation of a composite surface of a solid geometry of the building from the *FZK-Haus* dataset can be enriched. In this case, the seven member surfaces are grouped by an array node. The paths between the array node and the polygon nodes consist of two relationships: *ARRAY_MEMBER* and *object*. While the relationships *ARRAY_MEMBER* are enriched with the index value of their respective surfaces within the array, the relationships *object* are supplemented with the property *wasXLink*, indicating that the solid geometry was defined by reusing the existing polygons in the dataset.

The majority of the methods proposed in this study are programming language-agnostic and can be implemented using any compatible programming languages. However, the data types used to enrich nodes and relationships, like *java.lang.Integer* and *java.lang.String* in Java, are specific to the programming language used in the mapping process. Consequently, the reconstruction methods must also be implemented using the same programming language as in the mapping process.



Node Properties

Name	**Value**
id	UUID_d281adfc...
idType	*java.lang.String*
creationDate	2017-01-23
creationDateType	*java.time.ZonedDateTime*
storeysAboveGround	2
storeysAboveGroundType	*java.lang.Integer*
storeysBelowGround	0
storeysBelowGroundType	*java.lang.Integer*

Figure 3.14.: An illustration of the building node generated from the *FZK-Haus* dataset (KIT IAI, 2017) and the metadata stored as its additional properties (blue).

As explained in Section 3.3.4, arrays are a fundamental data type in many programming languages including Java, not only in direct definitions but also as an internal component of many other data structures, such as *ArrayList*. The library *citygml4j* manages collections of CityGML objects using the class *ChildList*$\langle$*T extends Child*$\rangle$, which is a subclass of *ArrayList*$\langle T \rangle$. Here, *Child* is an interface implemented by all CityGML objects, and T represents a generic type according to Java's generic typing. This generic type T serves as a placeholder for all subclasses of the class *Child*, which is its *bound*.

For instance, the single building of the *FZK-Haus* dataset in LOD2 contains a *Solid* geometry, which contains a *CompositeSurface*, which in turn contains a *ChildList* collection of all seven boundary surfaces of the building. Since *ChildList* is a subclass of *ArrayList*, the boundary surfaces are grouped using an *Array* node, as shown in Figure 3.15.

The property *arrayMemberType* of an *Array* node denotes the common type of all members of that array. For an array of point coordinates as floating-point numbers (i.e., of type *Double*[ ]), all members of this array are of the same type *java.lang.Double*. However, as arrays are also used internally in *ArrayList* and *ChildList*, the value of *arrayMemberType* is often *Object*, the superclass of all Java classes, as shown in Figure 3.15.

This is a consequence of **type erasure** in Java, where the generic types are replaced with their bounds, or *Object* if the types are unbounded, at compile time. This replacement leverages polymorphism, an object-oriented concept that allows a superclass to serve as a placeholder for all its subclasses. Despite this type erasure, the additional metadata provided is sufficient for the reconstruction of graphs back into fully functional CityGML objects.

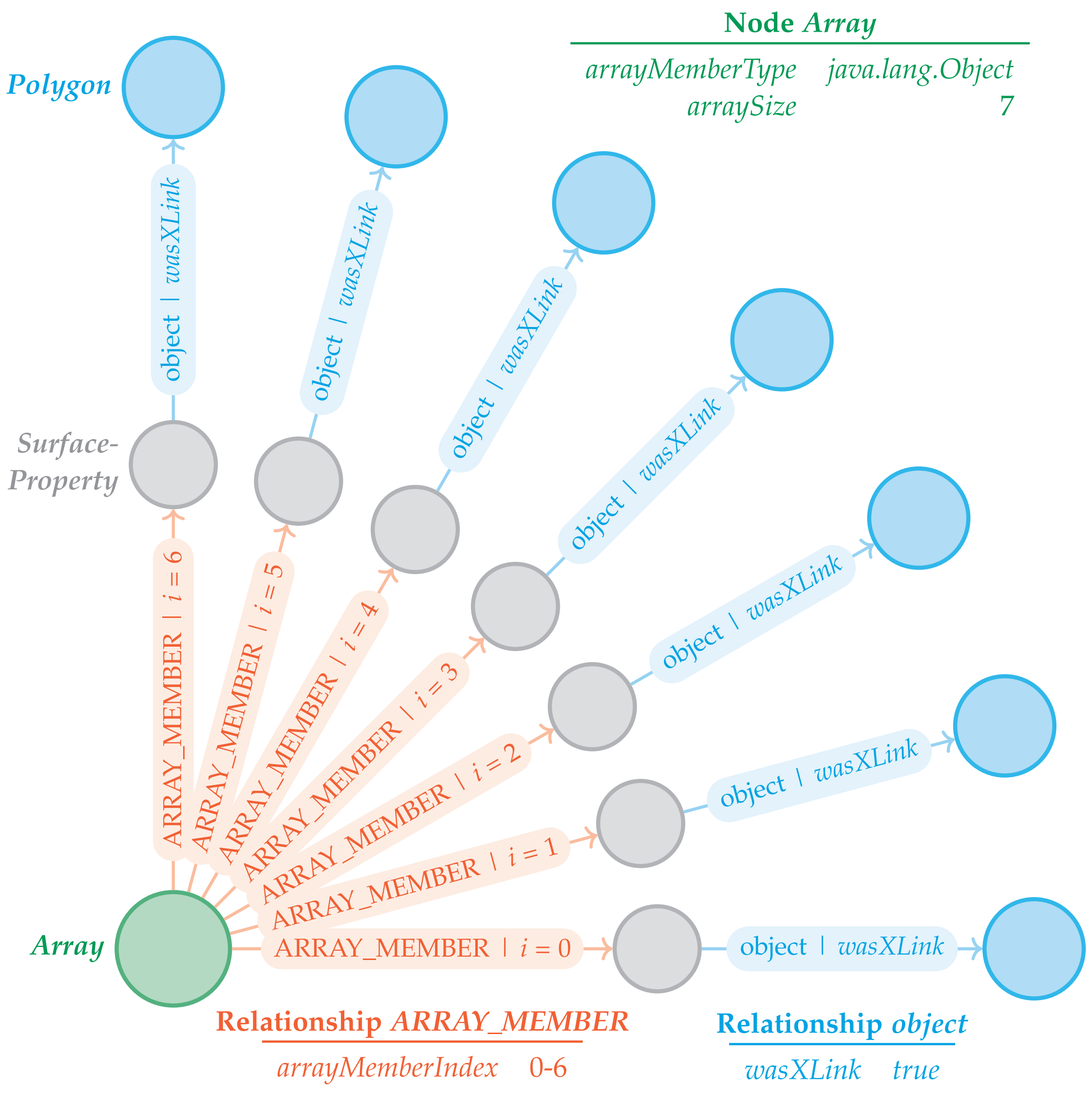


Figure 3.15.: An illustration of a node representation of an array of polygons with additional metadata. This array node is used to group the member surfaces of the solid geometry of the *FZK-Haus* dataset in LOD2 (KIT IAI, 2017).

3.6.2. Key Characteristics of the Reconstruction Methods

Similarly to the mapping methods introduced in Section 3.3, the key characteristics of the reconstruction methods are outlined in this section.

Recursive Reconstruction

The recursive method detailed in Algorithm 5 (refer to Lines 16 and 28) operates in a top-down manner. It rebuilds an in-memory object starting with a given graph node, followed by its properties and child nodes, then the properties and child nodes of those child nodes, and so forth. The process terminates when there are no more elements left for converting.

Preventing Repeated Reconstruction

When an object is referenced by multiple parents, such as when a boundary wall surface is shared between two adjacent rooms through the use of XLink, the subgraph representation of that child element obtains multiple incoming relationships from its respective parents. Since the reconstruction methods operates in a top-down manner, the same subgraph may be reconstructed multiple times. As a result, each of the aforementioned adjacent rooms would then gain a separate boundary wall surface, despite both surfaces being identical.

To prevent this redundancy, a key-value store, denoted as *visited* in Algorithm 4, is employed. This store is a map over tuples *(Node, Object)*, where the node representations serve as keys and their corresponding in-memory objects serve as values. The keys are unique, as each node can be identified by their allocated address in main memory within a transaction during execution or simply by their identifiers. In Neo4j, each node is automatically assigned an identifier. However, these internal identifiers are not unique, as the identifiers of deleted nodes may be recycled for new nodes (Neo4j, 2023). Therefore, this study (1) introduces a unique identifier for each node of specific important types such as top-level features and geometric elements, and (2) avoids deletion of nodes in all processes when possible.

During conversion of each node into objects, the process first determines whether the node has been previously visited and converted by searching for its key in the key-value map (refer to Line 1). The reconstruction proceeds only if the key does not exist, indicating that the node has not been visited before. Otherwise, the previously reconstructed in-memory object associated with this node is returned instead. This ensures that no node or subgraph is converted more than once, thereby guaranteeing termination of the algorithms and accurate reconstruction of complex references in the resulting in-memory CityGML objects.

Reconstructing Arrays

Based on the complexity of its members, a node representation of an array may store its members as properties or as child nodes, as dictated by Figure 3.7 during the mapping process. Therefore, the reconstruction of such arrays from their corresponding graph representations can be described as follows (refer to Line 7 through Line 18):

1. If the array node does not have any outgoing relationships of the predefined type *ARRAY_MEMBER*, then all its original members are of simple types and are stored as the node's properties. In this case, the array member type, the index value, and the content of each member can be extracted from the node's corresponding properties.

2. If the array node possesses at least one outgoing relationship of the predefined type *ARRAY_MEMBER*, then there must be exactly as many such relationships as there are members of the array, since all members of an array must be of the same type and thus mapped and reconstructed in the same manner. In this case, the end node of each such relationship represents an array member. The type and size of the array can be extracted from the corresponding properties stored in the array node, while the index value of each array member can be retrieved from its respective relationship outgoing from the array node. The array object can then be reconstructed by recursively reconstructing each of its member nodes.

Reconstructing Non-array Complex Objects

In contrast to arrays, non-array objects can contain various properties of primitive or complex types, as dictated by the data model and the classes they belong to. Therefore, to rebuild a complex object from its graph representation, the first and most important step is to create an empty instance of its class. The object's class can be retrieved from the label of its node representation.

In Java and many other object-oriented programming languages, the **default constructor** of a class can be invoked to create an instance with default values for its properties. This approach requires that the default constructor is available, as already is the case with employed CityGML classes provided by the library *citygml4j* in Java.

Alternatively, **duck typing** can be used. In computer programming, duck typing is an application of the abductive duck test (Devopedia, 2023): 'If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.' Duck typing allows for classifying any object as a certain type if it possesses the necessary properties and methods. This is useful when no constructors are available, or no type checking is enforced.

Once an empty object has been instantiated, its default values are then filled or updated based on the content derived from its graph representations. This is described as follows:

1. If a property exists in the node representation and a property with the same name is declared in the object model, then the node property represents a 'simple' property of the instantiated object, as previously explained in Section 3.3.6. In this case, the value and type of the node property are retrieved to update the corresponding property of the object (refer to Lines 24 and 25). For example, the node properties *creationDate* and *creationDateType* are used to update the simple property *creationDate* of a building object.
2. For each outgoing relationship from the node representation, if the relationship's type matches a property allowed by the object model, the end node to which the relationship points is used to reconstruct that property (refer to Line 28). For instance, the outgoing relationship *boundedBy* from a building node points to a subgraph representing a *BoundingShape* that describes the bounding box of that building, as previously illustrated in Figure 3.6.

The major advantage of this method lies in its **selective conversion**, where only graph contents matching the names and types of corresponding properties in the object model are considered. As a result, this targeted approach dismisses auxiliary or internally used graph entities, such as helper nodes, relationships, or properties of nodes and relationships, allowing them to coexist within the graphs without being included in the reconstructed in-memory objects. This is achieved due to the coherent mapping between the CityGML data model and its graph representations employed in this research. This name and type checking ensures the consistent conversion from the graph representations back into their original CityGML objects.

3.7. Summary and Discussion

This chapter discusses the key characteristics of the City Geography Markup Language (CityGML) that are relevant to this study, such as the semantic modelling, multi-scale modelling using different Level of Details (LODs), modularization of thematic modules, coherent spatio-semantic modelling, and geometric-topological modelling. The chapter then explores the object-oriented and relational representations of CityGML objects. Given the graph-based nature of CityGML, a graph data model is proposed for representing the CityGML data model.

This chapter then introduces methods for mapping CityGML objects onto their graph representations. These methods are recursive and highly flexible, capable of mapping any objects, including CityGML objects, without the need for any manual mapping rules. This chapter also explains how XLinks present in the original CityGML documents can be resolved in the produced graphs. The mapping methods are evaluated across all five different LODs and all fourteen thematic modules of CityGML.

Finally, the chapter proposes reconstruction methods, the reverse counterpart to the mapping methods to convert graphs back to their original CityGML objects. Like the mapping methods, the reconstruction process is recursive and highly flexible, capable of reconstructing any objects without the need for any manual reconstruction rules. This is achieved by enriching the graph representations of CityGML objects with additional metadata, such as the types of individual objects, as well as the member types, member indices, and the size of an array. Both the mapping and reconstruction process are implemented using the graph database Neo4j.

Some notable observations and insights related to the concepts introduced in this chapter include:

1. **Mapping of Thematic Content**: The mapping methods outlined in Algorithms 1 to 3 transform CityGML documents into labelled, attributed graphs. These graphs contain a wealth of thematic information, primarily stored within the key-value mappings assigned to each node. Additionally, the node labels and relationship types reflect the classes of CityGML objects and the nature of their interrelationships, respectively. Unlike conventional graph labellings in graph theory, such as 'graceful labelling', where no two distinct vertices are labelled the same or no two distinct edges connect the same pair of vertices, the node labels and relationship types in this study can be reused when necessary to represent the semantic content of the original CityGML objects. The values of these labels and types, as well as their labelling rules, are determined solely based on the schemas provided by the CityGML data model (Gröger et al., 2012).

2. **Mapping of Structural Content**: The structure of the graph representations of CityGML documents adheres to the graph data model proposed in Section 3.2 and closely mirrors that of the CityGML object model. Algorithms 1 and 2 dictate that the resulting graphs are directed, with relationships consistently pointing from objects to their respective components. Algorithm 3 ensures that these graphs exhibit weak connectivity. As a result, the graphs generated by these methods are Directed Acyclic Graphs (DAGs), as explained in Sections 3.1.4 and 3.1.5, and the existence of exactly one (universal) source node is guaranteed in each graph, such as the city model node. This implies two major advantages: (1) recursive methods, such as those employed in the change detection and interpretation process, can cover the entire graphs given their respective source nodes, and (2) all methods operating on these graphs can terminate due to the absence of directed cycles.

3. **Evaluation of the Mapping Methods**: As previously explained in Section 3.1.7, Neo4j operates as an instance-based graph database. This implies its capability to capture information of specific individual CityGML objects, rather than describing the underlying information model itself. As a result, the evaluation of both thematic and structural information preservation in generated graphs is conducted against their original CityGML datasets. This does not provide any verification of whether these contents adhere the CityGML encoding standard. Thus, this study assumes that the input CityGML datasets used for mapping already comply with the CityGML schemas, as mentioned in Section 3.2.1.

4. **Enhanced Information Representation**: The generated graph representations not only fully capture all thematic and structural information from the input CityGML documents, as shown in Section 3.5.3, but they also contain more information than explicitly available in the original documents. Such additional information is crucial for the subsequent processes. For example, additional metadata information is employed for reconstructing graph representations back into their original in-memory objects, as described in Section 3.6. Moreover, the graphs can also be enriched with an R-tree structure that contains the 2D footprints of all top-level features in the city model, as explained in Chapter 4. This R-tree, serving as a spatial index, is essential to minimize the time required to find the best candidates that spatially matches a reference object. The process of constructing such an R-tree is explained in Section 6.3.2.

5. **The Use of the City Model Node**: Each CityGML top-level feature is often mapped onto a separate subgraph. These subgraphs can be reached from a central node that represents the city model object. This graph structure allows for (1) better multi-threaded performance on isolated subgraphs, and (2) enhanced query time during traversing due to the reduced number of potential paths between nodes. The city model nodes are often the most important nodes in large CityGML datasets, as they have the highest *out-degree centrality*. This means that they are connected to many other nodes in the graph and can serve as a starting point for various traversal operations, as observed in many methods used in the change detection and interpretation process in the next chapters.

6. **Memory Footprint**: Both the mapping and reconstruction methods operate recursively in a top-down manner. They start from a source object or node and successively process all descendant elements. As a result, the memory consumption of these methods is proportional to the size of the source object or the subgraph, including all nodes and relationships reachable from the source node. In the worst-case scenario, this memory footprint could contain the entire

city model if the source object is a city model or if the source node represents a city model object. To reduce the memory footprint of the mapping methods, large input CityGML documents can be first divided into smaller segments, each of which is then mapped onto graphs sequentially. Such optimization strategies for handling massive CityGML datasets are explained in Chapter 6. On the other hand, since in-memory CityGML objects are held entirely in main memory, the reconstruction methods should be applied only to small subgraphs, such as the graph representations of individual geometries like polygons and solids.

7. **Export of Graphs to CityGML Datasets**: By utilizing the reconstruction methods outlined in Algorithms 4 and 5, along with the built-in functions of the library *citygml4j*, graph representations of city objects can be both reconstructed into their corresponding in-memory CityGML objects and re-exported to the original CityGML datasets. During this process, XLinks are automatically assigned based on the existing ones in the original datasets, or new ones are generated to ensure that each city object is exported only once.

8. **Creation and Modification of City Models using Cypher**: Instead of first mapping input CityGML datasets onto graphs, an entire city model can alternatively be built directly using Cypher scripts, which contain the structure and content of the city model in compliance with the CityGML data model. To generate these scripts, CityGML documents can be manually written in Cypher, or Neo4j's built-in functions can be used to automatically convert an existing CityGML graph into its corresponding Cypher scripts for later use. Once the city model has been created in the database, it can be edited, updated, and eventually exported into a new CityGML dataset. As Neo4j is schema-less, the city model graph can be modified freely. However, when reconstructing CityGML objects and exporting them to CityGML documents, only information conforming to the CityGML data model is considered. This selective process is done automatically without any need for manual intervention.

9. **Compatibility with CityGML versions 2.0 and 3.0**: The methods proposed in this chapter are highly flexible and can be applied to any in-memory objects and compatible graphs without the need to define any manual mapping and reconstruction rules. As a result, datasets of both CityGML versions 2.0 and 3.0, including all LODs and thematic modules, can be mapped onto graphs. Conversely, any compatible graph representations can be converted back into their original CityGML objects, even if these graphs have been enriched with information not prescribed by the CityGML data model, such as the various auxiliary nodes, relationships, and properties utilized in this study.

10. **Applications of the Generated Graph Representations**: In most use cases, the creation of the graph representations of CityGML models introduced above does not mark the end of the workflow. Instead, they are further used as a basis for a multitude of applications. These include, but are not limited to, the processes for detecting changes and interpreting their patterns, as explained in Chapters 4 and 5, respectively. Additional applications discussed in other studies include the use of a graph representation of IndoorGML documents to facilitate routing in indoor environment, such as finding an optimal route between two offices inside a building (Jang et al., 2023). These graph representations can also be enhanced with supplementary information from external data sources, such as OSM data (Ding et al., 2024). In another example, a graph representation of a segment of the detailed street and lane network of the cities of Ingolstadt and Grafing near Munich, utilizing the street space models of CityGML version 3.0, is employed for complex multimodal navigation between two locations (Olbrich, 2023; Olbrich et al., 2024).

In this chapter, the methods for mapping CityGML objects onto graphs are evaluated in Section 3.5. An excerpt of the evaluation results can be found in Table 3.2. The full version is available in Table A.1.

Conversely, to evaluate the reconstruction methods, the following steps can be applied:

1. Map a CityGML document onto its graph representation.
2. Convert this graph into a city model object, which is then exported to a new CityGML document.
3. Map the newly created CityGML document onto its graph representation.
4. Compare both the generated graph representations of the original and new CityGML document.

The first three steps can already be performed by employing the mapping and reconstruction methods proposed in this chapter. The export of a city model object into a CityGML document can be accomplished using the library *citygml4j*. In the last step, by comparing their graph representations, the similarity in the structure and content of both the original and newly created CityGML document can be evaluated. If the graphs are identical, both the mapping and reconstruction methods are considered to produce reliable results. This requires the ability to compare complex graphs, which can be achieved by employing the methods for comparing graph representations of CityGML documents presented in Chapter 4.

4. Change Detection in Semantic 3D City Models

Identifying changes between two different temporal versions of a CityGML document can present significant challenges when attempted through direct comparisons. This is primarily due to the structural constraints of plain CityGML text files, which impose a fixed order of occurrence when reading elements sequentially. At the same time, these text files have limitations in expressing explicit one-to-many and many-to-many relationships between elements. Even with the use of XLinks, sequential reading of plain CityGML text files cannot handle cases where an element is referenced before it has been parsed or after it has been discarded due to limited memory capacities in earlier read iterations.

Matching CityGML elements based on their relational representations like in the 3DCityDB is also difficult, as it may require a high number of *JOIN* operations on a regular basis, leading to significant performance costs. On the other hand, while object-oriented representations are ideally suited for matching the content and structure of CityGML objects, this approach is only feasible for small datasets, as it requires objects to be held entirely in main memory, leading to high memory consumption.

In this context, a more effective approach is to employ the graph representations of these CityGML documents instead. As previously shown in Chapter 3, these graphs can accurately capture all thematic and structural information available in the original CityGML documents and are thus fully capable of representing them during the matching process. Therefore, this chapter proposes methods to detect changes between these graph representations, thereby reflecting the changes that occurred between the corresponding CityGML documents.

The content of this chapter substantially expands upon the author's earlier publications, which are detailed as follows:

1. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2017, October). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database [12th International 3D GeoInfo Conference 2017, University of Melbourne, Melbourne, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 99–106, Vol. IV-4/W5). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-iv-4-w5-99-2017.

2. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2018). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database. In gis.Science (pp. 85–100, Vol. 3). Wichmann Verlag. https://gispoint.de/artikelarchiv/gis/2018/gisscience-ausgabe-32018.html.

3. Nguyen, S. H., & Kolbe, T. H. (2020, September). A Multi-Perspective Approach to Interpreting Spatio-Semantic Changes of Large 3D City Models in CityGML using a Graph Database [15th International 3D GeoInfo Conference 2020, University College London (UCL), London, UK]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 143–150, Vol. VI-4/W1-2020). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-vi-4-w1-2020-143-2020.

4.1. Foundations and Related Work

This section establishes the necessary groundwork and discusses the literature relevant to the concepts and methods presented in this chapter. Research Question RQB1 (Related Comparison Methods) and the first half of Research Question RQB2 (Graph and Subgraph Isomorphism) are addressed in this section.

4.1.1. Existing *diff* Algorithms for XML, GML, and CityGML

Traditional change detection or data comparison algorithms, often referred to as *diff* tools (Hunt & Szymanski, 1977; Myers, 1986; Wagner & Fischer, 1974), were primarily designed for comparing unstructured plain texts. However, the CityGML data model, despite often being encoded as text files, is a GML application schema, which, in turn, is an XML grammar. Both XML and GML, and especially CityGML, are structured exchange formats and cannot be treated as plain texts for comparison purposes. For instance, in the context of comparing XML text files, the order of sibling elements is predetermined when read from top to bottom. However, this sequence does not play any significant role, as the siblings can be reordered in another text file, and both files could still be identical in terms of content.

Every well-formed XML document must contain a single root element, serving as the starting point to access all other elements. While each XML element may have an arbitrary number of attributes and sub-elements, it can only be defined by exactly one parent element. Elements without any sub-elements are located at the lowest level and considered as leaf elements. As a result, the structure of each XML document mirrors that of a tree and can be conceptually depicted as a tree data structure, commonly known as an XML tree.

In graph theory, the term *tree*, first introduced in 1857 by the British mathematician Arthur Cayley (Cayley, 1857), refers to a mathematical structure consisting of a set of vertices and edges, where exactly one path must exist between any two vertices, making the tree loop-free. A *rooted tree* is a tree with a designated root vertex. In contrast, a tree without a designated root is referred to as a *free tree*. An *ordered tree* is a rooted tree, in which an ordering exists among the children of a vertex. In computer science, a tree is an Abstract Data Type (ADT) that implements the concept of rooted trees from graph theory, with nodes representing vertices and edges representing connections between nodes. A *binary tree* restricts the number of children a tree node can have to a maximum of two.

The comparison of XML documents has been a subject of extensive discussion. For example, a fast and memory-efficient *diff* algorithm, known as *XyDiff* (Cobena et al., 2002), was introduced for comparing ordered XML trees. This was accomplished by matching unchanged XML subtrees between the older and newer version. Subsequently, an effective algorithm named *X-Diff* (Wang et al., 2003) was proposed to detect changes between unordered XML trees. The research incorporated several key XML characteristics by introducing concepts such as node signature and XHash. Coupled with the standard tree-to-tree correction techniques (Zhang, 1993), the difference between two XML documents could be detected efficiently. The research argued that, despite the implementation being considerably more difficult compared to that for the ordered tree model, matching XML documents using their unordered tree model yielded more accurate results.

Building upon this unordered tree model, a later study (Redweik & Becker, 2014) extended *X-Diff* to more effectively incorporate spatial and geometric information present in the tree representation of CityGML documents. However, the approach did not factor in the use of XLinks, an important component in the GML and CityGML encoding standard that allows the reuse and linking of previously defined elements within the same document. The inclusion of XLinks would result in cycles in the employed tree representation of CityGML documents, contradicting the inherent definition of trees being loop-free. As a result, despite being XML-based, CityGML documents are basically considered as graphs, as explained in Chapter 3, and cannot be solely confined to tree structures (Schade & Cox, 2010).

Recent studies have suggested the use of graphs for comparing CityGML documents through their graph representations (Nguyen et al., 2017; Nguyen, 2017; Nguyen et al., 2018). Changes detected in these graphs can provide insights into the changes in the original CityGML documents. These studies have contributed to one of the first open-source implementations for change detection in semantic 3D city models. However, these studies employed a large number of manually defined rules for matching. While these rules were capable of managing the syntactic ambiguities of simple geometric objects, they fell short in supporting more complex geometric objects and their changes, such as the translation or resizing of a surface. In addition, due to the lack of support for automatic conversion of graphs back into their original in-memory CityGML objects, the implementation of these studies constantly referred to the graph database to compare the structure and content of these subgraphs, a process that is labour-intensive and inefficient compared to the object-oriented approach. Despite these limitations, these studies have been instrumental in laying the groundwork for the matching process proposed in this thesis.

In BIM, graphs have also been utilized to detect and document changes across different temporal versions of an IFC model (Esser, 2024; Esser et al., 2022). By leveraging graphs as a central medium for the representation of IFC documents and detailed documentation of their changes, these studies laid the groundwork for enabling version control and interdisciplinary collaboration in the built environment. Using well-defined formal descriptions of incremental changes, these studies were capable of handling diverging concurrent versions originating from a common document, as well as merging different versions back into one unified state. Changes in the thematic properties and topological positions of nodes within the graphs could be detected and directly accessed, allowing for exchange with other collaborating partners.

However, these studies did not provide further guidelines on how to match complex 3D geometries, including those that may have been translated or resized. They did not consider the possible different syntactic representations of the same (geometric) object allowed by the underlying encoding. In addition, as these studies employed graph and subgraph isomorphism, as will be introduced in Section 4.1.2, an open question remains whether the proposed methods could be applied to the very large datasets often found in the field of GIS. While the detected changes could be automatically described in great detail, it has not yet been established how these changes can be further interpreted to cater to specific needs of different stakeholders. Such questions are addressed throughout this thesis.

There exist studies that did not utilize graph representations of CityGML documents for detecting changes, such as in the use case of Lyon, France (Pédrinis et al., 2014), where a building may cover a large area with many polygons. The authors employed a purely geometric approach. Roofs were first projected onto the ground to extract buildings' footprints (in the absence of ground surfaces in the CityGML datasets). The projected footprints were then matched based on their intersections and predefined thresholds. Since the change detection process relied entirely on geometric matching, the Hausdorff distance (Hausdorff, 1914) was used as an additional check to further distinguish buildings with matching footprints, such as when a building had increased in height. Moreover, the CityGML geometries were further combined with cadastre data of the corresponding area to improve accuracy.

The aforementioned study could only provide information on whether a building had been deleted, inserted, or geometrically modified. Its methods could not detect more complex geometric changes, such as translations and resizes in 3D space. Moreover, cadastre information may not be always available. Additionally, without an expressive intermediate representation like graphs employed in this thesis, no further changes within the semantic contents of the matched buildings could be found, which is an important information aspect of CityGML that should be considered.

As will be explained in Section 4.5, this thesis also performs an overlap check as one of the first steps to quickly identify potential geometric matches. This applies to all geometries except points, such as line segments, surfaces, and solids. This type of geometric matching is employed not only for change detection in CityGML (Pédrinis et al., 2014; Redweik & Becker, 2014), but also in many other studies and tools for a variety of purposes. These include enriching 3D city objects with external information (Ding et al., 2024) and managing 3D city objects within a database like the 3DCityDB (Yao et al., 2018). As a result, there exist many variations to this overlap check. To prioritize speed, this research only considers axis-aligned minimum bounding boxes of city objects. Bounding boxes of top-level objects like buildings, if absent, are computed and stored during the mapping process, as explained previously in Chapter 3.

For the 3D geometries A and B of two such city objects, their respective volumes $v(A)$ and $v(B)$, their overlap $v(A \cap B)$, their union $v(A \cup B)$, and a sufficiently large threshold h (ideally close to 1), a geometric match can be established based on one of the following conditions:

$$\frac{v(A \cap B)}{v(A)} \geq h, \qquad \frac{v(A \cap B)}{v(B)} \geq h, \qquad \frac{v(A \cap B)}{v(A \cup B)} \geq h. \tag{4.1}$$

The first condition determines whether A is contained within B, while the second condition verifies whether B is contained within A. The third condition, utilizing the Jaccard index (Jaccard, 1912), ensures that both A and B are contained within each other, meaning that they are geometrically equal. This research utilizes all three of these variations across different cases.

4.1.2. Graph Isomorphism

In graph theory, an **isomorphism** between two graphs G and H is defined as a bijective mapping between their vertex sets V_G and V_H, respectively, $f : V_G \to V_H$, such that if two vertices $u, v \in V_G$ are adjacent in G, then the vertices $f(u), f(v) \in V_H$ are also adjacent in H. Therefore, an isomorphism between two graphs is a bijection that maintains adjacency (McKay & Piperno, 2014), often referred to as an 'edge-preserving bijection', as it conserves the structure within the graphs.

The aforementioned definition applies to graphs that are undirected, unlabelled, and unweighted. This suggests that the isomorphism of two graphs is not influenced by their node labels or layout but is solely determined by the similarity of their structure (Trudeau, 1994). Graphs that are isomorphic to each other belong to the same *isomorphism class*. Figure 4.1 illustrates a set of such graphs.

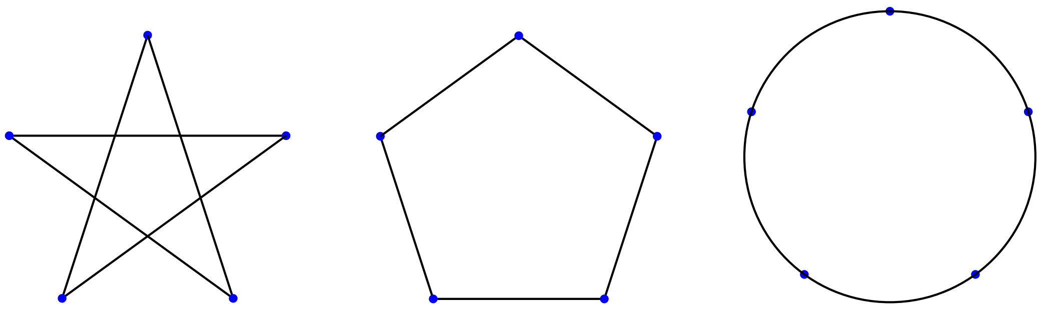


Figure 4.1.: An illustration of isomorphic graphs. By rearranging the vertices of the pentagon in the middle, the star graph on the left is formed. Similarly, by depicting the edges of the pentagon as curves, the circular graph on the right is created. All these graphs belong to the same isomorphism class.

In the context of directed or weighted graphs, the aforementioned definition of isomorphism can be extended with additional conditions to further ensure the preservation of both direction and weights. On the other hand, when dealing with labelled graphs, two cases are differentiated:

1. Each vertex and edge is unique and can be identified by a distinct label. For instance, the vertices and edges of the graphs shown in Figure 4.1 are distinct from each other. In this case, the isomorphism of these graphs is reducible to the standard isomorphism between two corresponding unlabelled graphs, also known as *normal graphs*.

2. The same labels may be assigned to multiple vertices or edges, grouping them into *equivalent classes*. Within each equivalent class, all vertices or edges have the same label. In this case, an isomorphism between such graphs must preserve both the structure and the equivalence classes of labels. This means that, in addition to adjacency properties, vertices and edges sharing a label in one graph must also share a label in the other graph (Hsieh et al., 2006).

The latter definition of isomorphism applies to the graph representations of CityGML documents employed in this thesis, since they are directed, labelled, and attributed graphs that contain multiple equivalence classes of labels derived from various CityGML classes.

4.1.3. The Graph and Subgraph Isomorphism Problem

The **graph isomorphism problem** is the computational task of determining whether there exists an isomorphism between two given graphs. This problem has been a focus of research in multiple disciplines, including mathematics, theoretical computer science, structural chemistry, and Geographic Information System (GIS) with hundreds of published studies dedicated to it. Despite its prominence, the graph isomorphism problem remains one of the few major algorithmic problems, besides integer factorization, for which the exact computational complexity is still unknown (Skiena, 2008).

In computational complexity theory, decision problems (yielding 'yes' or 'no'), belong to the complexity class P (polynomial time complexity) if they can be solved using a deterministic Turing machine (Turing, 1936) and their worst-case runtime is polynomial with respect to their input size n, i.e., in $\mathcal{O}\left(n^k\right)$, for a certain positive constant k. A decision problem is in the complexity class NP (non-deterministic polynomial-time) if it can be solved in polynomial time using a non-deterministic Turing machine and verified in polynomial time using a deterministic Turing machine. A decision problem is NP-complete (non-deterministic polynomial-time complete) if it is in NP and every other problem in NP can be reduced to this problem in polynomial time. While a solution to an NP-complete problem can be quickly verified (Cobham, 1965), finding it quickly is much more challenging. Thus, NP-complete problems are the most difficult problems for which a solution can still be verified quickly. Notable NP-complete problems include the Boolean satisfiability problem (SAT) (Cook, 1971), the Hamiltonian path problem (Garey & Johnson, 1979), and the travelling salesman problem (Held & Karp, 1970).

The graph isomorphism problem is unique because it is not known to be solvable in polynomial time nor to be NP-complete (McKay & Piperno, 2014; Skiena, 2008). This is under the assumption that P is not equal to NP, implying that P and NP-complete are disjoint subsets of NP. The majority of studies consider the runtime complexity of the graph isomorphism problem to reside somewhere between P and NP-complete (Skiena, 2008). The most efficient solution for the graph isomorphism problem is currently believed to be in quasi-polynomial time $\exp\left(\log^{\mathcal{O}(1)} n\right)$ (Babai, 2016). The previously best known bound was $\exp\left(\mathcal{O}\left(\sqrt{n \log n}\right)\right)$ (Babai & Luks, 1983).

A large number of studies focused on solving the graph isomorphism problem for unlabelled graphs (Conte et al., 2004; Hsieh et al., 2006). To verify the isomorphism between two labelled graphs, many studies performed the graph canonization (or graph canonicalization) to convert these graphs into a *canonical form* (Huan et al., 2003; Kuramochi & Karypis, 2004). The canonical form of a labelled graph serves as its identifier, establishing that two graphs are isomorphic to each other if and only if they have the same canonical form (Hsieh et al., 2006). For a labelled graph of n vertices, where each vertex is uniquely identifiable, the time complexity to derive its canonical form is $\mathcal{O}(n!)$, accounting for all permutations over its vertices. This essentially renders the approach impractical even with a small vertex set. In the case of a graph where the same vertex or edge label may recur multiple times, *vertex invariants*, which are vertex properties that remain unchanged across isomorphism mappings (Kuramochi & Karypis, 2004), such as vertex degrees or labels in this context, are utilized. This allows the vertices to be segregated into equivalence classes $\pi_1, \pi_2, \ldots, \pi_c$. The time complexity to derive the canonical form is $\Theta\left(n^2 \prod_{i=1}^{c} (|\pi_i|!)\right)$, where $|\pi_i|$ represents the number of vertices of class π_i (Hsieh et al., 2006; Kuramochi & Karypis, 2004). This can be further reduced to $\mathcal{O}\left(n \sum_{i=1}^{c} (|\pi_i|!)\right)$ by encoding the vertex invariants into compact vertex signatures, which can be compared in linear time (Hsieh et al., 2006).

Given the complexity of the graph isomorphism problem, a majority of successful research has concentrated on partitioning vertex sets based on specific criteria and invariants (Arlazarov et al., 1974; Corneil & Gotlieb, 1970). This approach is often referred to as the 'individualization-refinement' paradigm (McKay & Piperno, 2014). One of the most prominent implementations of this approach is *nauty*, which leverages **automorphism**, an isomorphism of a graph to itself, to organize the graphs into automorphism groups and apply canonical labelling for search pruning (McKay, 1978, 1980). Decades later, it was demonstrated that *nauty* could generate large automorphism groups for certain types of graphs, causing exponential runtime (Miyazaki, 1997). This led to the introduction of a new software called *Traces* (Piperno, 2011). Unlike *nauty*, *Traces* does not rely on backtracking to navigate within the search space. Instead, it employs a strategy that combines the benefits of both the breadth-first traversal for early pruning of irrelevant subtrees and the depth-first search for detection of automorphism. Currently, both *nauty* and *Traces* are available in a single software package (McKay & Piperno, 2023). While *nauty* supports labelled vertices, it does not support labelled edges (Hsieh et al., 2006; Kuramochi & Karypis, 2004; McKay & Piperno, 2023). However, certain types of graphs with labelled edges can be converted to graphs with labelled vertices (McKay & Piperno, 2023). On the other hand, *Traces* does not support directed graphs (McKay & Piperno, 2014; Piperno, 2011).

To further reduce the runtime complexity of the graph isomorphism problem, additional constraints can be imposed on the graphs. For instance, polynomial time can be achieved for specific types of graphs, such as interval and planar graphs (Colbourn & Booth, 1981; Hopcroft & Wong, 1974), both labelled or unlabelled trees (Aho & Hopcroft, 1974), and graphs with excluded minors (Grohe, 2010; Ponomarenko, 1988). Further classes of graphs that also achieve this polynomial time when certain constraints are applied include bounded genus (Filotti & Mayer, 1980; Miller, 1980), bounded maximum vertex degree (Luks, 1982), bounded tree-width (Bodlaender, 1990), bounded eigenvalue multiplicity (Babai et al., 1982), bounded rank-width (Grohe & Schweitzer, 2015), and unit square graphs (Neuen, 2016).

The **subgraph isomorphism problem** is a more complex task closely related to the graph isomorphism problem. It involves determining whether there exists a subgraph of a given graph G that is isomorphic to another given graph H. Thus, the subgraph isomorphism problem is a generalization of the graph isomorphism problem, as G and H are isomorphic if and only if they are isomorphic subgraphs of each other. However, unlike the graph isomorphism problem, whose runtime complexity is still undetermined between P and NP-complete, the runtime complexity of the subgraph isomorphism problem is known to be NP-complete (Ullmann, 1976). This is because the subgraph isomorphism problem is a generalization of other NP-complete problems, such as the Boolean satisfiability problem (SAT) (Cook, 1971), the clique problem (Bomze et al., 1999), and the Hamiltonian path problem (Garey & Johnson, 1979). However, polynomial time can be achieved for certain types of graphs, such as planar graphs (Eppstein, 1999).

Applications of both the graph and subgraph isomorphism problem cover a wide range of use cases in various fields, such as exact graph matching in computer vision and pattern recognition, and identification of known chemical compounds in mathematical chemistry and cheminformatics. This research employs a heuristic variant of graph matching that has been optimized for graph representations of CityGML documents, which will be explained throughout this chapter. The runtime complexity of this matching process, as demonstrated later in Chapter 7, exhibits a **linear growth rate** with respect to dataset sizes.

4.2. Advantages and Challenges of Using CityGML Graphs

The main objective of this chapter is described as follows: Given two graph representations of a CityGML document, compare both their structure and content, and identify all deviations and their locations within the graphs. However, the unique characteristics of these graph representations, as generated from Chapter 3, present both advantages and challenges. This section addresses Research Question RQB3 (Advantages and Challenges of CityGML Graphs). The advantages of using the graph representations of CityGML documents for the matching process include:

1. **Semantic Graphs**: These graphs are semantically rich with node and relationship labelling. The node labels and relationship types are not unique and may recur multiple times, such as in child nodes of the same parent node in a one-to-many relationship, or the same node label but at different locations within the graph like the node label *gml:Polygon* that occurs at different levels as shown in Table 3.2. This allows the graphs to be divided into several equivalence classes of recurring node labels and relationship types, simplifying the matching process.

2. **Well-defined Data Model**: The graph entities are not only labelled, but the relationships that should follow a node and vice versa are dictated by the underlying GML and CityGML data model. This semantic context serves as a guide, where node labels and relationship types are already predefined. These semantic rules are the additional constraints imposed on the graphs, allowing for fast retrieval of matching candidates that satisfy these rules and dismissing those that do not.

3. **Object-oriented Structure**: The graph representations of CityGML objects closely mirror their object-oriented representations. This means that a subgraph can be easily transformed back into its original in-memory object, facilitating the matching of their structure and content. For example, a polygon object can be rebuilt from a polygon node and its associated subgraph, as explained in Section 3.6, allowing for matching its geometric extents without the constant need to fetch the data from each node and relationship manually. However, this approach is only applicable for small subgraphs.

4. **Spatial Extent**: The employed graphs are not only rich in semantic content, but also in spatial information, such as the available or computed bounding box of each building in the city model. This spatial data is crucial to limit the number of candidates that could potentially match a reference subgraph. For example, when given a reference building, the matching process avoids a brute-force search for matching candidates by specifically looking for those that have the most overlapping bounding volume with that building.

However, the use of the graph representations of CityGML documents in this research also presents many challenges:

1. **Complex Class Hierarchy**: The CityGML data model utilizes a multitude of classes to represent city objects. These classes are organized within a complex class hierarchy. This translates into a high number of node labels and relationship types employed within the graph database. However, since Neo4j operates as an instance-based database, it lacks the ability to handle many object-oriented concepts, such as inheritance and polymorphism. For instance, by employing the graph data model explained in Section 3.2, each node only has one label for its type, which is the lowest subclass in the class hierarchy of its respective CityGML object. As a result, the database is unable to determine whether a node represents an instance of a certain superclass. This restriction in the number of labels allowed per node is made to avoid excessive number of indexes in the database, which not only requires more storage but also slows down the processing time. Therefore, such type checking is typically performed outside of the database using an object-oriented programming language such as Java.

2. **Large, Directed Graphs**: The graphs used in this research are directed and therefore not compatible with the tool *Traces* (Piperno, 2011). Additionally, since the relationships are typed, the graphs cannot be processed by the tool *nauty* without first transforming them to graphs with only labelled nodes (McKay & Piperno, 2023). Furthermore, these graphs are often very large in size, with millions of nodes and gigabytes of data stored in the graph database. Graph and subgraph isomorphism are generally not suited for large graphs (McKay & Piperno, 2014).

3. **Attributed Graphs**: Each node and relationship of the graphs is attributed. In Neo4j, while properties within a node or a relationship are uniquely identified, there may exist properties across multiple nodes or relationships with identical names but potentially different meanings. Thus, matching a node or a relationship based solely on their labelling is not sufficient; their properties must also be taken into account.

4. **Alternating between Graph and Subgraph Isomorphism**: The change detection process in this context is not a decision problem, meaning it does not yield a simple *true* or *false*. Instead, when two subgraphs representing the same object are identified, they are compared, and the exact extent and location of their deviations in the graphs must be reported. Moreover, graph isomorphism is typically employed in cases where both the graphs have the same number of vertices. However, this is not always applicable in the graph representations of CityGML documents. Therefore, the more complex subgraph isomorphism, or inexact graph matching, is a more suitable approach. As a result, graph isomorphism is applied first to determine whether two graphs are identical. If not, the process switches to subgraph isomorphism and attempts to find a matching subgraph.

In addition, challenges, which are not strictly specific to graphs, but are rather the direct consequences of the underlying encoding standards of GML and CityGML, include: *order and identifier-independence*, *syntactic ambiguities*, and *uncertainties in detecting geometric changes of objects*, such as translations and size changes of surfaces with measurement errors taken into account. These challenges are described previously in Section 2.2.

Therefore, all these factors add a significant layer of complexity to the already challenging graph and subgraph isomorphism problem employed in this chapter. These shall be addressed in Section 4.3.

4.3. Methods for Comparing CityGML Graphs

The main method for detecting changes between two graph representations of CityGML documents is outlined in Algorithm 6. This method *compare(left, right)* utilizes the graphs generated by the mapping methods introduced in Chapter 3. However, graphs with similar structure and data model can also be applied. The techniques and strategies used in Algorithm 6 are explained in the following sections, addressing the first half of Research Question RQB4 (Matching Methods).

4.3.1. Recursive Matching

The method *compare(left, right)* is recursive (refer to Line 29) and operates in a top-down manner. Its two input nodes, *left* and *right*, are node representation of the old and new CityGML object, respectively. These are the source nodes of the subgraph representations of CityGML objects, from which all remaining nodes in the subgraphs can be reached. The method first compares the properties of these nodes, then proceeds to their child nodes, and so forth until all elements of the subgraphs reachable from the given nodes *left* and *right* have been traversed. The advantage of this approach is its concise implementation while remaining applicable to graphs of any scale, given sufficient computational resources.

The graph representations of CityGML objects utilized in this research are directed and weakly connected, as explained in Section 3.1.4. As a result, once this recursive method is invoked, it will be executed for the entire content of the CityGML objects represented by the provided nodes *left* and *right*. Moreover, as these graphs do not contain directed cycles, as explained in Sections 3.1.4 and 3.2.1, the termination of the recursive method on these finite graphs is guaranteed.

4.3.2. Semantic Matching

As mentioned in Sections 4.1.3 and 4.2, the challenges of graph and subgraph isomorphism are significant, particularly when dealing with graph representations of large-scale CityGML documents. However, the runtime complexity can be reduced significantly by utilizing the rich semantic information available in these graphs. This includes node labels, relationship types, and property names.

Therefore, a comparison of two nodes, relationships, or properties is only performed if they share the same labels, types, or names, respectively (refer to Lines 1, 8, and 20). This use of semantic labelling ensures that most of the graphs can be excluded when matching. For massive graphs, *thematic indexing* is utilized to enhance the retrieval of nodes or relationships based on their labelling, as detailed in Section 6.2.

Algorithm 6: Recursive graph comparison method *compare(left, right)*

```
Input   : Node representations left and right of old and new CityGML object
Outcome: Change nodes directly attached to the graphs when detected
if left.getLabels() != right.getLabels() then return
foreach property prop of left and not right do
    attach a Change node indicating removal of prop from left
end
foreach property prop of right and not left do
    attach a Change node indicating insertion of prop into right
end
foreach property prop in both left and right do
    if left.get(prop) $\neq$ right.get(prop) then
        attach a Change node indicating updated prop between left and right
    end
end
foreach outgoing relationship rel from left and not right do
    attach a Change node indicating removal of rel.getEndNode() from left
end
foreach outgoing relationship rel from right and not left do
    attach a Change node node indicating insertion of rel.getEndNode() into right
end
matchedRight $\leftarrow$ empty list of matched child nodes of right
foreach outgoing relationship rel from both left and right do
    leftChild $\leftarrow$ rel.getEndNode()
    $\{rightChild, similarity\} \leftarrow$ findBest(left, right, leftChild)
    if rightChild = null then
        attach a Change node indicating removal of leftChild from left
    else
        matchedRight.add(rightChild)
        if similarity $\neq$ EQUIVALENCE then
            attach a Change node indicating similarity between left and right
            compare(leftChild, rightChild)
        end
    end
end
foreach child node rightChild of right not found in matchedRight do
    attach a Change node indicating insertion of rightChild into right
end
```

4.3.3. Comparing Node Properties

When two nodes share the same label or two relationships share the same type, the matching process proceeds to compare their properties (refer to Line 2 through Line 12). In Neo4j, both node and relationship properties are stored in a key-value mapping, with property names serving as unique keys. This means that while the same property names can be reused across nodes and relationships, no two properties within the same node or relationship can have the same name.

As a result, retrieving properties by their names between two nodes or relationships is a straightforward process. If a property only exists in the node representation of the older CityGML object, it may indicate that the property has been deleted during the given time period (refer to Line 2 through Line 4). Conversely, if a property only exists in the node representation of the newer CityGML object, it may suggest that the property has been added during that period (refer to Line 5 through Line 7). A property is considered to have been updated from the old to new value only when it exists in both given nodes (refer to Line 8 through Line 12).

The comparison of these property values, as shown in Line 9, is complex, as it involves not only a type comparison but also the consideration of error tolerances when dealing with measurement values.

Handling Property Types

Since Neo4j does not store any type information for node and relationship properties in its instance, all property values retrieved from the database are typically treated as strings. Two solutions to this issue exist: (1) use the enriched type information of each property, as introduced in Section 3.6 for the reconstruction of graphs back into their original CityGML objects, or (2) perform a type conversion from string at runtime. The exact type information is crucial for matching, as these string property values can represent a variety of data types, including booleans, numbers, date-time values, and plain texts.

The first option, using the enriched type information, can be applied if such information is readily available in the graphs. If not, the second option can be employed. The process of deducing the corresponding data types from a given property text, denoted as t, is described as follows:

1. Trim all spaces at the beginning and end of t, and then convert t to lower case.
2. If t is equal to *true* or *false*, then it is identified as a boolean.
3. If t contains only digits and optionally a sign, then it represents a natural number.

4. If t contains only digits and optionally a sign, an exponent symbol 'e', or a single dot or comma, then it may represent a floating-point number.
5. If t contains only digits and date-time delimiters, then it may represent a date-time value.
6. If none of the above applies, t is treated as a plain text.

To implement these steps, *regular expressions* can be utilized.

Numeric Comparison

If the type of a given property value is determined as a natural or floating-point number, its string value is converted to a corresponding numeric format. Two natural numbers are considered identical if and only if their subtraction is equal to 0. On the other hand, two real numbers, denoted as p and q, are considered to be equal if and only if the absolute value of their difference is less than a small, acceptable error tolerance, denoted as ϵ. This can be described as:

$$p \approx q \iff |p - q| < \epsilon \tag{4.2}$$

Measurement Comparison

Each measurement in CityGML is composed of two components: a real value, and an associated unit of measurement (uom). Both of these are stored as separate node properties within the same node, as explained in Section 3.3.6. Therefore, when comparing two measurements, it is crucial to consider not only the measured values but also their units. Moreover, the length error tolerance itself also has a unit that may differ from those used in the measurements.

As a result, to compare two measurements, they must first be in the same unit. If not, they are converted to a common reference unit, typically metres. The length error tolerance is also converted if it is not in the reference unit. Once a common unit of measurement is established, the measured values can be compared in the same manner as when comparing two numeric values, as previously explained.

For instance, consider two measurements: 1 m and 99 cm. These are considered equal for a length error tolerance ϵ of 5 cm. This is because, when converted to metres, the two measurements are 1 m and 0.99 m, respectively. Their absolute difference, 0.01 m, is smaller than and the length error tolerance in metres, 0.05 m. Therefore, even though both the measurement values and units of measurements differ, they are considered identical, and no real change is recorded.

Date-Time Comparison

Similarly to measurements, date-time values also have units. However, unlike length measurements, date-time values can have multiple units at the same time: years, months, days, hours, minutes, seconds, and even milliseconds, along with the time zone. There exist numerous formats of describing the same date and time across many different regions of the world. For instance, dates can be given as *mm-dd-yyyy*, *mm/dd/yyyy*, *dd-mm-yyyy*, or *dd/mm/yyyy*. Similarly, the same time value can be described differently depending on the time zones.

To ensure a consistent and unambiguous way of describing date-time values worldwide, the international standard ISO 8601 provides a standardized format for defining and displaying these values (ISO, 2019a, 2019b). For example, the format *YYYY-MM-DD* is defined for calendar dates, such as '1970-01-01', and *hh:mm:ss* for time in a 24-hour clock system, like '01:00:00'. This time value can be directly appended with a time zone designator, such as *01:00:00Z* in Coordinated Universal Time (UTC). The character 'Z' stands for Zulu time, which corresponds to the time zone 0 of the military time zones (CCEB, 2010). Alternatively, a time zone offset can replace the letter *Z*, such as '+01:00' for time zone 1. In the absence of time zone information, the time is assumed to be local. An example of a combined date-time representation with a time zone could be '1970-01-01T01:00:00+01:00', which is equivalent to '1970-01-01T00:00:00Z'.

To handle these various syntactic representations of date-time values, Unix time, also known as epoch time, is commonly employed. It converts any given date-time value into the number of seconds, or milliseconds for greater precision, that have passed since 00:00:00 UTC, January 1, 1970, excluding leap seconds. Therefore, to compare two date-time values, they are first converted to Unix time. Then, the resulting values are compared in a similar manner to length measurements, with the unit being seconds (or milliseconds, depending on the level of granularity chosen).

The CityGML encoding utilizes the XML data type *xs:date* for defining date attributes, such as *creationDate*, where *xs* denotes the namespace of the XML schema. This data type leverages the ISO 8601 standard to describe a set of Gregorian calendar dates, with a period of one day between two nearest dates. For instance, the following date values are considered equivalent in XML: '1970-01-01', '1970-01-01Z', '1970-01-01+01:00', given a time error tolerance of one day. However, when CityGML documents are parsed using the library *citygml4j* to generate their in-memory objects, these dates are automatically appended with a default time value, such as with the suffix 'T00:00+01:00'. The combined date-time format is then stored as a single node property in the graph representations employed in this research.

Boolean and String Comparison

If none of the above conditions applies, the property values are simply considered as strings. Two strings are identical if they contain the same sequence of characters. The comparison of string values is case-sensitive.

On the other hand, boolean values are equal if they both evaluate to either *true* or *false*. This can be achieved by performing a logical comparison on their boolean values or by matching their string representations, irrespective of case. For instance, the string 'True' is considered equal to 'true'. This implementation can be extended to also allow for the string values of 'T' and 'F', or 't' and 'f'.

4.3.4. Matching Child Nodes and Subgraphs

In the employed graph database Neo4j, each node can have an arbitrary number of relationships, both in the incoming and outgoing direction. However, every relationship must be assigned with exactly one type. Moreover, a node can have multiple relationships of the same type. The matching process, which operates in a top-down manner, consistently follows the outgoing direction of all relationships to minimize repeated traversal. Thus, in the context of one-to-many and many-to-many semantic relationships, a node can establish multiple outgoing relationships of the same type to its child nodes.

If a relationship type only exists for the node representation of the older CityGML object, it may indicate that all corresponding relationships and their subsequent paths have been removed during the given time period (refer to Line 13 through Line 15). Conversely, if a relationship type is exclusive to the node representation of the newer CityGML object, it may suggest that all its relationships and their subsequent paths have been added during that period (refer to Line 16 through Line 18). A relationship type that exists for both nodes indicates that its relationships and associated subgraphs can be further compared (refer to Line 19 through Line 35).

Matching one-to-one relationships is straightforward, as it requires at most only a single comparison for the end node of each relationship. For example, if both *Building* nodes contain a *measuredHeight* relationship, their measured heights can be matched quickly, since each building can have only one measured height. However, when dealing with one-to-many and many-to-many relationships, more efficient strategies are required to accurately identify the best matching candidate for a reference node. For instance, when comparing an older CityGML document containing n buildings with a newer CityGML document containing m buildings, a brute-force comparison would result in nm comparisons between each possible pair of buildings. This approach is not only inefficient but also impractical for large datasets.

Therefore, an additional method *findBest(left, right, ref)* is employed to identify the best potential match for a reference node (refer to Line 22). In the example above of comparing two CityGML documents, this method returns at most a single building from the new city model that best matches the reference building from the old city model. If no match is found, the reference object is considered to have been deleted from the old dataset (refer to Line 24). If a match is found, both the reference node and the returned node are compared (refer to Line 29), significantly reducing the matching runtime by performing only one comparison. This method for finding the best potential match, which will be further explained in Section 4.5, often serves as a **preliminary evaluation method** to find the best matching candidate before any actual comparison. The similarity levels between the reference node and the matching candidates are evaluated for this purpose. These similarities are further explained in Section 4.4. Lastly, any remaining unmatched relationships from the new node are considered as additions to the new dataset (refer to Lines 19 and 26, as well as Line 33 through Line 35).

In contrast to node and relationship properties, where changes can be detected by comparing their literal content, solely comparing the structure of subgraph representations of CityGML objects may not always reveal actual changes. For example, even with identical geometric content, a polygon defined using a different syntactic way, such as by dividing its interior into two smaller adjacent interiors that together form the original interior, would result in a series of node deletions and insertions. Such syntactic ambiguities of geometric elements are addressed in Section 4.5.

4.3.5. Direct Attachment of Change Nodes to Graphs

During the method *compare(left, right)*, each detected change leads to the insertion of an additional node into the database (refer to Lines 3, 6, 10, 14, 17, 24, 28, and 34), serving as a change indicator. The type of this change determines whether its *change node* is attached to the *left* node representing the older CityGML object, the *right* node representing the newer CityGML object, or both. For example, for a deleted node property, a corresponding change node is attached to the *left* node, where the property was removed. Conversely, for an added node property, a corresponding change node is attached to the *right* node, where the property was inserted. In the case of an updated node property, the change node is attached to both the *left* and *right* node.

The benefits of attaching these change nodes are manifold. Firstly, it enables the marking and documentation of all detected changes, even after the matching process is complete, as these change nodes are persistently stored in the database. Secondly, a change node's direct links to its context nodes allow for quick extraction of relevant information required for further change analyses and classification, without the need for complex queries. Lastly, as explained in Chapter 5, change nodes and their linked nodes provide crucial semantic context in the graph database that allows for the interpretation of more complex change patterns.

In addition to change nodes, *match nodes* can also be used to indicate equality between two nodes. However, these change and match nodes require expensive write operations as they are written directly into the database. This constant writing can significantly slow down the matching process, potentially blocking other parallel processes or even causing a deadlock in a multi-threaded environment, as described in Section 6.4. For example, when enabled, comparing two similar graphs may result in a large number of match nodes, thereby slowing down the process. Therefore, in this research, considering that the graph representations of two temporal versions of the same CityGML document often exhibit more similarities than differences, in most cases, only change nodes are utilized and no match nodes are added.

The change nodes inserted in this context are often referred to as **edit nodes**, representing the edit operations created after the matching process between the two graphs is complete. This is further explained in Section 4.6.

4.4. Node and Subgraph Similarity

The method *findBest(left, right, ref)*, as shown in Line 22 of Algorithm 6, not only identifies the best potential match for a reference node, but also provides further information about the similarities in both their structural and semantic contents. In this study, the term **similarity** between two nodes refers to the similarity between them and the entire subgraphs they represent (i.e., the subgraphs that are reachable from these nodes when applying the 'downward' direction from higher semantic level elements to lower ones). For instance, the similarity between two polygon nodes is assessed by considering all their geometric properties extracted from their subgraph representations. The process of calculating these similarities is detailed in Section 4.5.

In the presence of a large number of potential matches, the similarities between the reference and each candidate are assessed first. Based on these similarity values, the matching process determines whether to consider both the reference and the corresponding candidate as identical, thereby eliminating the need for further comparison, or to proceed with the actual comparison of the content of both subgraphs. The metrics used in this process include both the semantic and geometric similarities, providing an order, based on which candidates can be evaluated and sorted.

In this research, the similarities between two node candidates for matching are categorized into several levels, each assigned with a unique label. A detailed description of these levels, given in ascending order, can be found in Table 4.1.

Table 4.1.: Similarity levels for matching nodes and subgraphs they represent, shown in ascending order.

Similarity Level	Scope	Description
SAME_LABELS	N	Both nodes share the same labels.
SAME_ID	N	Both nodes share the same identifier.
SAME_STRUCTURE	N G	Both nodes and their respective subgraphs share the same property names and graph structure.
SAME_GEOMETRY	N G	Both nodes and their respective subgraphs represent the same geometry.
EQUIVALENCE	N G	Both nodes and their respective subgraphs represent the same object.

N Applicable to nodes G Applicable to subgraphs represented by given nodes

While the majority of these similarity levels can be applied to both given nodes and the subgraphs they represent, some levels are specific to single nodes only. For example, the levels *SAME_LABELS* and *SAME_ID* indicate that both the given nodes share the same labels and identifier, respectively. These levels are exclusively employed when the nodes do not represent any further subgraphs.

On the other hand, similarity levels like *SAME_STRUCTURE*, *SAME_GEOMETRY*, and *EQUIVALENCE* apply to both nodes and their corresponding subgraphs. They are further explained as follows:

1. ***SAME_STRUCTURE***: This level suggests that both the given nodes and their respective subgraphs share the same naming schema for existing properties (without comparing property values), as well as the same graph structure. In this context, two graphs are considered to have the same structure if they are isomorphic to each other, with both vertex and edge labelling taken into account. This is typically the case for graph representations of different instances of the same object class: they have the same structure but contain different values.

2. ***SAME_GEOMETRY***: This level implies that even if two nodes and their subgraphs do not share the same graph structure, they can still represent the same geometry. This is due to the multiple syntactic possibilities allowed by the GML and CityGML encoding standard. This level does not include the equality of the thematic property values stored within the nodes and subgraphs, except for geometric coordinates and measurements used to represent the geometry.

3. ***EQUIVALENCE***: This level ensures that both nodes and their subgraphs represent the same object. This means that both nodes and their subgraphs are identical in both content and structure, or they may be structured differently but share the same content. This is the highest level of similarity, and no further comparison between the nodes and their subgraphs is needed once this level is reached.

The distinction between the levels *SAME_STRUCTURE* and *EQUIVALENCE* lies in the property values within the graphs being different in the former. Similarly, the level *SAME_GEOMETRY* differs from *EQUIVALENCE* in that it does not enforce the equality of the thematic properties stored within the graphs. Therefore, both the levels become *EQUIVALENCE* if either all respective thematic property values also match, or they do not contain any thematic properties.

For instance, by computing their orientations, positions, bounding boxes, and key vertices, two boundary surfaces can be quickly matched without processing their geometric content. Thus, they are first categorized as *SAME_GEOMETRY*, indicating that a geometric comparison can be omitted, and further comparison of their thematic elements may still be needed. However, if these surfaces do not have any thematic elements, they are considered equal and categorized as *EQUIVALENCE*.

The similarity levels introduced above are employed by the method *findBest(left, right, ref)* in Section 4.5 to provide the matching process with additional information about the found candidates, including how they were selected, and how similar they are to the given reference node. This information is then reflected in the additional change nodes directly attached to the graphs (refer to Line 28 of Algorithm 6).

While the similarity levels shown in Table 4.1 are crucial for the matching process in this research, additional, more granular levels can be introduced when needed.

4.5. Finding the Best Potential Match

As outlined in Line 22 of Algorithm 6, the method *compare(left, right)* utilizes *findBest(left, right, ref)* to identify the best potential match for a reference node. This serves as an **initial filtering mechanism**, allowing for fast selection of the most suitable match from a large candidate pool. This is achieved by evaluating their most prominent features that can be extracted quickly, thereby circumventing the need to parse and compare entire subgraphs. This process is outlined in Algorithm 7, addressing the last half of Research Questions RQB2 and RQB4, as well as Research Questions RQB5 to RQB8 (Graph and Subgraph Isomorphism, Matching Methods, Syntactic Ambiguities, Geometric Uncertainties, Geometric Transformations, and Finding Best Match, respectively). The following sections explain how the method *findBest(left, right, ref)* leverages the unique characteristics of various GML and CityGML objects, with an emphasis on geometries.

Algorithm 7: Method for finding the best potential match *findBest(left, right, ref)*

```
Input  : Node representations left and right of old and new CityGML object
         A child node ref of left as reference
Output: The best matching candidate match for ref, which is a child of right
         Similarity value between match and ref
candidates ← right.getChildren(ref.getLabels())
if candidates is empty then return {null, null}
switch object type represented by ref do
    case generic attribute do
        match ← candidates.matchGenericByValue(ref)
        if match = null then
            match ← candidates.matchGenericByName(ref)
            return {match, SAME_LABELS}
        end
        return {match, EQUIVALENCE}
    end
    case geometry do
        match ← candidates.matchGeometry(ref)
        if match = null then
            return {null, null}
        end
        if ref and match contain thematic properties then
            return {match, SAME_GEOMETRY}
        end
        return {match, EQUIVALENCE}
    end
    case top-level feature do
        match ← candidates.matchBBox(ref)
        if match = null then
            return {null, null}
        end
        return {match, SAME_LABELS}
    end
    otherwise do
        return {match, SAME_LABELS}
    end
end
```

4.5.1. Matching Generic Attributes

The generic module of CityGML enables the extension of its data model with arbitrary additional attributes. These generic attributes, declared in the class *CityObject* of the core module of CityGML, can be incorporated into all city objects (Gröger et al., 2012). Each generic attribute can have a *name* and a *value*. Six different types of generic attributes are available, defining this value as a natural number, a floating-point number, a date-time value, a measurement object (containing a measured value and its unit), a string value, and a Uniform Resource Identifier (URI).

A city object can have an arbitrary number of generic attributes, either directly as a collection of individual generic attributes or as a collection of generic attribute sets, which, in turn, are collections of individual generic attributes. As a result, when matching graph representations of two city objects with multiple generic attributes, the method for finding the best potential match for each reference generic attribute can be applied (refer to Line 4 through Line 11). This process is further described as follows:

1. **Value-based Matching**: The value of a reference generic attribute is compared to all candidates to find the optimal match (refer to Line 5). The comparison method varies depending on the types of the generic attributes:
   a) **Numeric Values**: The numeric values, both natural and floating-point numbers, are compared considering a small error tolerance.
   b) **Measurement Values**: Generic measurements are compared based on their values and units, factoring in unit conversion and a small error tolerance.
   c) **Date-time Values**: Generic date-time values are parsed and compared, accounting for different conventions for describing dates and time.
   d) **Text Values**: Generic text values, such as string and URI values, are compared based on their exact string representations.

   The techniques for comparing these different value types have been previously discussed in Section 4.3.3. A non-empty result from this matching process suggests an exact correspondence between two generic attributes, as indicated by the similarity level *EQUIVALENCE* shown in Line 10, eliminating the need for additional comparisons.

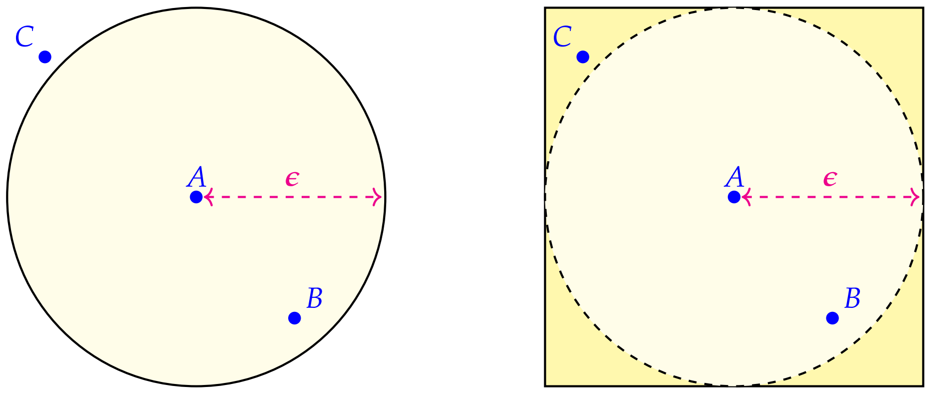
2. **Name-based Matching**: If the value-based matching above fails to identify a potential match for a given reference generic attribute, the process proceeds to find the next best match with the same name (refer to Lines 6 and 7). A non-empty result suggests that the two generic attributes share the same type and name, but with different values, as indicated by the similarity level *SAME_LABELS* shown in Line 8.

4.5.2. Matching Points

As fundamental geometric elements in 0D, points are pivotal in constructing more complex, higher-dimensional geometries. This results in a CityGML document often containing a vast number of points, with even a small building potentially employing hundreds of points to define its geometries. Therefore, the ability to quickly find a matching point from a large set is crucial for various subsequent processes.

Two points are considered equivalent if they are located within each other's neighbourhood, defined by their coordinates and a margin of error or error tolerance, denoted as ϵ. This neighbourhood forms a circle in 2D or a sphere in 3D space, with the point at the centre and ϵ as the radius. As illustrated in Figure 4.2a, point B lies within the neighbourhood of A, making them equivalent, while A and C are not.

In a Projected Coordinate System (PCS) with Cartesian coordinates like x and y, Euclidean distances can be employed to compute the planar distances between points. Conversely, in a Geographic Coordinate System (GCS) with spherical coordinates like longitude and altitude, the haversine formula is applicable for calculating the great-circle distances between points.



(a) Neighbourhood as a circle (b) Neighbourhood as a square

Figure 4.2.: Matching points by calculating their Euclidean distances (left) or direct comparison of their coordinates (right) in 2D, with an error tolerance ϵ. Points located in the same neighbourhood (yellow) are considered equal.

Alternatively, for a sufficiently small ϵ, individual point coordinates can be compared directly. As these are numerical values, the same comparison techniques used for numbers apply, as explained in Section 4.3.3. This approach defines the neighbourhood as a square in 2D and a cube in 3D, with the point at the centre and 2ϵ as the side length. As shown in Figure 4.2b, point C now also falls within the enlarged neighbourhood of A. The points A, B, and C are thus considered equivalent.

Compared to Euclidean distances or the haversine formula, which require expensive arithmetic operations like multiplications and square roots, or complex trigonometric operations and their inverses like sine, cosine, and arcsine, matching point coordinates only requires simple operations such as subtractions and comparisons, which are significantly faster. However, this approach is only applicable for a sufficiently small ϵ. The new neighbourhood shown in Figure 4.2b is larger than that in Figure 4.2a by $(4 - \pi)\,\epsilon^2$ in 2D or $(8 - 4\pi/3)\,\epsilon^3$ in 3D. With a very small value of ϵ, this difference is negligible in most cases.

In CityGML datasets consisting of millions of points, the task of retrieving and comparing points can be optimized by employing *point indexing*, a feature available in the graph database Neo4j. This point index is capable of determining quickly whether two points are in proximity or identifying all points near a given point. A detailed explanation of this indexing is provided in Section 6.3.1.

4.5.3. Matching Line Segments

While a curve can take the form of an arc or a straight line, in CityGML, curves are specifically restricted to straight lines. As a result, only the GML class *LineString* is utilized (Gröger et al., 2012). A *LineString* is a special type of curve that is defined by two or more points, with linear interpolation between them (Cox et al., 2004). Consequently, a *LineString* is a series of connected line segments, representing a polygonal chain. For example, as illustrated in Figure 4.3, the blue *LineString* geometry is composed of five points that define four connected segments AB, BC, CD, and DE.

Given that multiple *LineString* elements with collinear segments can represent the same geometry, such as the blue paths $ABCDE$ and $ABDE$ illustrated in Figure 4.3, the first step is to resolve this syntactic ambiguity. This is achieved by grouping all collinear points, with an error tolerance ϵ taken into account, to form larger segments confined by their two outermost points, often referred to as vertices. For example, the points B and D serve as the vertices of the collinear segments BC and CD. The line segments that result from this process are distinct and can therefore be further compared.

To find the best potential match for a given reference *LineString* geometry efficiently, the following steps can be performed:

1. **Bounding Box Calculation**: For each *LineString* candidate, its Axis-aligned Minimum Bounding Box (AABB) is calculated. A bounding box of a *LineString* is defined by the minimum and maximum coordinates of its points in each dimension. These bounding boxes are utilized for the following evaluations:
    a) **Overlap Check**: All candidates, where the overlapping area or volume of the bounding boxes exceeds a certain sufficiently high threshold, such as

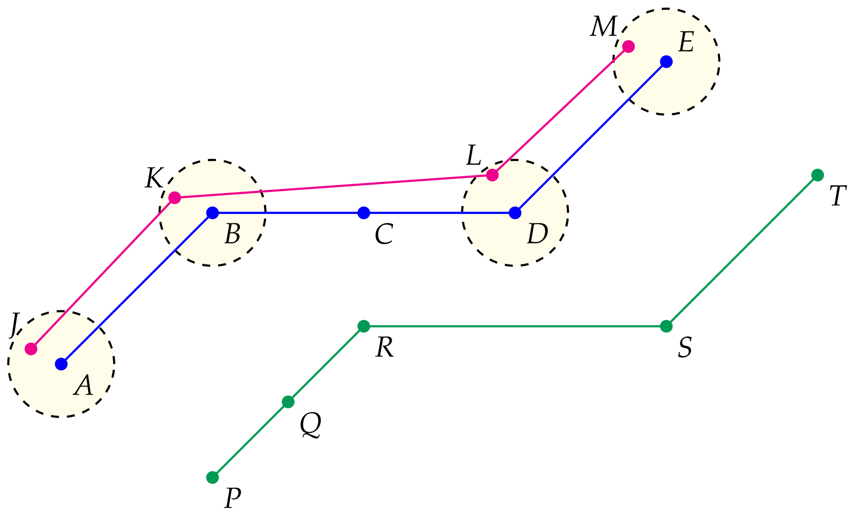


Figure 4.3.: An illustration of the process of matching line segments. Collinear points are grouped to form larger segments, bound by two furthest points, referred to as vertices. For example, the segments BD and PR represent the collinear points B, C, D and P, Q, R, respectively. Then, the path $JKLM$ is considered to be geometrically equivalent to the reference path $ABCDE$, as their vertices are located within each other's neighbourhood, represented by the yellow circle areas with the error tolerance ϵ as radius. The path $PQRST$ could also be considered equivalent, with translations by an acceptable amount taken into count.

90 % of the area or volume of the reference bounding box, are considered. If the reference *LineString* contains only collinear segments, resulting in its bounding box being also a line, the same principle can be applied but to its overlapping length instead of area or volume.

b) **Point-wise Comparison**: Among these, the best candidate is found where all its vertices are equivalent to those of the reference *LineString*, as explained in Section 4.5.2. For a small value of ϵ, this should yield at most one result.

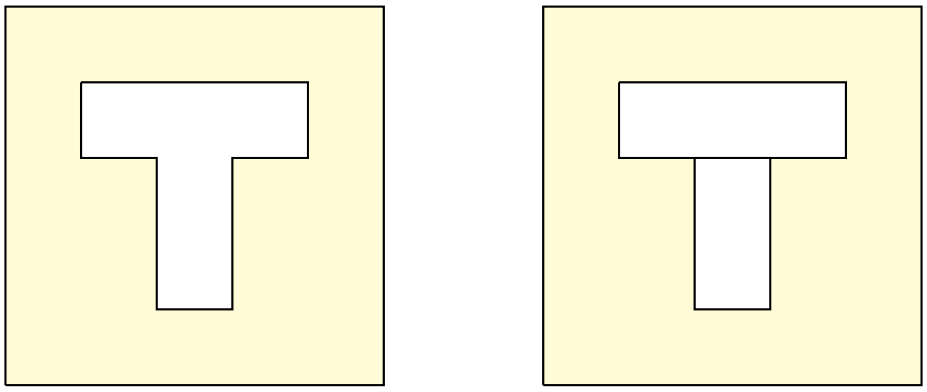
2. **Minimum Distance Check**: If no candidates are found with a sufficiently large overlapping area or volume, the candidate with the smallest distance to the reference bounding box is selected. If this candidate contains the same number of vertices and there exists a consistent translation vector v for each corresponding vertex pair between the two geometries, the best match is found. If not, the selection process continues with the candidate with the next smallest distance

until a match is found or none remains. When found, this match represents the same geometry as the reference but has been translated by v. As illustrated in Figure 4.3, the green path $PQRST$ represents the same line segments as $ABCDE$, but it has been translated by the distance $\|v\| = AP = BR = DS = ET$.

The computation of the axis-aligned bounding boxes is straightforward, as it only requires comparison between point coordinates to determine their minimum and maximum in each dimension. Moreover, the overlap check is simple, yet effective to filter out most unrelated candidates. The minimum distance check ensures that the best optimal match can be found when translations between the geometries are also taken into account. However, the maximum allowed value of this distance offset $\|v\|$ should be kept small to reduce the number of potential matches. Moreover, this approach matches line segments where points are given in the same sequence. For line segments in opposite order, the points are first reversed, and then the same method is applied.

4.5.4. Matching Surfaces

The CityGML encoding standard employs the GML class *Polygon* to represent its surfaces (Gröger et al., 2012). In GML, a polygon is composed of an exterior and any number of interiors (also known as holes), which together form its boundary (Cox et al., 2004). Each polygon is planar, implying that both its exterior and interiors must be on the same plane. The exteriors and interiors are defined as closed rings, most notably represented by the GML class *LineString*. Figure 4.4 illustrates an example of such polygon surfaces.



(a) One exterior, one interior (b) One exterior, two interiors

Figure 4.4.: An example of a polygon with one single interior (left) and two interiors (right). Both shapes are equal.

Considering a polygon may contain any number of interiors, the first step is to combine all adjacent interior rings into bigger ones. This is achieved by successively performing a union operation on every two adjacent coplanar interiors until they merge into one. The shape of the polygon is then determined by subtracting this combined interior from its exterior. However, due to the computational intensity of both the interior merging and the subtraction process, these operations are only executed when two polygons share the same exterior ring.

Polygons, despite being 2D geometries, are given in 3D space. This presents both challenges and advantages for the matching process. The challenge lies in the increased complexity of geometrical functions needed to handle these surfaces, as they can be parallel, intersecting, or neither when treated as bounded regions in space. However, the advantage is that most of these surfaces, including parallel ones, can often be quickly filtered by calculating their normal vectors[1]. Moreover, as many surfaces in CityGML are created to form a closed 3D volume, such as a solid representing the 3D shape of a building, their normal vectors vary for a large number of member surfaces.

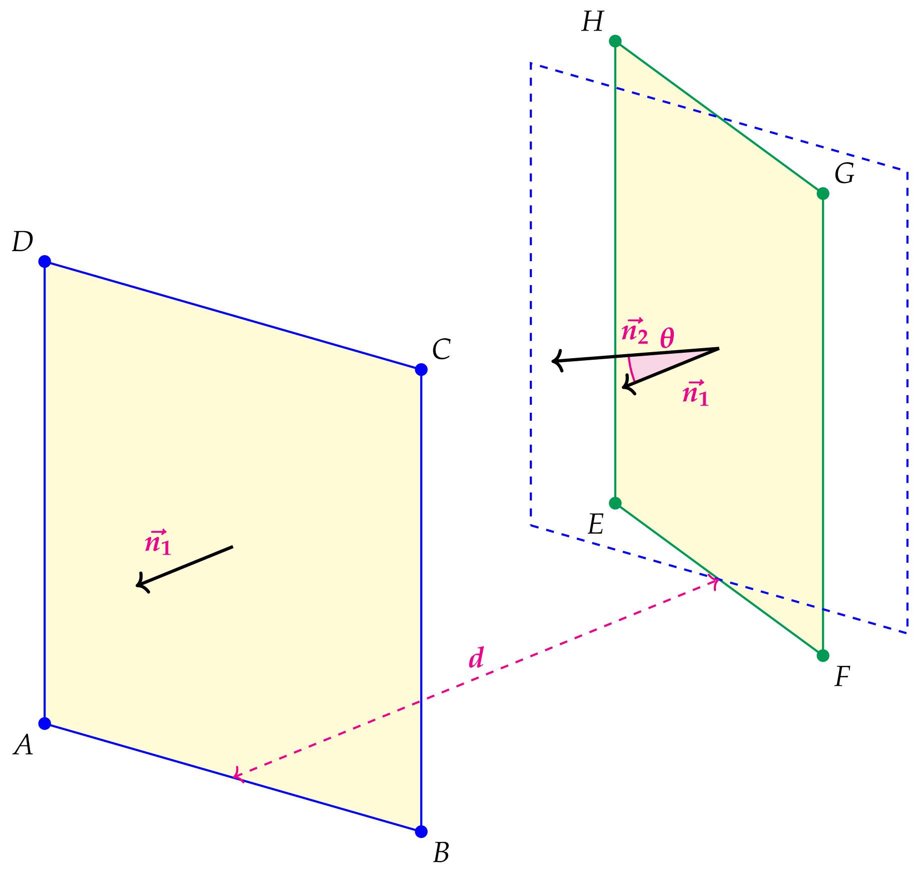
Numerous transformations can be applied between two surfaces, including affine transformations such as scaling, reflection, translation, and rotation. However, in the context of CityGML, where surfaces serve as both semantic and geometric representations of real-world objects, only certain transformations are considered. These include translation, rotation, and any combination of these transformations. These transformations belong to the class of rigid transformations, which preserve the Euclidean distances between any pair of points. Additionally, random variations in size or extent can occur between two surfaces. In this thesis, they are referred to as resize changes, which should not be confused with the resizing associated with affine transformations. Figure 4.5 provides an example of a translation and rotation between two surfaces.

To efficiently find the best potential match for a given reference polygon geometry, the following steps can be performed:

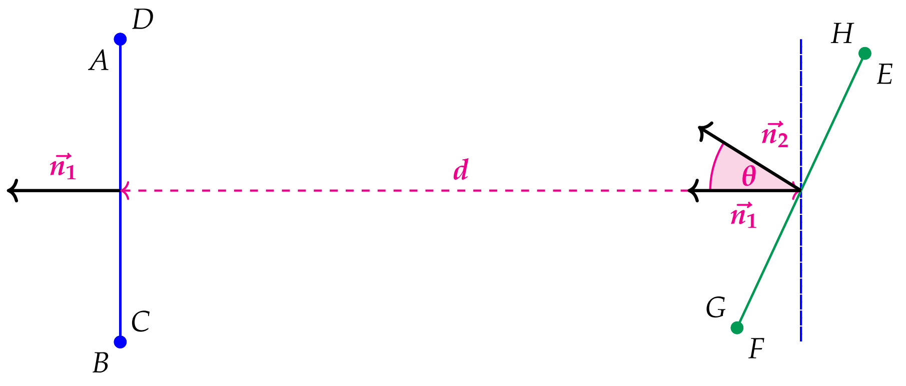
1. **Orientation Check**: The normal vector for each polygon candidate is computed. Its orientation is compared with that of the reference surface. Candidates with the same orientation, considering an angle error tolerance, are considered.

2. **Overlap Check**: Similarly to *LineString*, the Axis-aligned Minimum Bounding Box (AABB) of each polygon candidate is calculated. An overlap check is performed to determine the intersection between the candidates and the reference. Candidates with sufficiently high overlapping area or volume are considered.

[1]This study assumes that the input CityGML documents are both syntactically and geometrically valid. This implies that all points within a surface must be coplanar (with small error tolerances), and the order of points given in exterior and interior rings must be consistent.



(a) 3D view



(b) Top-down view (different scale)

Figure 4.5.: Matching two surfaces with a translation offset d and a rotation angle θ.

3. **Minimum Distance Check**: Among the remaining candidates, their centroids are calculated, whose coordinates are the mean values of the bounding box coordinates in each dimension. A distance between the centroid of each candidate and that of the reference is measured. This is to exclude parallel surfaces that have the same orientation. The candidate with the smallest distance is selected. Centroids, rather than corner points of the bounding boxes, are utilized to calculate distances to minimize the measurement's sensitivity against size changes in surfaces. The following steps are further performed:
   a) **Boundary Comparison**: The selected candidate with the minimum distance is considered a match for the reference surface. However, to account for translations indicated by non-zero minimum distance, the exterior and interior ring of the selected candidate are translated towards the reference by the found distance, and then compared with the exterior and interior ring of the reference polygon. This can be accomplished similarly to *LineString* comparison, as rings are equivalent to closed line segments. If the translated and the reference rings are equal, the selected candidate is considered a translated match for the reference polygon.
   b) **Size Check**: If the rings do not match, it implies a change in the surface size between the selected candidate and the reference, in addition to a potential translation indicated by the non-zero distance.
4. **Transformation Matrix**: If the overlap check fails but the orientation deviates from that of the reference surface within an acceptable range, a transformation matrix can be derived from the point set of the exterior ring of each candidate and the reference. This can be achieved by first translating one point set to the other using the distance between their centroids, followed by the application of the Kabsch-Umeyama algorithm (Kabsch, 1976; Umeyama, 1991). The resulting transformation matrix estimates an optimal rotation between the surfaces. However, this method does not consider surface size changes.

A major challenge in matching surfaces lies in determining not only their equivalence, but also potential transformations such as translation, rotation, resizing, or a combination thereof. Figure 4.5 gives an example where a matching polygon underwent both a translation and a rotation relative to the reference. The polygons $ABCD$ and $EFGH$ correspond to the same geometry, yet a translation and rotation exist. The translation offset d can be determined by calculating the centroids and their distance, while the rotation angle θ is measured between the normal vectors of the surfaces. Moreover, normal vectors can only be computed for valid surfaces, which is not always guaranteed in real-world datasets, where surfaces may consist solely of collinear points.

However, to minimize the number of potential matches and prevent false positives, the maximum allowed values for the distance offset d and the rotation angle θ should be kept small, such as 0.1 radian or 5°. This is particularly crucial, as polygon surfaces are often defined spatially adjacent or in close proximity to each other, collectively forming a more complex 3D form, such as the solid geometry of a building.

As adjacent interior rings within a surface are merged into a larger one, composite or triangulated surfaces are combined into a single, larger surface before matching. This method applies to geometric aggregates and composites in all dimensions.

4.5.5. Matching Solids

The CityGML encoding standard utilizes the *GML* class *Solid* as the basis for representing 3D geometric objects. Similarly to a polygon, a solid geometry is bounded by an exterior and any number of interiors. As the boundaries of a polygon are composed of closed 1D rings, the boundaries of a solid are formed by 2D surfaces.

Therefore, the first step is to merge all adjacent interior surfaces into a single larger one, enabling each solid to be distinctly compared based on their unique representation of exterior and interior boundaries. However, given that the combined interior cannot be expressed solely in 2D space, and the merging of interior surfaces is computationally intensive, it is generally sufficient to determine whether two solid objects represent the same geometry based solely on their exterior boundary.

Therefore, the matching process of solids is performed using the following steps:

1. **Overlap Check**: The 3D Axis-aligned Minimum Bounding Box (AABB) of each solid candidate is calculated. An overlap check is performed to determine the overlapping volume between the candidates and the reference. Candidates that exhibit a significant overlapping volume are considered.

2. **Minimum Distance Check**: Among the remaining candidates, their centroids are computed, whose coordinates are the mean values of the bounding box coordinates in each dimension. A distance between the centroid of each candidate and that of the reference is measured. The candidate with the smallest distance not exceeding a certain threshold is considered.

3. **Boundary Comparison**: The candidate selected in the previous step is considered a match for the reference. This step ensures the detection of any existing transformations. Since the exterior boundary of a solid is composed of adjacent surfaces, the surfaces of the solid candidate can be matched with those of the reference by employing the matching process for polygons, as described in Section 4.5.4. If a consistent transformation is found across all its boundary surfaces, this transformation is assumed to be applied to the selected candidate as well.

Figure 4.6 gives an example of two overlapping solid geometries with a potential translation. In contrast to *LineStrings* and polygons, where the geometric comparison between the optimal match and the reference is no longer required, the geometric comparison between a selected solid match and its reference still needs to be performed on their boundary surfaces, which employs the techniques covered in Section 4.5.4 for polygons.

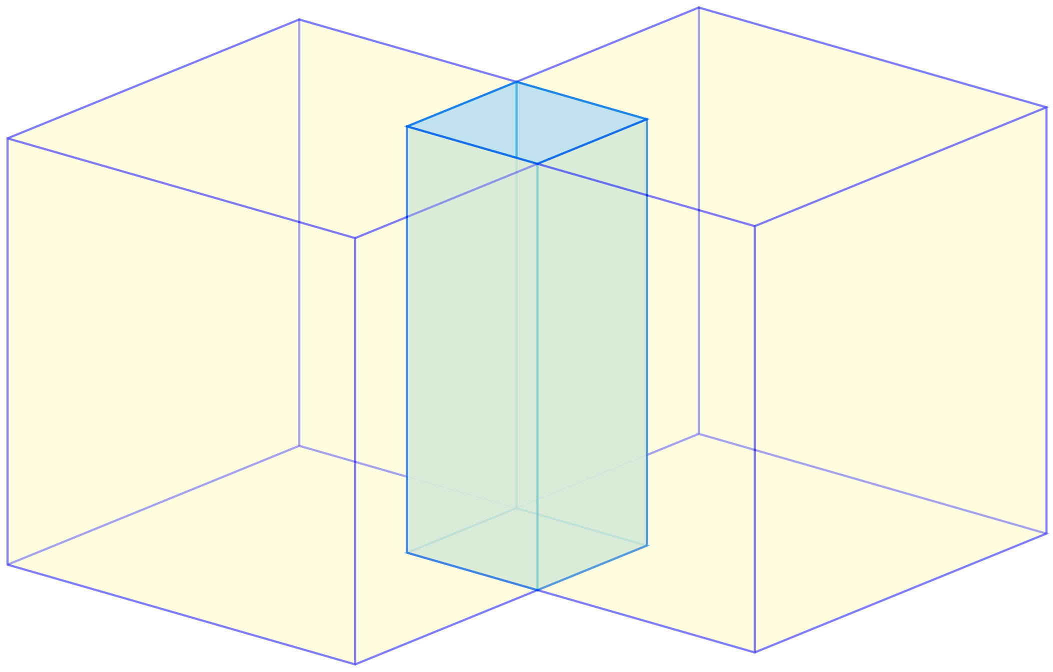


Figure 4.6.: An illustration of two overlapping solid geometries of identical size, potentially subjected to a translation. They are considered equivalent if their overlapping volume reaches that of each solid. If not, a translation is considered if the translation offset is within a certain threshold and no other solid is located in the vicinity.

4.5.6. Matching Buildings

Massive CityGML datasets may contain millions of top-level features, such as buildings. The brute-force approach of matching each possible pair of buildings from the old and new version of a city model can be computationally expensive or even infeasible. Thus, efficient strategies are crucial to first quickly determine the best optimal match for a reference top-level feature.

As outlined in Line 22 through Line 28 of Algorithm 7, bounding boxes of top-level features provide necessary spatial information for effective matching. For buildings, this thesis uses 2D footprints due for several reasons. Firstly, buildings are primarily distributed horizontally on the Earth's surface. Secondly, using building footprints also is advantageous when additional storeys are added. As the building height increases, its 3D bounding box expands vertically and may no longer match its original 3D bounding box. In contrast, its footprint remains unchanged and can be utilized for matching. However, this approach does not apply to horizontally expanded buildings with additional installations, which may be considered as separate buildings. For vertically allocated top-level features, like tunnels, a 3D bounding box overlap test is required for all matching candidates retrieved from the R-tree. This section focuses on matching buildings, with similar methods applied to other types of top-level features, as they share many thematic and geometric components.

As the employed bounding boxes are axis-aligned, it is common for a building and its footprint to overlap with the footprints of multiple nearby buildings. In such cases, similarly to matching geometries, the building with the largest overlapping area is considered the best match for a given building.

However, an optimal match only indicates that both it and the reference spatially correspond to the same building across the datasets. It does not guarantee the equality in their content and structure. Thus, the similarity level assigned to this match, as shown in Line 27, is *SAME_LABELS*, suggesting that further comparisons are required.

Additionally, candidates that do not satisfy the overlapping volume test may still correspond to the reference building. This is particularly the case where the reference building from the old dataset has been divided into smaller, adjacent ones in the new dataset, such as when a building has been split into two smaller, adjacent buildings. This is classified as an *ObjectSplit* change, as will be discussed in Section 5.2.

To detect such changes, the matching process examines whether the combined bounding box of two or more adjacent candidates is (mostly) equal to the bounding box of the reference building. If this condition is met, an *ObjectSplit* change is detected. Without this change, one *DeletedNode* change and multiple *InsertedNode* changes would have been created for the reference building and its candidates.

Conversely, to detect an *ObjectMerge* change, such as when two smaller buildings from the old city model are combined into a larger building in the new city model, the following steps are performed. For each element in the list of potential inserted buildings, a search is conducted among the elements in the list of potential deleted buildings. If the combined bounding box of two or more buildings from the second list is (mostly) equal to the bounding box of the building from the first list, an *Object-Merge* change is detected. Without this change, one *InsertedNode* change and multiple *DeletedNode* changes would have been created.

To handle massive datasets, an R-tree (Guttman, 1984) can be employed for the spatial indexing of these 2D footprints of buildings, enabling more efficient and rapid retrieval of their contents. Since the graph database Neo4j does not officially support spatial indexing of geometric elements beyond 0D points, an R-tree is incorporated on top of the existing graphs. The construction of this R-tree and the utilization of spatial indexing are described in Section 6.3.2.

4.5.7. Matching Other Types of Objects

The spatial extent of implicit geometries is first transformed based on their anchor points and then stored in an R-tree for matching. For objects that are neither generic attributes, geometric objects, nor top-level features, and lack distinctive features that can be utilized for efficient matching, the following rules, listed in descending order of priority, can be applied for matching one-to-many and many-to-many relationships:

1. **Key-value Matching**: Two nodes are considered matched, if they share the highest number of properties stored in the key-value mapping compared to other candidates.
2. **Identifier Check**: Despite the potential for an object's identifier to change across different temporal versions of CityGML documents, in the absence of other information useful for identifying the best optimal match, two nodes with the same identifier can be considered as a match for further comparisons.

If none of the above rules is applicable, the literal content and structure of nodes are compared. This measure is considered the last resort when all other criteria fail to apply. In such cases, it is crucial to assign the resulting match with the similarity level *SAME_LABELS*, as shown in Line 30 of Algorithm 7, to enforce further comparisons on its structure and content.

4.6. Edit Operations and Edit Nodes

As mentioned in Section 4.3.5, the method *compare(left, right)*, outlined in Algorithm 6, creates a change node each time a change in the content and structure of the graphs is detected (refer to Lines 3, 6, 10, 14, 17, 24, 28, and 34). These change nodes serve as markers, highlighting the locations of changes within the graphs. They not only allow for rapid retrieval of relevant information directly from the graphs, but also enrich the detected changes with semantic meaning required in further analyses, such as during the interpretation process of changes in Chapter 5. This section addresses Research Question RQB9 (Representation of Changes).

Change nodes, which are directly linked to their sources in the graphs where they were detected, are often referred to as *edit nodes* in this study. These nodes represent the concept of *edit operations*, which have been utilized in many change detection and update systems for structured data (Chawathe & Garcia-Molina, 1997), including graphs (Nguyen & Kolbe, 2020). This research categorizes the edit operations into five distinct classes based on their scope and linked data: *InsertedProperty*, *DeletedProperty*, *UpdatedProperty*, *InsertedNode*, and *DeletedNode*.

The proposed edit operations are described as follows:

1. **Property-based Edit Operations**: These edit operations are directly linked to nodes where property changes are detected.
   a) ***InsertedProperty*** $(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{R})$: This edit operation indicates the addition of a node property in the newer CityGML document. Each *InsertedProperty* instance contains information about the name p and value v of the inserted property, as well as the target 'right' node R, which is located in the graph representation of the newer CityGML document.
   b) ***DeletedProperty*** $(\boldsymbol{p}, \boldsymbol{L})$: This edit operation indicates the removal of a node property in the older CityGML document. Each *DeletedProperty* instance contains information about the name p of the inserted property, as well as the target 'left' node L, which is located in the graph representation of the older CityGML document.
   c) ***UpdatedProperty*** $(\boldsymbol{p}, \boldsymbol{v_L}, \boldsymbol{v_R}, \boldsymbol{L}, \boldsymbol{R})$: This edit operation indicates the modification of a node property in both the older and newer CityGML document. Each *UpdatedProperty* instance contains information about the name p, the old value v_L, and the new value v_R of the updated property, as well as the target 'left' node L and 'right' node R, which are located in the graph representation of the older and newer CityGML document, respectively.
2. **Node-based Edit Operations**: These edit operations are directly linked to nodes where changes on the node level are detected.
   a) ***InsertedNode*** $(\boldsymbol{C}, \boldsymbol{r}, \boldsymbol{R})$: This edit operation indicates the addition of a node in the newer CityGML document. Each *InsertedNode* instance contains information about the inserted child node C, the incoming relationship r associated with C, and the parent 'right' node R. All C, r, and R are located in the graph representation of the newer CityGML document.
   b) ***DeletedNode*** $(\boldsymbol{C}, \boldsymbol{L})$: This edit operation indicates the removal of a node in the older CityGML document. Each *DeletedNode* instance contains information about the deleted child node C, and its parent 'left' node L. Both C and L are located in the graph representation of the older CityGML document.

While the edit operations *InsertedProperty* and *DeletedProperty* are linked to only one corresponding node within the graph representation of either the older or newer CityGML documents, the edit operation *UpdatedProperty* provides information linked to both. On the other hand, the node-based edit operations *InsertedNode* and *DeletedNode* provide information about the affected node C, which can be located in either the older or the newer graph, depending on the type of the edit operations used, as described above.

Additionally, the elements C, L, and R serve as references to nodes within the graphs. This implies that the node-based edit operations also extend to subgraphs reachable from these nodes. For instance, when a building node is marked as inserted, an edit operation *InsertedNode* is attached to this node, indicating that it, along with all its reachable subgraphs, has been inserted.

The data stored in each type of edit operations described above contains the minimum amount of information required for tracking the locations and extracting the semantic context of changes efficiently. This can be extended to include further contents needed in specific use cases.

For each edit operation, its corresponding edit node is created in the graph database and filled with the following contents:

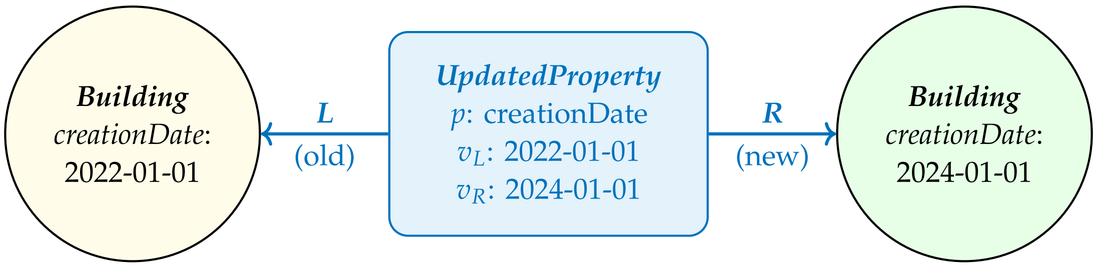
1. **Node Label**: The type of the edit operation is assigned as the label of the edit node.

2. **Node Properties**: The elements p, v, v_L, v_R, and r are stored as properties of the edit node.

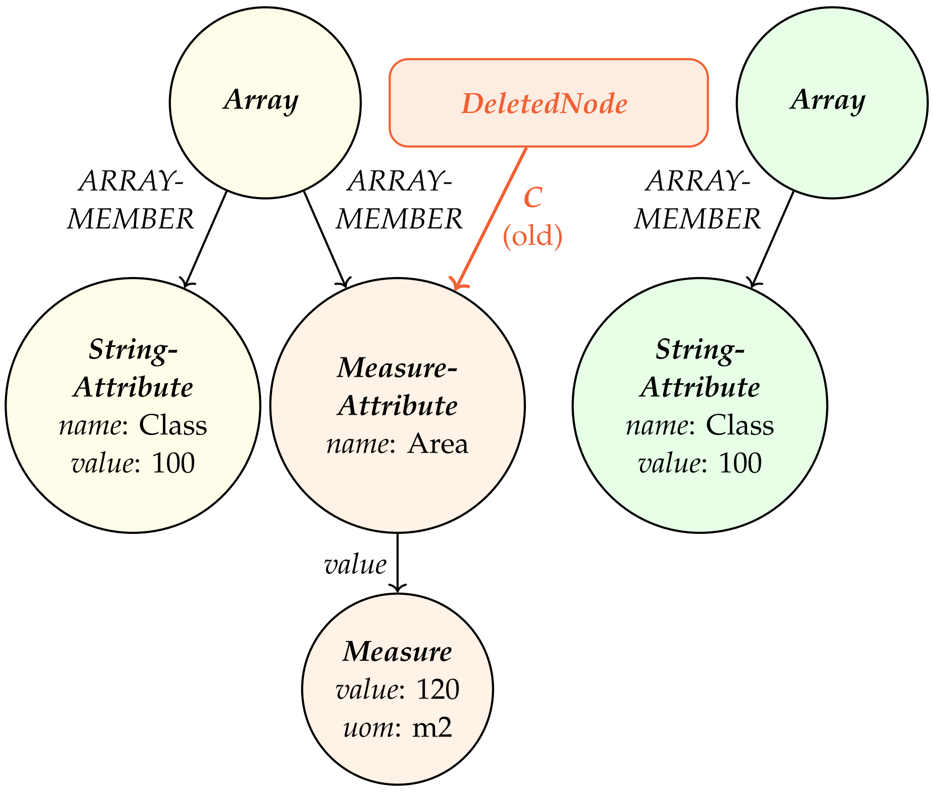
3. **Outgoing Relationships**: Corresponding outgoing relationships are created from the edit node to link it with the nodes L, R, and C.

Figure 4.7 gives two examples of these edit nodes. The upper edit node, as shown in Figure 4.7a, indicates an updated property between the old and new node representation of a building. The lower edit node, as illustrated in Figure 4.7b, depicts the removal of not only the node *MeasureAttribute*, but also its entire subgraph, including the node *Measure*. This subgraph represents a generic attribute for measurements in CityGML.

A corresponding UML class diagram of these edit operations can be found in Figure 4.8. The detailed property and method names, as well as their types are selected for clarity. The displayed class hierarchy is structured according to the scope of the edit operations, specifically focusing on property-based and node-based operations, as discussed above. Alternatively, these edit operations could be modelled based on their functions. This would prioritize the operation types deletion, insertion, and modification of both node properties and subgraphs over their scopes.



(a) Updated building property *creationDate*



(b) Deleted subgraph (orange) in an array of generic attributes

Figure 4.7.: Examples of edit nodes in graphs between the old (yellow) and new (green) graph representation of CityGML objects. The edit node *DeletedNode* indicates the removal of the orange subgraph.

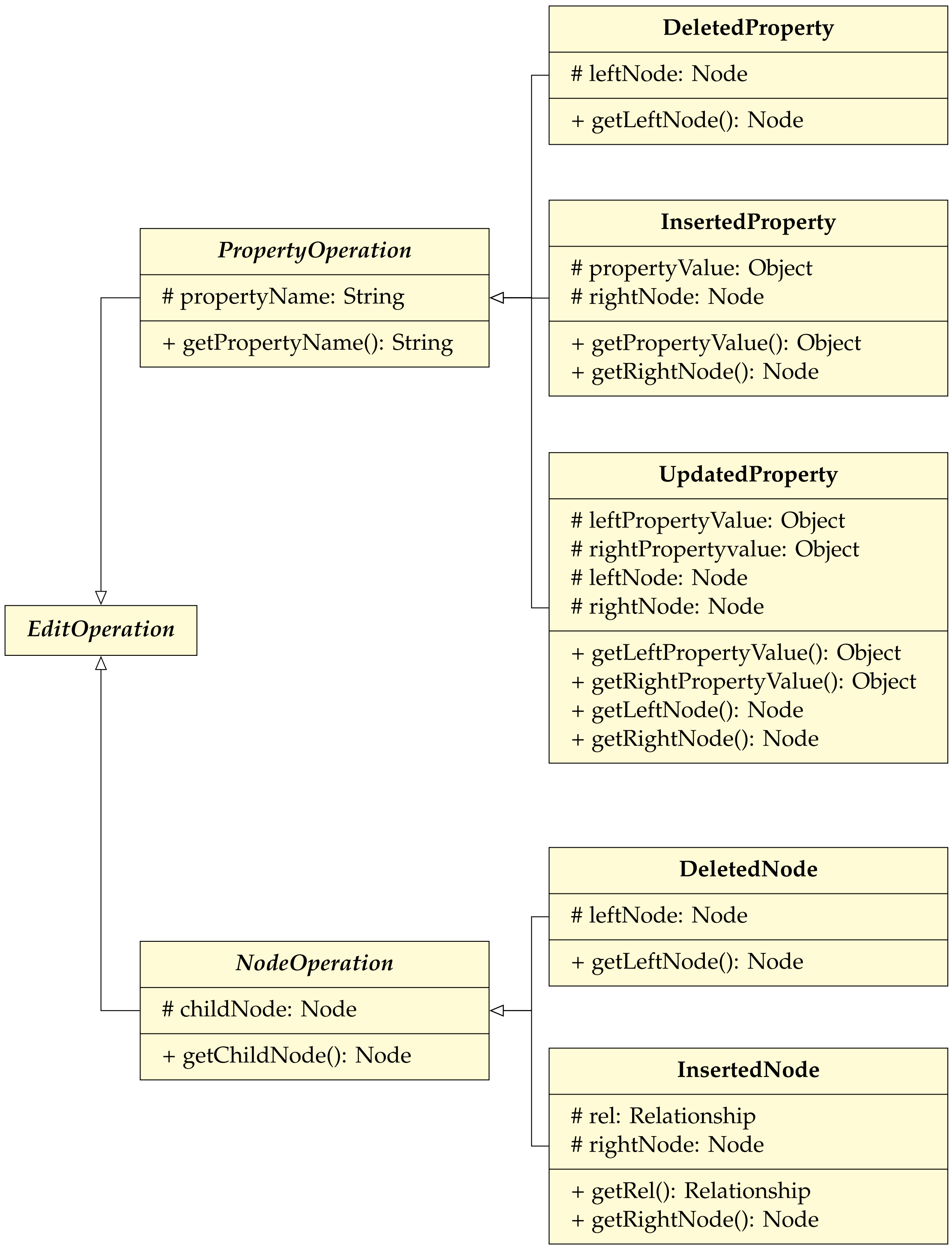


Figure 4.8.: A UML class diagram of edit operations.

4.7. Change Detection: Illustrative Examples

This section demonstrates the techniques proposed previously in this chapter for detecting changes between two different temporal versions of CityGML documents using their graph representations. These changes include semantic, structural, and geometric changes between CityGML objects.

For demonstration purposes, the CityGML document *FZK-Haus* (KIT IAI, 2017) in LOD2 is used. Its content is shown in Listing 3.1, and its 3D visualization is depicted in Figure 3.3. Despite its small size, the document includes many elements commonly found in many CityGML documents, such as generic attributes, *LinearRings*, polygons, and solids. This compact size also allows for manageable result control and evaluation.

In this experiment, the content of the only building in the original *FZK-Haus* document is manually modified with the following changes:

1. **Reordering of Generic Attributes**: The order of generic attributes is changed arbitrarily. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.1 and 4.2, respectively.

```xml
<gen:measureAttribute name="GrossPlannedArea">
  <gen:value uom="m2">120.00</gen:value>
</gen:measureAttribute>
<gen:stringAttribute name="ConstructionMethod">
  <gen:value>New Building</gen:value>
</gen:stringAttribute>
<gen:stringAttribute name="IsLandmarked">
  <gen:value>NO</gen:value>
</gen:stringAttribute>
```

Listing 4.1: Before changing the order of generic attributes.

```xml
<gen:stringAttribute name="ConstructionMethod">
  <gen:value>New Building</gen:value>
</gen:stringAttribute>
<gen:stringAttribute name="IsLandmarked">
  <gen:value>NO</gen:value>
</gen:stringAttribute>
<gen:measureAttribute name="GrossPlannedArea">
  <gen:value uom="m2">120.00</gen:value>
</gen:measureAttribute>
```

Listing 4.2: After changing the order of generic attributes.

2. **Reordering of Boundary Surfaces**: The order of all boundary surfaces is changed arbitrarily. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.3 and 4.4, respectively.

```
<bldg:boundedBy>
  <bldg:RoofSurface gml:id="Roof_Surface_1_North">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
<bldg:boundedBy>
  <bldg:WallSurface gml:id="Outer_Wall_Surface_1_West">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
<bldg:boundedBy>
  <bldg:GroundSurface gml:id="Ground_Surface">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
...
```

Listing 4.3: Before changing the order of boundary surfaces.

```
<bldg:boundedBy>
  <bldg:GroundSurface gml:id="Ground_Surface">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
<bldg:boundedBy>
  <bldg:WallSurface gml:id="Outer_Wall_Surface_1_West">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
<bldg:boundedBy>
  <bldg:RoofSurface gml:id="Roof_Surface_1_North">
    ...
  </bldg:WallSurface>
</bldg:boundedBy>
...
```

Listing 4.4: After changing the order of boundary surfaces.

3. **Reordering of Solid Member Surfaces**: The order of all seven boundary surfaces of the building's solid is changed arbitrarily. These surfaces are the same as the building's boundary surfaces and referenced by the solid using XLinks. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.5 and 4.6, respectively.

```xml
<gml:Solid>
  <gml:exterior>
    <gml:CompositeSurface>
      <!-- XLink references to the surfaces -->
      <gml:surfaceMember xlink:href="#Roof_Surface_1_North"/>
      <gml:surfaceMember xlink:href="#Roof_Surface_2_South"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_1_West"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_2_South"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_3_East"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_4_North"/>
      <gml:surfaceMember xlink:href="#Ground_Surface"/>
    </gml:CompositeSurface>
  </gml:exterior>
</gml:Solid>
```

Listing 4.5: Before changing the order of the solid's surfaces.

```xml
<gml:Solid>
  <gml:exterior>
    <gml:CompositeSurface>
      <!-- XLink references to the surfaces -->
      <gml:surfaceMember xlink:href="#Roof_Surface_1_North"/>
      <gml:surfaceMember xlink:href="#Ground_Surface"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_3_East"/>
      <gml:surfaceMember xlink:href="#Roof_Surface_2_South"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_2_South"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_4_North"/>
      <gml:surfaceMember xlink:href="#Outer_Wall_Surface_1_West"/>
    </gml:CompositeSurface>
  </gml:exterior>
</gml:Solid>
```

Listing 4.6: After changing the order of the solid's surfaces.

4. **Conversion of Measurements**: The unit associated to the building's measured height is changed from *m* to *mm*. At the same time, the measured value is updated from 6.52 to 6,520.0 Despite these modifications, both the elements represent the same measurement. The excerpts of the CityGML documents, both before and after these modifications, are displayed in Listings 4.7 and 4.8, respectively.

```
<bldg:measuredHeight uom="m">6.52</bldg:measuredHeight>
```

Listing 4.7: Before converting the unit of measurement.

```
<bldg:measuredHeight uom="mm">6520.0</bldg:measuredHeight>
```

Listing 4.8: After converting the unit of measurement.

5. **Update of Building Identifier**: The identifier of the building is changed. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.9 and 4.10, respectively.

```
<bldg:Building gml:id="FZK_HAUS_LOD2">...</bldg>
```

Listing 4.9: Before updating building ID.

```
<bldg:Building gml:id="FZK_HAUS_LOD2_UPDATED">...</bldg>
```

Listing 4.10: After updating building ID.

6. **Translation of Surfaces**: All seven boundary surfaces of the building are shifted 'upwards' by 1 length unit. This is achieved by updating the height values of their exterior boundaries stored in *LinearRing* elements. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.11 and 4.12, respectively.

```
<gml:LinearRing gml:id="PolyID...">
  <gml:pos>457842 5439088 118.317691453624 </gml:pos>
  <gml:pos>457842 5439093 115.430940107676 </gml:pos>
  <gml:pos>457842 5439093 111.8 </gml:pos>
  <gml:pos>457842 5439083 111.8 </gml:pos>
  <gml:pos>457842 5439083 115.430940107676 </gml:pos>
  <gml:pos>457842 5439088 118.317691453624 </gml:pos>
</gml:LinearRing>
```

Listing 4.11: Before moving surfaces 'upwards' by 1 height unit.

```xml
<gml:LinearRing gml:id="PolyID...">
  <gml:pos>457842 5439088 119.317691453624 </gml:pos>
  <gml:pos>457842 5439093 116.430940107676 </gml:pos>
  <gml:pos>457842 5439093 112.8 </gml:pos>
  <gml:pos>457842 5439083 112.8 </gml:pos>
  <gml:pos>457842 5439083 116.430940107676 </gml:pos>
  <gml:pos>457842 5439088 119.317691453624 </gml:pos>
</gml:LinearRing>
```

Listing 4.12: After moving surfaces ‘upwards’ by 1 height unit.

7. **Representation Change of Rings**: The syntactic representation of all seven *LinearRing* elements is changed, while preserving their geometric content. The excerpts of the CityGML documents, both before and after the modification, are displayed in Listings 4.13 and 4.14, respectively.

```xml
<gml:LinearRing gml:id="PolyID...">
  <gml:pos>457842 5439088 119.317691453624 </gml:pos>
  <gml:pos>457842 5439093 116.430940107676 </gml:pos>
  <gml:pos>457842 5439093 112.8 </gml:pos>
  <gml:pos>457842 5439083 112.8 </gml:pos>
  <gml:pos>457842 5439083 116.430940107676 </gml:pos>
  <gml:pos>457842 5439088 119.317691453624 </gml:pos>
</gml:LinearRing>
```

Listing 4.13: Before changing syntactic representation of rings.

```xml
<gml:LinearRing gml:id="PolyID...">
  <gml:posList srsDimension="3">
    457842 5439088 119.317691453624
    457842 5439093 116.430940107676
    457842 5439093 112.8
    457842 5439083 112.8
    457842 5439083 116.430940107676
    457842 5439088 119.317691453624
  </gml:posList>
</gml:LinearRing>
```

Listing 4.14: After changing syntactic representation of rings.

Both the original and modified versions of the CityGML document are subsequently mapped onto graphs using the mapping methods proposed in Chapter 3. These graphs are then compared using the matching strategies presented in this chapter. The results of this comparison process are summarized in Table 4.2.

Table 4.2.: Comparison results of both the original and modified *FZK-Haus* document.

Changes Made to the Document			**Result Evaluation**
Reordering of generic attributes		*N*	✓ Order independence
Reordering of boundary surfaces		*N*	✓ Order independence
Reordering of solid surfaces		*N*	✓ Order independence
Conversion of measurements		*N*	✓ Handling of measurements
Update of building identifier	1×	*U*	✓ ID independence
Translation of surfaces	7×	*T*	✓ Transformation detection
Representation change of rings		*N*	✓ Handling of syntactic ambiguities

N No real changes   *U* Updated property   *T* Translated surface

These results demonstrate that the matching techniques proposed in this study can effectively match CityGML objects irrespective of their identifiers, order of occurrence in the document, or their syntactic representations allowed by the encoding. In addition, the matching process is able to handle measurements coupled with unit conversions. Such syntactic and representation changes are purely model-based and do not reflect actual changes in the physical world. Thus, while relevant to data brokers, they hold little significance for other stakeholders like firefighters or the city mayor.

In contrast, the geometric translations observed in this experiment, along with the detected translation vectors, may indicate real changes in the city. They can be valuable to multiple stakeholders, such as architects and real estate managers. The interpretation of changes will be discussed in Chapter 5. Further demonstrations on the comparison of larger CityGML datasets are provided in Chapter 7.

4.8. Summary and Discussion

This chapter first discusses the relevant comparison algorithms currently in use for XML and GML documents, with a special focus on CityGML documents. It suggests a shift from direct comparison of CityGML documents in text form, which fails to account for their graph-based nature, to a comparison of their lossless and unambiguous graph representations. The changes detected in these graphs can then be used to reflect the changes that occurred between the original CityGML documents.

This chapter introduces the concept of graph isomorphism, a mathematical concept for structural similarity between two graphs. It then explores the problem of graph and subgraph isomorphism in graph theory, which determines whether two given graphs or subgraphs are structurally identical. Despite extensive research, these problems remain complex.

The chapter further discusses the challenges and complexity, but also the benefits of matching graph representations of CityGML documents. It proposes recursive methods for comparing these graphs, leveraging the rich semantic information available in CityGML to reduce runtime by only matching nodes and relationships with the same labelling. This approach is capable of handling both the structure and content of graphs.

To implement this, the chapter first proposes different techniques for comparing different types of thematic contents, such as generic attributes, measurements, and date-time values. It then presents methods for matching nodes and subgraphs. Each time a change is detected in the graphs, a change node is created and directly attached to the graph elements where the change occurred. This change node acts as a marker, highlighting the locations of the detected changes within the graphs for efficient retrieval and analysis in subsequent processes.

The chapter introduces the use of node and subgraph similarity to assess the similarity of graph elements based on their semantic and geometric content. These similarity levels can be extended depending on the applications and use cases.

One of the biggest challenges of the matching process is the handling of one-to-many and many-to-many relationships in large graphs, where a given subgraph may have a large number of potential matches. To avoid brute-force comparisons, the chapter proposes a variety of strategies for quickly determining the best potential match by leveraging their prominent features, without having to match their entire content. This applies to generic attributes, geometric objects, top-level features, and other object types. The geometric objects considered include 0D points, 1D line segments, 2D surfaces, and 3D solids. The presented methods are also capable of detecting transformations or deviations among these geometries, such as translation, rotation, resizing, or any combination thereof.

The chapter then describes how the detected changes can be documented as edit operations and edit nodes. It provides a conceptual modelling of edit operations, as well as details on how these edit operations can be mapped onto graph nodes. Finally, the chapter demonstrates the proposed methods and concepts in an illustrative case and evaluates the results.

Some notable observations and insights related to the concepts introduced in this chapter include:

1. **The Use of In-memory Objects**: The method *toObject(node)*, as presented in Section 3.6, can be utilized to reconstruct a subgraph back into its original CityGML object. These objects ensure full compatibility with object-oriented modelling principles employed in CityGML. Given that objects can be defined using different syntactic ways, and their graph representations only capture one such definition, the use of in-memory objects allows for the comparison of only their contents, regardless of how they are represented. This eliminates the need for constant data fetching and parsing from the subgraph in the graph database. However, as these in-memory objects are stored entirely in main memory, this approach is primarily used for small, predominantly geometric objects with high syntactic ambiguities while matching.

2. **Prioritization of Speed during Matching**: The method *findBest(left, right, ref)*, as proposed in this chapter, is employed to determine the best match for each reference subgraph in one-to-many and many-to-many relationships. As a large number of matching candidates may exist in large graphs, the primary objective of this method is to identify the most suitable match as quickly as possible. Therefore, complex geometric operations, especially in 3D space, are typically avoided or only employed when necessary. Instead, heuristic measures, such as matching geometries based on their centroid distances, normal vectors, and overlaps, are generally prioritized.

3. **Handling of Geometric Uncertainties**: To deal with geometric uncertainties in matching, this thesis employs predefined, small error tolerances for metrics such as lengths, angles, areas, and volumes. Given the cost-effectiveness of a single numeric comparison required for these values, this approach suffices in the majority of use cases and test datasets examined within this study. For a more comprehensive approach, various complex statistical tests can be employed, such as the chi-squared (χ^2) test (Pearson, 1900) along with a covariance matrix and its pseudo-inverse (Dehbi & Plümer, 2011).

4. **Application of Edit Nodes**: The edit operations and edit nodes introduced in this chapter represent the most fundamental level of change nodes. This is indicated by their direct link to the source graph elements where the changes occurred. These edit nodes serve as a basis for subsequent analyses, such as the automatic update of existing CityGML documents. In Chapter 5, the process of change interpretation is explored, where more complex and expressive changes are derived based on these edit nodes.

5. Change Interpretation in Semantic 3D City Models

The methods presented in Chapter 4 compare CityGML documents by matching their graph representations. For each detected change in the data, from updated properties to deleted subgraphs, an edit node is created and attached directly to the graph nodes where the changes are detected. While these edit nodes can provide direct access to all detected changes and their context, their sheer number, often reaching millions or billions for large datasets, poses a significant challenge for humans to manage and comprehend. Detecting changes is only half of the solution; the other half is to understand them.

While each individual change at the lowest level may be insignificant, collectively, they may uncover valuable insights into the procedures and reasoning hidden behind these changes. By considering changes within a constellation or pattern as a whole, the sheer number of detected edit nodes can be reduced significantly, while the semantic content of interpreted changes can be increased at the same time. For example, as illustrated in Section 4.7, the matching process reported a shift in the location of all roof, wall, and ground surfaces of a building. When all translations are in the same orientation and have the same offset, a pattern is found, indicating that the entire building has been moved. Thus, all these individual translations can be represented by a single interpretation suggesting a movement of the building, which exhibits a much higher level of semantic content than each of the individual changes.

However, most studies on changes in CityGML documents often solely focus on identifying base changes (or edit nodes) without further considering their interpretation or pattern identification. In many current smart city deployments, the common approach to detecting patterns is to formulate database queries for each pattern and execute them in an ad-hoc manner. This not only requires expertise in the database structure but may also result in undesired scheduling and efficiency issues, particularly when rules are interdependent, forming a 'pattern' of patterns.

Furthermore, the perception of different types of changes varies among different groups of stakeholders. For example, a change in the geometries of buildings might be crucial to data brokers and urban analysts, whereas a city mayor may prioritize top-level changes for decision making, such as recently constructed or demolished buildings that contribute to the district's changes in living space. Additionally, like data, stakeholders' interest in changes evolves over time, adding complexities to the modelling process of both stakeholders and changes. Initial studies have touched on several aspects of this problem (Nguyen & Kolbe, 2020, 2021, 2022; Nguyen & Kolbe, 2024), but none has yet provided a comprehensive workflow containing both the detection of changes in CityGML datasets and the derivation of meaningful insights for each group of stakeholders.

The methods presented in this chapter are illustrated through a use case example, as visualized in Figure 5.1. These methods are used to interpret changes between graph representations of CityGML documents. They are summarized as follows:

1. **Defining Pattern Rules** (Figure 5.1a): A hierarchical classification of changes is proposed in Section 5.2. Changes detected between the graph representations of CityGML documents are matched against a set of predefined rules for finding change patterns. A rule network is proposed in this step, allowing all pattern rules to be defined within a single graph, as explained in Section 5.3.

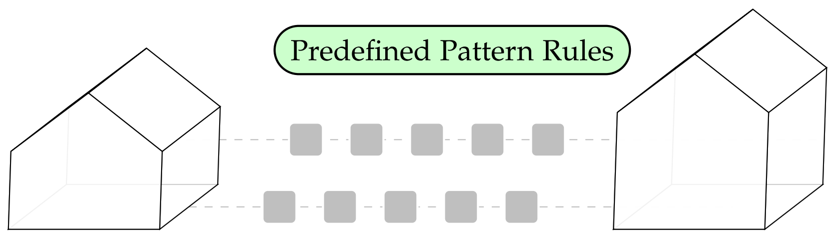
2. **Matching Change Patterns** (Figures 5.1b and 5.1c): Patterns in the changes are identified in a bottom-up manner. Lower-level changes are aggregated to form higher-level changes that contain more semantic meaning. Meaningful interpretations can then be derived from these detected change patterns. The methods for the pattern matching process are presented in this step in Section 5.4.

3. **Evaluating Change-Stakeholder Relations** (Figure 5.1d): The derived interpretations are further evaluated to determine which changes are relevant to a given stakeholder and, conversely, identify which stakeholders are interested in a particular type of changes. Path-tracing techniques within the network that models the interrelations between changes and stakeholders are explained in Section 5.5.

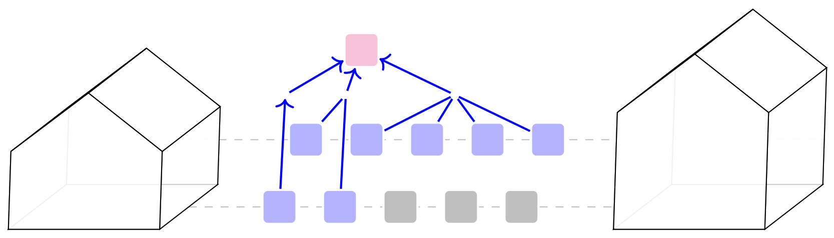
The entire workflow takes place within the same graph database used to store the graph representations of CityGML documents and their changes, as discussed in previous chapters.

The content of this chapter substantially expands upon the author's earlier publications, which are detailed as follows:

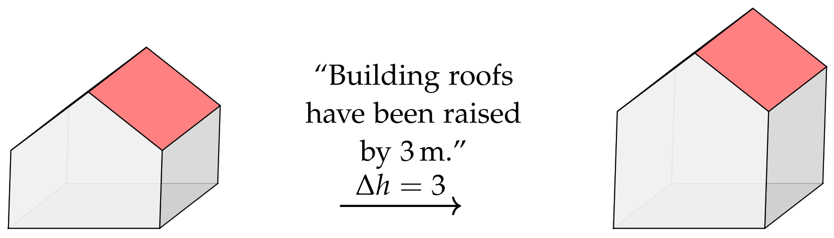
1. Nguyen, S. H., & Kolbe, T. H. (2020, September). A Multi-Perspective Approach to Interpreting Spatio-Semantic Changes of Large 3D City Models in CityGML using a Graph Database [15th International 3D GeoInfo Conference 2020, University College London (UCL), London, UK]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 143–150, Vol. VI-4/W1-2020). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-vi-4-w1-2020-143-2020.



(a) Matching detected changes (grey) against predefined pattern rules (green)



(b) Creating a new interpretation node (red) that represents a matched pattern



(c) Deriving interpretations from the detected patterns

Path-tracing techniques in *change-stakeholder network*

"Building roofs have been raised by 3 m." $\xrightarrow{\Delta h = 3}$

Relevant to:
Property Manager
Fire Safety Officer
City Mayors

(d) Evaluating interpretations with respect to stakeholders

Figure 5.1.: An illustration of the change interpretation process. The 3D model on the left represents the original building, while the model on the right visualizes that same building, but with raised roofs and vertically enlarged walls. The geometries of the roof and ground surfaces remain the same.

2. Nguyen, S. H., & Kolbe, T. H. (2021, October). Modelling Changes, Stakeholders and their Relations in Semantic 3D City Models [16th International 3D GeoInfo Conference 2021, New York University (NYU), NY, USA]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 137–144, Vol. VIII-4/W2-2021). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-viii-4-w2-2021-137-2021.

3. Nguyen, S. H., & Kolbe, T. H. (2022, October). Path-tracing Semantic Networks to Interpret Changes in Semantic 3D City Models [17th International 3D GeoInfo Conference 2022, University of New South Wales (UNSW), Sydney, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 217–224, Vol. X-4/W2-2022). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-x-4-w2-2022-217-2022.

4. Nguyen, S. H., & Kolbe, T. H. (2024, September). Identification and Interpretation of Change Patterns in Semantic 3D City Models [18th 3D GeoInfo Conference 2023, Technical University of Munich (TUM), Munich, Germany]. In T. H. Kolbe, A. Donaubauer & C. Beil (Eds.), Recent Advances in 3D Geoinformation Science (pp. 479–496). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43699-4_30. Reproduced with permission from Springer Nature Switzerland.

5.1. Foundations and Related Work

In computer science, pattern matching is the process of determining whether a known sequence or pattern exists within an input. The most basic form is sequence matching of strings with one-dimensional input and rules often written as regular expressions. As the complexity of data increases, the pattern matching problem becomes more challenging. Given that CityGML documents and their changes are represented as graphs in this study, this leads to the pattern matching problem of graphs.

One of the main objectives of this research is to detect patterns among detected changes based on predefined rules. This involves the use of rule-based systems and other related concepts. Basic components of a rule-based system include a knowledge base containing predefined rules and facts, a rule interpreter that assesses and executes these rules, a working memory that holds temporary data, and a user interface for human interaction (Grosan & Abraham, 2011; Masri et al., 2019).

To interpret changes in CityGML graphs, this thesis proposes concepts that are, to some extent, related to graph pattern matching and rule-based systems. These include Rete networks, Petri nets, and graph transformation systems, which will be introduced briefly. This section addresses Research Question RQC1 (Existing Rule-based Systems).

5.1.1. ECA Rules and Rete Networks

The Event-driven Architecture (EDA) is an established software architecture paradigm, specifically designed for event-based applications. In this context, an event signifies a state change, a problem, or an opportunity (Michelson, 2006). Immediately upon its occurrence, an event is distributed to all observers, including both humans and machines, and subsequent actions are initiated based on predefined rules. In an Active Database Management System (ADBMS), a database management system equipped with an event-sensitive response mechanism, Event Condition Action (ECA) rules are employed to describe the reactive behaviour of the system (Dittrich et al., 1995). Each ECA rule is typically composed of three components: an event, some logical conditions, and an action. An action is executed if both the corresponding event occurs and its conditions are satisfied. To evaluate these conditions and trigger actions, many database systems employ rule engines, such as the Rete match algorithm and its variants.

The Rete match algorithm, an efficient technique for matching a multitude of patterns against a large set of objects, was initially developed for production system interpreters (Forgy, 1982). A production system typically consists of a set of conditional *if-then* statements, a global **working memory** that holds temporary data, and a rule interpreter that evaluates rule conditions and triggers actions.

When dealing with a large set of interdependent rules (known as inference, where one rule's outcome influences another), a brute-force approach may iterate over each rule, evaluate its conditions, trigger actions if necessary, and then repeat the whole process. The Rete algorithm circumvents this repetitive iteration over input data by storing matched objects in their respective rules. When a new element is added or an existing one is removed from the shared working element, the impacted rules are notified, and their list of stored objects is updated accordingly. To further avoid repeated iteration over rules, a directed acyclic graph representation of rules is employed for pattern matching. This graph is referred to as a *Rete network* (Forgy, 1982).

A significant advantage of the Rete match algorithm is its processing speed, attributed to the working memory holding temporary data for each pattern during execution. This memory stores previously read input objects and processed results in containers, triggering actions when full. Therefore, it enables both on-the-fly type checking and on-demand reactivation of pending rules.

Moreover, the Rete algorithm is particularly effective in scenarios where the input is a stream of unordered, differently typed objects, which is subject to frequent insertion and deletion operations. The performance of the algorithm largely depends on the implementation of its working memory. However, in worst-case scenarios, the original Rete algorithm may store all temporary data in main memory, degenerating memory efficiency.

The methods proposed in this thesis, which enable formulation of rules for identifying patterns in detected changes, utilize an extended version of the aforementioned working memory, effectively avoiding excessive memory consumption. Additionally, the Rete network only permits nodes with a maximum in-degree of two, implying that joining n inputs may require $n - 1$ consecutive nodes. While numerous variants of the Rete algorithm exist today that offer improvements to such limitations, some are either not fully disclosed (as in the case of Rete II) or tailored for specific applications and use cases. Therefore, these variants are not further discussed in this thesis.

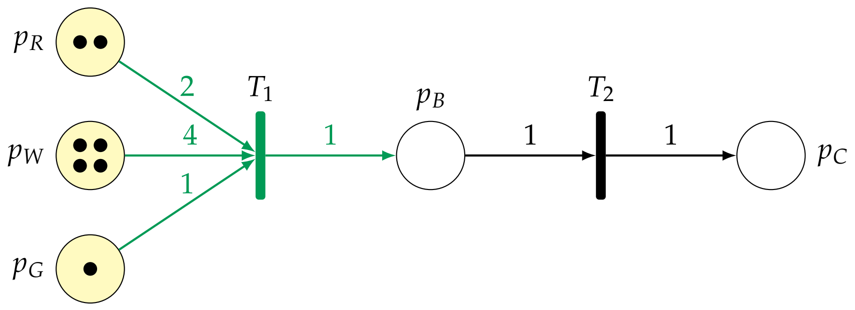
5.1.2. Petri Nets

Petri nets were first proposed to model parallel and distributed systems (Petri, 1962). The inception of this concept was driven by the author's intention to graphically represent and study chemical reactions (Reisig, 2013). A Petri net is a bipartite graph consisting of two types of nodes: *places* and *transitions*. Nodes between partitions are connected via directed *arcs*.

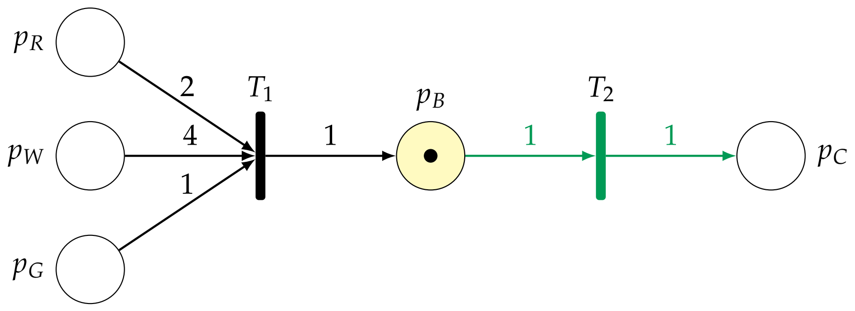
In a Petri net, **tokens** are elements that can be stored within places and can traverse between adjacent places via their connecting transitions. In rule-based systems, places semantically represent the current state or conditions of rules, while transitions represent actions. When a place obtains sufficient number of tokens, it triggers its outgoing transitions. This process consumes all input tokens and produces new ones. The quantity of tokens consumed and produced is dictated by the weights assigned to arcs connecting these places and transitions.

Given an initial configuration or distribution of tokens across places, Figure 5.2 illustrates an example of such a Petri net as an attempt at modelling the patterns of translation changes in the boundary surfaces of a building, as well as in buildings of a city model. This net corresponds to the example presented in Section 4.7. When all boundary surfaces of a building have been translated by the same translation vector, it indicates that the entire building has been translated by that same vector. Similarly, if all buildings within a city model have been translated by the same translation vector, it indicates a systematic translation in the entire dataset.

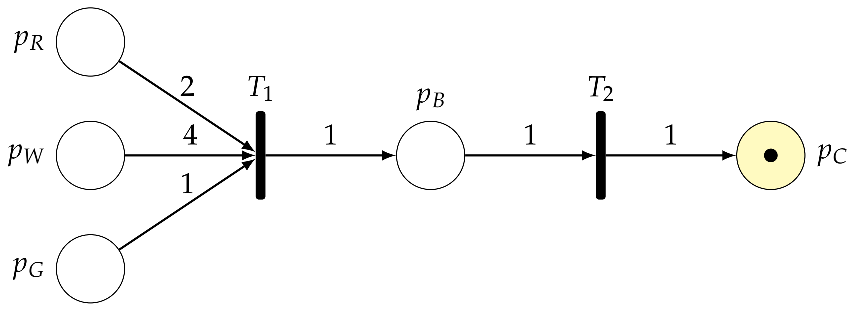
Reachability is one of the most prominent and long-established algorithmic problems in Petri nets. This problem involves determining whether a certain configuration of the net can be achieved, given an initial configuration of a Petri net. It was shown in the 1970s that the reachability problem requires exponential space (Lipton, 1976). Subsequent studies in the following decades further established that the problem is non-primitive recursive (Czerwiński & Orlikowski, 2022; Leroux, 2022). A *reachability graph* is often constructed to study this problem. However, many structural analyses of Petri nets prefer to employ transition and place invariants instead of a reachability graph due



(a) Initial configuration



(b) After T_1 has been triggered



(c) After T_2 has been triggered

Figure 5.2.: An example of a Petri Net in its different configurations. The nodes p_R, p_W, p_G, p_B, and p_C symbolize the places representing roof, wall, and ground surfaces, as well as buildings and city models, respectively. Each filled circle denotes a token. Tokens in each place depict translations of the same vector. Therefore, this Petri net simulates a pattern, where consistent translations in all boundary surfaces of a building indicate its translation by the same vector. When applying this to all buildings within a city model, a systematic translation can be deduced. In this example, all buildings must contain two roof surfaces, four wall surfaces, and one ground surface. Moreover, the city model contains only one building.

to their compactness and simplicity. In certain Petri nets, particularly those containing cycles, the reachability graph may become infinite. In such cases, an approximation of the reachable configurations can be computed instead. This is known as the *coverability graph*.

Petri nets have strong scalability and modelling potential in rule-based applications. The mechanism of token consumption and production aligns with the description of numerous aggregation rules of changes employed in this thesis. Moreover, Petri nets and their configurations can be mathematically represented as both graphs and matrices. This dual representation allows for the application of many well-established mathematical methods from both the fields of graph theory and linear algebra.

However, tokens in the classical Petri nets lack attributes and types, rendering them indistinguishable. In contrast, changes in CityGML documents analysed in this thesis are both typed and attributed. Some variants of Petri nets, such as the Coloured Petri Nets (Jensen, 1987), permit the use of typed and attributed tokens.

While tokens in a Petri net can be used to model rule-based systems, their quantity for each place and transition is static and must be fully known before execution. For instance, the Petri net shown in Figure 5.2 dictates that, in order to produce exactly one token in p_B, there must exist exactly two tokens in p_R, four tokens in p_W, and one token in p_G. However, the number of boundary surfaces varies vastly across buildings in a CityGML document and are thus not known before execution. In addition, while the example shown in Figure 5.2 presumes that all tokens represent translations by the same vector, the Petri net itself is incapable of enforcing such constraints.

Therefore, this thesis employs a special type of graphs to define rules for matching change patterns in graph representations of CityGML documents. These graphs are related to Petri nets, particularly in their use of tokens to emulate the aggregation relationships among changes. In contrast to the classical Petri nets, the tokens used are attributed and typed to allow for the modelling and handling of complex, semantic changes.

5.1.3. Graph Transformation Systems

Graphs serve as both an expressive data structure and a modelling instrument for a vast number of theoretical and practical problems. Rule-based systems employing graphs often also involve graph transformation techniques to a certain degree. Since this study stores both the graph representations of CityGML documents and their detected changes in the same graph database, the process of matching their change patterns can also be considered as an application of graph transformation.

Graph transformation was first proposed as a graph grammar for rule-based rewriting of non-linear data structures (Heckel, 2006; Pfaltz & Rosenfeld, 1969; Pratt, 1971).

This concept finds applications in fields such as image recognition and string-graph translation. In graph transformation, rules are given using two distinct types of graphs: **type graphs** and **instance graphs**. A type graph defines the conceptual model of object classes and their behaviour. On the other hand, an instance graph is a snapshot that contains concrete values and structure, as prescribed by the corresponding type graph. Consequently, all instance graphs of a specific type can be represented by a single type graph.

Graph transformation systems are a powerful tool for handling complex semantic structures. Type-enabled graph transformation can extract and utilize hidden context information from the graph representations of objects, such as the semantic context between graph elements, enabling more complex analyses. This advantage is relevant to this research and has been utilized in previous studies, such as the analysis and semantic transformation of the CityGML data model (including its ADEs), originally defined using XSDs (Yao, 2020).

However, graph transformation utilizes graph and subgraph isomorphism, which, due to their inherent complexity, are not suitable for large search graphs or cases where runtime efficiency is a priority. Another limitation of graph transformation is that the structure of both the type and instance graphs, as dictated by the transformation rules, must be known before execution.

For instance, like the Petri net example provided in Figure 5.2, when defining a rule for a pattern in translation changes between a building and its boundary surfaces, the number of roof, wall, and ground surfaces of each building must be known. However, this information is only available at runtime. Moreover, graph and subgraph isomorphism invariably employs exact structural matching, which may not be feasible in real-world scenarios where some deviations should be tolerated.

In addition, graph transformation typically exhibits non-deterministic behaviour. The outcomes of the rewriting process vary depending on the sequence in which both the rules are executed and the graph occurrences are selected for the application of these rules. For instance, when multiple rules are applicable for a subgraph, the application of any rule may alter the structure of the subgraph, rendering subsequent rules inapplicable. On the other hand, when multiple instance graphs are eligible for the same transformation rule, the results can differ depending on the sequence in which each graph is transformed.

Therefore, the methods proposed in this thesis employ a simplified approach to graph transformation that employs heuristic optimizations for graph and subgraph isomorphism. Like the matching process discussed in Chapter 4, these methods leverage the labelling of nodes and relationships, along with their semantic context within the graphs, to improve the runtime efficiency of the pattern matching process in large graphs.

5.2. Hierarchical Modelling of Changes in CityGML

Addressing Research Question RQC2 (Hierarchical Modelling of Changes), Figure 5.3 presents a UML class diagram of changes typically detected between temporal versions of CityGML documents examined in this thesis. While this class hierarchy is specifically designed for changes in semantic 3D city models given in CityGML, it can also be extended and applied to other domains, such as in the field of BIM, which similarly employs semantic and geometric contents. The classes proposed in the UML diagram will be discussed in further detail in the following sections.

5.2.1. Appearance Changes

The class *AppearanceChange* represents changes in the visually observable properties of city objects that are associated with their surfaces. These properties extend beyond visual data and can be linked to any theme, such as solar potential and infrared radiation. In each Level of Detail (LOD), different appearances can be defined for a variety of themes. As a result, themes serve as identifiers for thematic appearance groups. Changes in these themes may include a name change for a specific theme or reassignment of themes among different LODs, while the associated appearances remain unaffected. The class *ThemeChange* covers these changes.

On the other hand, changes can also occur within the appearances of a theme. In CityGML, appearances are portrayed in the surface data elements, which enable the modelling of simple surface properties with constant light reflection as materials, and other coordinate-based surface properties as textures (Gröger et al., 2012). Therefore, the class *SurfaceDataChange* represents changes in these objects. Additionally, the surface data elements are linked to their corresponding geometric objects using their identifiers, enabling the connection of both the closely related surface data and surfaces while preserving the original geometric contents. Hence, this class solely covers changes that occurred on the surface data side; changes of the surfaces themselves are considered as geometric changes, as will be explained in Section 5.2.3.

5.2.2. Semantic Changes

CityGML documents are rich in semantic content, which covers the thematic, functional, and logical aspects of city objects. Thematically, an object can store information specific to its type, such as its name and identifier. Moreover, objects of the same type can be grouped together. This enables the identification and distinction of city objects of various types (such as buildings, bridges, and tunnels) in relation to their hierarchy. For instance, an office is a building, which belongs to the class of all city objects.

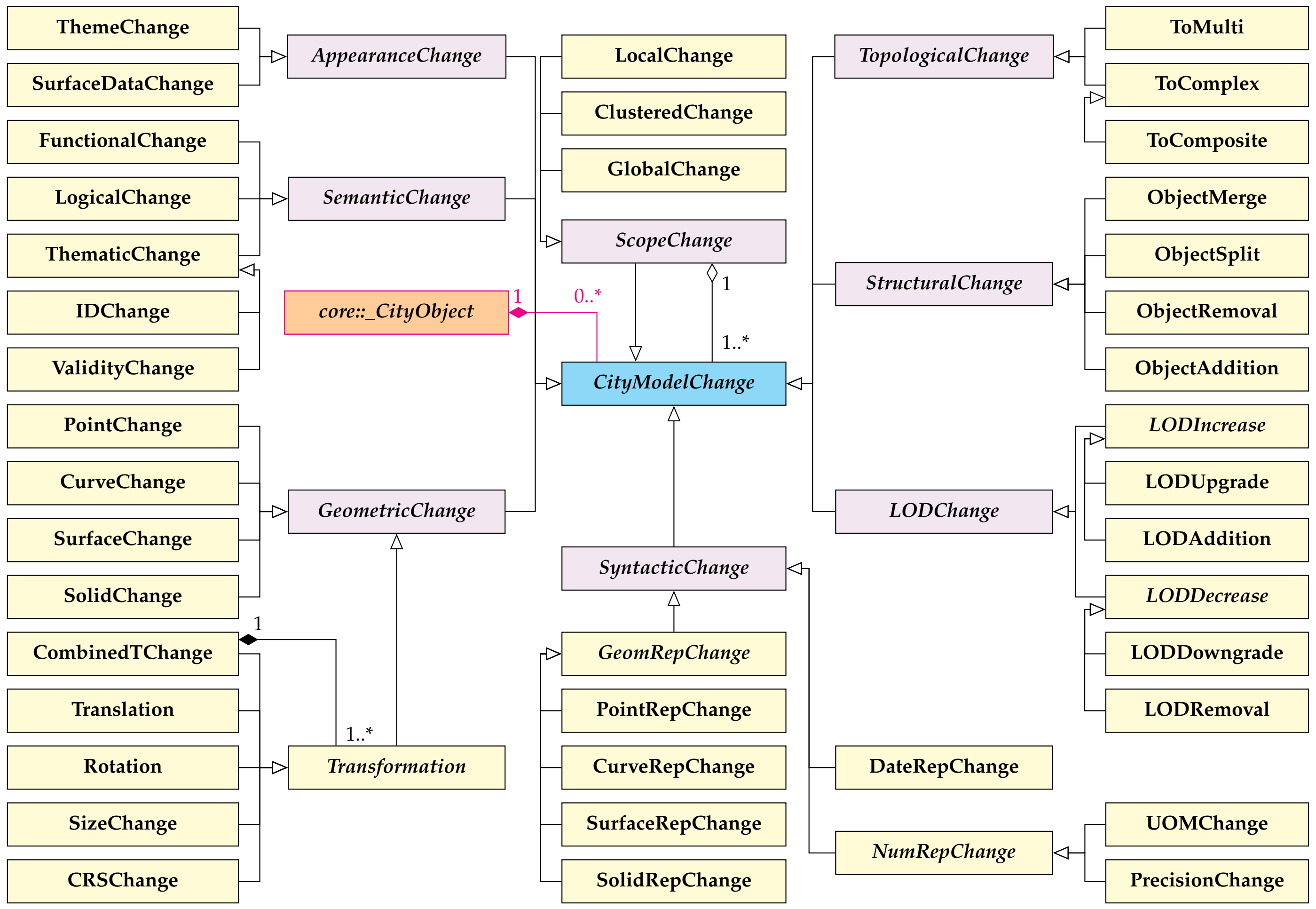


Figure 5.3.: A UML class diagram of changes in CityGML documents. Each change is attached to a source city object, represented by the class _*CityObject* (orange) from the core module of CityGML.

Functionally, an object can be a component of a collection, or a collection of other components. This is typically observed in the *composition* and *aggregation* relations. Lastly, objects are structured based on their logical criteria and interrelationships (Kolbe & Donaubauer, 2021). City objects are represented using a set of predefined classes, such as *Building*, *Bridge*, and *Tunnel*. Complex objects can be recursively divided. For example, a building may consist of building parts, installations, and rooms, which are further defined using boundary roof, wall, and ground surfaces.

Therefore, changes in the aforementioned aspects are represented by the classes *ThematicChange*, *FunctionalChange*, and *LogicalChange*, respectively. For instance, updating building's height is categorized as a thematic change, while converting a wall surface into a window is considered a functional change. Additionally, a thematic change can be further classified as *IDChange* or *ValidityChange*. An instance of *ID-Change* represents a change in an object's identifier, typically its *gml:id*. On the other hand, an instance of *ValidityChange* indicates a change in the attributes *creationDate* and *terminationDate*. These attributes are displayed in Figure 3.4.

Notably, certain logical changes in semantic 3D city models may suggest adjustments in the class structure and relationships within the encoding specification itself. For example, building parts may no longer consists of other building parts. These refer to the level M1 (i.e., model level) and higher according to the Model-driven Architecture (MDA) in UML in general, and CityGML in particular (Kleppe et al., 2003; Kutzner, 2016). This implies that such changes typically occur between different versions of the conceptual model, such as between CityGML versions 1.0, 2.0, and 3.0. However, a comparison between these versions is out of the scope of this research. Conversely, the structural changes, as introduced in Section 5.2.5, refer solely to the level M0, which is the instance level according to the MDA. Thus, such changes can be classified using the modelling rules allowed by the same version of CityGML.

5.2.3. Geometric Changes

CityGML employs GML3 geometries, which is based on the ISO 19107 model (Herring, 2020). This includes a set of geometric primitives for each dimension, ranging from 0D up to 3D, specifically *Point*, *_Curve*, *_Surface* and *_Solid*. As mentioned in Section 3.1.1, besides the classes *Point* in 0D and *Solid* in 3D, the CityGML encoding standard only utilizes the class *LineString* of *_Curve*, and the class *Polygon* of *_Surface* for the modelling of 1D and 2D geometric objects, respectively. These geometric primitives can then be combined to form more complex geometries, such as aggregates, complexes, and composites. Therefore, geometric changes in CityGML documents are represented by the class *GeometricChange* and its subclasses *PointChange*, *CurveChange*, *SurfaceChange*, and *SolidChange*, covering also corresponding aggregates and composites.

Additionally, the class *Transformation*, a specialization of the class *GeometricChange*, represents commonly observed transformations of geometric objects in semantic 3D city modelling. As mentioned in Section 4.5.4, these transformations may include translation, rotation, resizing, and any combination thereof. Furthermore, given that each geometry is assigned with a 3D Coordinate Reference System (CRS), a deviation in these spatial reference systems results in a transformation of all point coordinates within the affected geometries. This transformation is required to align geometries to one common coordinate system prior to conducting any further spatial analyses. Thus, such changes are represented by the classes *Translation*, *Rotation*, *SizeChange*, *CRSChange*, and *CombinedTChange*, which is both a subclass and a composition of the class *Transformation*.

Compared to other types of geometric changes, transformation changes are more challenging due to their high computational complexity in 3D. However, once detected, they offer a more profound understanding of the geometric modifications made to the city objects. For example, an increase in the height of all walls may also indicate a translation of roof and ground surfaces. When combined with the scopes introduced in Section 5.2.8, transformation changes can aid in identifying many common systematic changes, such as a consistent shift in the locations of all city objects or changed spatial reference systems across entire datasets.

Furthermore, CityGML permits the reuse of geometric contents via implicit geometries. A single change to the source geometry will result in changes in all objects that reference it. Such changes can be identified by examining their scope within the datasets, as mentioned in Section 5.2.8.

All geometric changes covered by the class *GeometricChange* indicate actual modifications to the numeric and geometric content of objects. Changes, which merely impact the syntactic representations of the same geometry, are considered as instances of the class *SyntacticChange*, as explained in Section 5.2.7.

Moreover, modifications to the aggregate, complex, and composite geometries often lead to changes not only to the geometric but also the topological content. Such topological changes are addressed separately in Section 5.2.4.

5.2.4. Topological Changes

Topology, like appearance, semantics, and geometry, is a crucial information aspect of the CityGML data model. It allows for the storage of geometric objects with implicit topological relations, as enabled by the geometrical-topological model of CityGML briefly explained in Section 3.1.1. This is particularly the case for complex objects formed from geometric primitives in each dimension: aggregates, complexes, and composites.

While an aggregate permits unrestricted spatial and topological relations between its members, a complex is a collection of geometric primitives that either are disjoint or touch at most at their boundaries. Moreover, a composite is a specialized complex that allows only geometric components of the same dimension, and they must be adjacent at their boundaries. A 2D illustration of these topological relations are shown in Figure 3.2 in Section 3.1.1.

In addition, CityGML also allows for the explicit modelling of the topological adjacency relations between objects. A shared boundary, such as a curve boundary between two adjacent surfaces or a surface boundary between two adjacent solids, can be defined once and then referenced by other adjacent features and geometries. For example, a wall surface shared by two adjacent buildings can be defined only once in the solid representing the first building and then referenced by the second solid representing the other building. These adjacency relations are realized using the shared geometries' identifiers as hyperlinks, or XLinks. This not only reduces redundancy but also maintains the explicit topological relations between objects.

To model changes in the topological relations between city objects, the classes *ToMulti*, *ToComplex*, and *ToComposite* are proposed, where the class *ToComposite* is a subclass of *ToComplex*, reflecting that a composite is also a complex geometry. These classes represent changes that result in unrestricted, disjoint or connected, and explicit connected topological relations between geometries of the same city object, respectively.

5.2.5. Structural Changes

In CityGML, top-level features, such as buildings, bridges, and tunnels, can be subdivided into smaller components, mirroring their physical structure in the real world. For example, a building can comprise an arbitrary number of building parts, rooms, and installations. These are further bounded by boundary surfaces, such as roof, wall, and ground surfaces. The class *StructuralChange* represents modifications to the structure of such objects. Depending on the methods, these changes can be further categorized into *ObjectMerge*, *ObjectSplit*, *ObjectRemoval*, and *ObjectAddition*. Further discussion can be found in Section 7.4.2.

5.2.6. LOD Changes

As mentioned in Section 3.1.1, CityGML supports multi-scale modelling with five different LODs (LOD0-4). A feature can be assigned with one or multiple LODs at the same time. Consequently, changes regarding the LODs can occur in two directions: an increase or a decrease in the level of detail, represented by the classes *LODIncrease* and *LODDecrease*, respectively. Both of these are subclasses of the class *LODChange*.

An increase in the level of detail of an object can be thought of as an increase in its highest available LOD. This includes an upgrade of a single available LOD to a higher one, or an addition of a higher LOD to the existing ones. These are covered by the subclasses *LODUpgrade* and *LODAddition* of *LODIncrease*, respectively. For instance, if an object was defined in LOD1 and LOD2, and is now available in LOD2, LOD3, and LOD4, the presence of the new highest LOD4 indicates an increase in the level of detail. This change can be interpreted as a combination of one upgrade from LOD1 to LOD3 and one newly added LOD4. Although changes within the objects of LOD2 may still persist, they do not reflect changes in the LODs themselves and should be covered by other types of changes, such as geometric and appearance changes.

Similarly, a decrease in the level of detail of an object can be thought of as a reduction in its highest available LOD. This includes a downgrade of a single available LOD to a lower one, or the removal of the highest LOD from the existing ones. These are represented by the subclasses *LODDowngrade* and *LODRemoval* of *LODDecrease*, respectively.

5.2.7. Syntactic Changes

The majority of the changes discussed above have a direct correlation with the reality and are typically the outcomes of some actions in the real world. However, some changes do not exhibit this relationship and are relevant at the model level only. These are primarily caused by the change in the syntactic representations of objects, as allowed by the underlying encoding standards. Such changes are covered by the superclass *SyntacticChange* and its subclasses *DateRepChange*, *NumRepChange*, and *GeomRepChange*.

As outlined in Sections 4.3.3 and 4.5, date-time values can be depicted in various ways while maintaining the same content. The representation of date and time accommodates many variations, such as based on the calendar used (Gregorian, lunar calendar, etc.), along with a variety of localizations based on countries and time zones. Such variants are covered by the class *DateRepChange*.

The class *NumRepChange* covers changes in the representations of numeric values. These changes are often observed in floating-point numbers, which can be represented by a combination of significands, bases, and exponents. For example, the numbers 1.234, 12.34×10^{-1}, and 1234×10^{-3} are different floating-point representations, yet they represent the same value. In addition, real-world object measurements often contain instrument and rounding errors that can be tolerated up to a certain threshold. A lower tolerance threshold requires a higher precision in the measurements. In practice, values that fluctuate within a small enough error tolerance threshold, referred to as ϵ in Chapter 4, are often not further differentiated and can be considered as acceptable numeric representations of the same value. For example, both 1.234 and 1.235 represent the same value within an error tolerance of 0.001. These variations are represented by the class *PrecisionChange*.

Furthermore, as explained in Section 4.3.3, measurement values are always assigned with a predefined unit of measurement (uom), such as in millimetres and metres. Similarly, the aforementioned error tolerance can also be defined with a unit. This is a crucial factor when comparing two measurements. For instance, with an error tolerance of 1 mm, two measurements of 1.001 m and 1,000 mm are considered equal. This is classified as a *UOMChange* combined with a *PrecisionChange*.

A significant part of representation changes is observed in geometric objects. As detailed in Section 4.5, two points are considered geometrically equivalent if they are located within the neighbourhood of each other, confined by a given error tolerance. *LineString* objects, whether composites or singular geometries, are considered geometrically equivalent if all their distinct vertices are geometrically equivalent. Two polygon surfaces are considered geometrically equivalent if the shape bounded by their exterior, subtracted by their interiors, represent the same geometry. Lastly, two solid objects are considered geometrically equivalent if their overlapping volume equals their respective volumes. These correspond to the similarity level *SAME_GEOMETRY* returned by the method *findBest(left, right, ref)* outlined in Algorithm 7. Therefore, these changes in the representation of geometric objects are covered by the classes *PointRepChange*, *CurveRepChange*, *SurfaceRepChange*, and *SolidRepChange*, respectively.

5.2.8. Scope Changes

A change can have an effect and impact on numerous elements in the city model. This can be observed by analysing its scope, as represented by the class *ScopeChange*. A change in scope is a collection of other types of changes proposed above, such as semantic, geometric, and syntactic changes. Depending on the extent and size of this collection, scope changes are further classified into three subclasses: *LocalChange*, *ClusteredChange*, and *GlobalChange*. This aggregation allows for efficient retrieval of associated changes within the same scope.

A local change impacts a specific attribute, element, or object. For example, a change in the generic attribute storing the energy consumption of a building has a limited scope and is considered local. On the other hand, a clustered change occurs over a number of objects that are spatially or semantically related. For instance, all objects within a spatial region may be extracted, modified, and re-imported into the city model.

Lastly, a change is considered global or systematic when it is applied to all objects of the same type in the entire city model. For instance, a global height offset between two city models can be derived by finding a systematic change in the height coordinates of all geometries in the entire datasets.

The complexity to determine these scopes increases with their coverage and granularity. An indicator accounting for the affected elements (both semantically and spatially) can be stored for each change. Their scope can then be analysed by comparing the indicator's value with a set of predefined thresholds, such as those relative to the number of objects within a top-level feature and to the number of top-level features of the same type within the city model.

For instance, the number of buildings with updated property *creationDate* within the city model is stored in a semantic scope, while their bounding boxes are merged to form a bigger one representing their spatial scope.[1] By analysing such semantic and spatial scope, it can be determined whether these changes apply to all buildings or only to buildings within a specific region. Section 5.4.3 explains further how such scope information can be computed and stored during the interpretation process.

5.3. Defining Rules for Identifying Change Patterns

This section introduces a compact network for defining aggregative rules to identify and match change patterns, addressing Research Question RQC3 (Rule Definition for Change Patterns). The network is a type graph that stores all rules in a connected graph, thus enabling the modelling of interdependent rules without redundancy.

5.3.1. Requirements for the Pattern Matching Process

For the purpose of defining aggregation rules that can identify patterns among changes in the graph representations of CityGML documents, thereby reducing the number of changes to report and at the same time forming new ones with enhanced semantic levels, the following key technical requirements must be fulfilled:

1. **On-the-fly Typing**: Given that the CityGML data model is rich with semantic information, their changes are correspondingly typed and attributed. These attributes primarily originate from the edit nodes and their directly linked nodes in the graphs, as explained in Section 4.6. The types can be taken from the class hierarchy depicted in Figure 5.3. However, in this research, input changes for the interpretation process are provided as a sequence of arbitrary length, and graphs do not preserve any specific order among sibling elements when retrieved from the database. This leads also to the mixed order of their corresponding changes in the input sequence. A simple approach would first divide this input sequence into smaller groups of changes of the same type and then operate on each group.

[1]Alternatively, city object groups could be used to mark buildings sharing the same scope. However, this approach changes the contents of the original city models and is therefore not implemented.

This not only is inefficient as each change is processed multiple times, but it also requires the length of the input sequence to be known, which is often difficult to achieve while iterating over large and dynamic data. As a result, the ability to efficiently distinguish these changes by type and attributes on-the-fly, irrespective of their order and without the need for repeated iterations, is required.

2. **Origin Handling**: Even when two changes within a sequence share the same type, they could still belong to two unrelated objects from two distant parts of the dataset. For example, a database query may return a series of all translated surfaces in the graphs. However, these surfaces may not be part of the same building; in fact, each surface could even belong to a different building. Therefore, to differentiate them beyond their type information, their semantic context should also be employed to uncover valuable information about the original objects to which they belong.

3. **Dynamic Aggregation**: The majority of aggregation rules specify a fixed number of input changes per type that are needed to formulate new ones. For instance, the rules shown in Figure 5.2 require exactly two roof surfaces, four wall surfaces, and one ground surface. However, this information is typically not available until execution. Furthermore, this number varies significantly among city objects. For example, one building may have two roof surfaces, four wall surfaces, and one ground surface, while another building may have only one roof surface, but six wall surfaces, and two ground surfaces. Consequently, aggregation rules with dynamic input quantities must be allowed and managed accordingly.

4. **Memory Efficiency**: Graph representations of cities may become very large, leading to a potentially overwhelming number of produced changes. Additional strategies must be introduced to efficiently process all changes without the need for repeated iterations, thereby reducing the memory consumption.

Therefore, based on the strengths and limitations of the Rete networks, Petri nets, and graph transformation systems, as discussed previously in Section 5.1, this research first proposes a flexible and compact aggregative rule network for establishing rules for identifying change patterns in the graph representation of CityGML documents in one place. Subsequently, the rules are parsed and applied to the graphs, and corresponding patterns are matched during the pattern matching process.

5.3.2. Definitions

This section provides definitions and terms related to the content networks and rule networks that are employed throughout this thesis.

Content Network

The directed and attributed graph representations of CityGML documents, as employed thus far in Chapters 3 and 4, are referred to as **content networks** in this study, as they contain all substantial information about the city models. Nodes and relationships within a content network are denoted as *content nodes* and *content relationships*, respectively. Each content network holds the graph representations of both the old and new CityGML documents.

An example of such a content network is shown in Figure 5.4. For visual simplicity, this illustration replaces the *ARRAY* node and its outgoing *ARRAY_MEMBER* relationships, as seen in Figure 3.12, with the relationships *boundarySurface* between the building node and its boundary surfaces. Moreover, edit nodes representing the detected changes, such as updated properties, are attached to both the old and new graph representations, as explained in Section 4.6. However, for clarity, this figure only shows one of these graph representations with connected edit nodes, as will be shown later in Figure 5.10.

Rule Network

A **rule network** is a directed acyclic and attributed graph serving as a centralized place for defining all rules required for pattern matching. Operating as a special *type graph*, it describes the characteristic behaviours associated with various types of changes, allowing the dependencies between rules to be explicitly captured. Therefore, the subgraphs, which contain all change nodes within the corresponding content network, are considered *instance graphs* of this rule network.

Organizing rules for patterns among changes poses a major challenge due to their high interdependence: one rule may rely on the outcomes of others. For example, the rule for determining whether a building's roofs have been raised relies on the results of rules assessing whether roof surfaces have been moved upwards, and whether wall surfaces have been enlarged vertically. These rules, in turn, depend on the rules examining whether each corresponding surface geometry underwent a translation or size change by the same amount. Absence of an effective model to handle such complex interdependence can result in substantial redundancy and an excessive number of duplicate rules, leading to an unsustainable representation of that rule set.

The rule network's nature as a type graph eliminates this problem by ensuring each type of changes is presented **only once** in the network. This approach minimizes redundancy, thereby allowing all interdependent pattern rules to be succinctly and efficiently described within a single rule network. In this thesis, rules may be interpreted in a non-deterministic order, but the final results remain consistent.

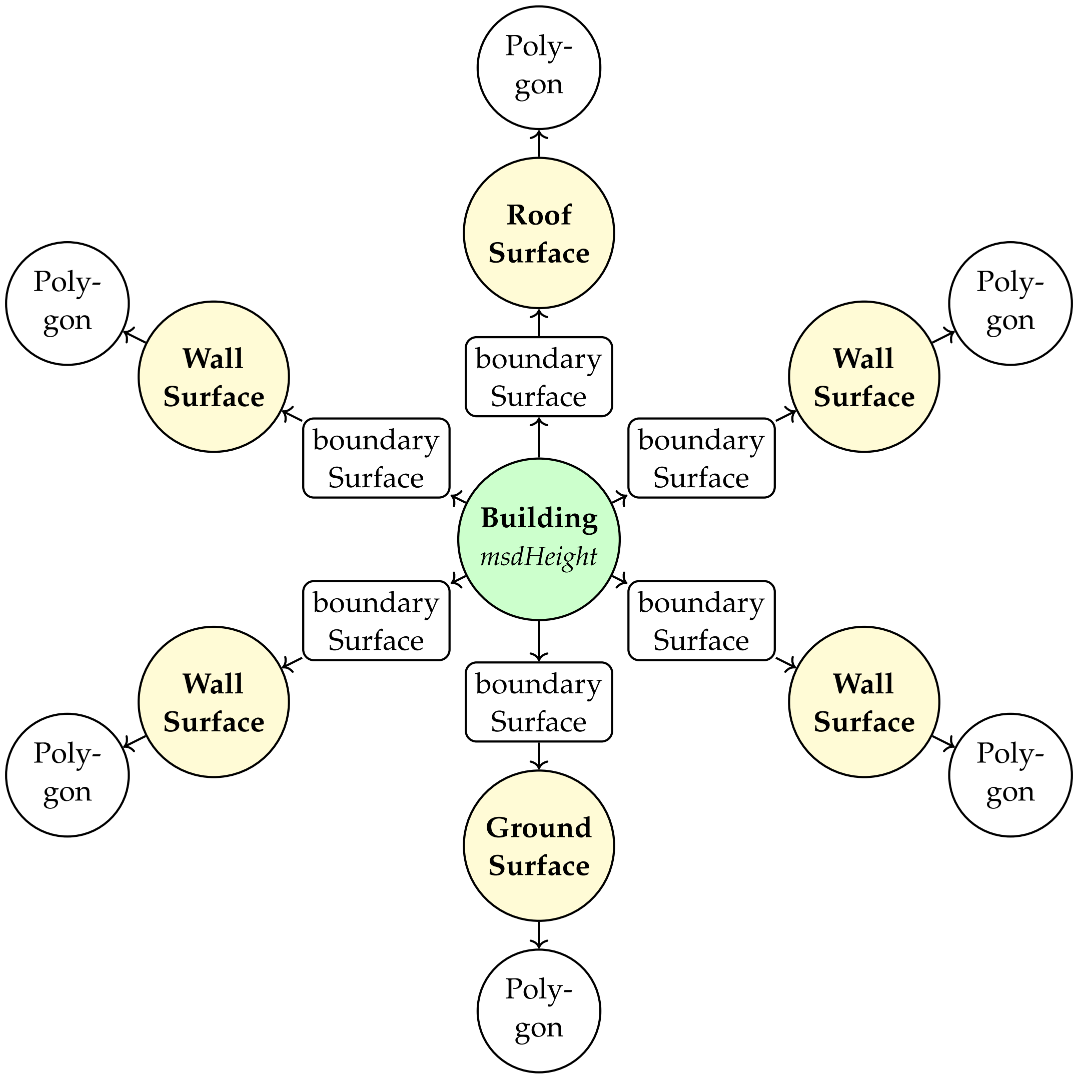


Figure 5.4.: A content network of a simplified building model, representing multiple one-to-many relations between a building and its boundary surfaces.

Nodes and relationships within a rule network are denoted as *rule nodes* and *rule relationships*, respectively. A rule node can represent a type of edit nodes listed in Section 4.6, a change class depicted in Figure 5.3, or an interpreted change at a later stage. Figure 5.5 illustrates an example of such a rule network applicable to the content network previously presented in Figure 5.4.

Change Network

Similarly, the network formed by changes, including both the edit nodes and interpreted changes, is referred to as the **change network**.

In this research, to enable further queries and analyses, all changes, including lower-level base changes and all higher-level interpreted ones, are directly linked to the graph elements from which they originate. The content networks of CityGML documents, as explained in Chapter 3, are directed, with content relationships pointing from higher semantic level content nodes to lower semantic level ones, such as from a building node to its boundary surfaces. In contrast, the pattern matching process operates in a bottom-up manner, aggregating lower-level changes to formulate higher-level ones. New, interpreted changes are connected to preceding changes from which they are derived. As a result, the direction of the change network is opposite to that of its associated content network.

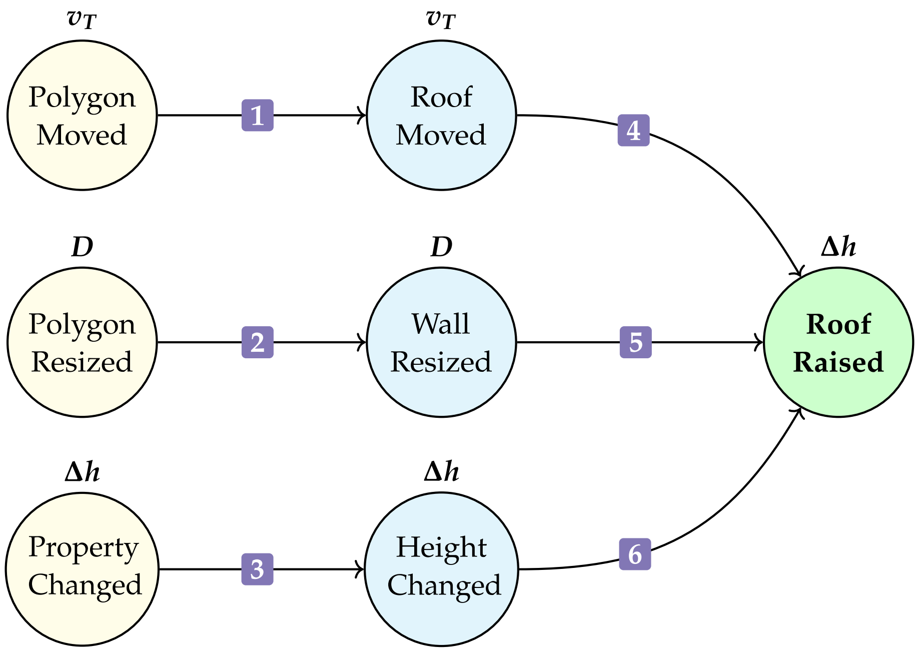
5.3.3. Properties of Rule Nodes

Each rule node has the label *RULE* and contains a list of properties. An overview of these properties is given in Table 5.1.

Table 5.1.: An overview of properties available in rule nodes.

Property	Description	Example value	Required
changeType	The type of changes represented by this rule node.	*PolygonMoved*	Yes
calcScope	Directive to calculate scope over *ChangeType* and a set of properties.	$\{\Delta h\}$	No
join	Boolean conditions for joining incoming rule relationships.	$r_1 \land (\neg r_2 \lor r_3)$	No

These properties are described further in the following sections.



Rule Relationship Properties

ID	Next content type	Conditions	Propagate	Weight
1	*RoofSurface*	*true*	$\{v_T\}$	1
2	*WallSurface*	*true*	$\{D\}$	1
3	*Building*	*true*	$\{\Delta h\}$	1
4	*Building*	$v_T = (0, 0, \Delta h)$	$\emptyset$	$*$
5	*Building*	$D = (0, 0, \Delta h)$	$\emptyset$	$*$
6	*Building*	$\Delta h > 0$	$\{\Delta h\}$	1

Figure 5.5.: An example of a rule network for the content network shown in Figure 5.4. Each rule node is depicted with its label and (propagated) properties. Rule nodes within the same semantic level are highlighted in the same colour. Each relationship is assigned with a set of predefined properties required for describing the pattern rules.

Property *changeType*

Each rule node is associated with the type of the changes it represents. These changes can be edit nodes that are attached directly to the source graph elements where the changes were detected, or they can represent higher-level changes derived from these edit nodes. In this study, the information about this change type is stored in the property *changeType* of each rule node, rather than its label. This approach minimizes the label set of rule nodes for indexing and provides the flexibility in naming and referencing rule nodes.

In the illustration provided in Figure 5.5, each depicted node is a rule node. The rule nodes on the left-hand side denote edit nodes that were identified after the matching process is complete, as explained in Chapter 4. Positioned at one semantic level above the edit nodes, the rule nodes in the middle represent changes at a higher abstraction level, condensing more semantic information. Such additional semantic information is gained from the semantic context extracted from the content network while traversing. For example, a change representing a translated polygon becomes a translated roof surface once it acquires context information about the type of the boundary surface containing that polygon. Lastly, the rule node on the right obtains the highest semantic level of this change pattern.

This semantic ordering of the rule network, distinguishing between the low and high-level rule nodes, is dictated by the direction of the rule relationships interconnecting these nodes. In this arrangement, these relationships point from lower-level rule nodes towards higher-level rule nodes, establishing a hierarchical structure. Source nodes, those without incoming relationships like the edit nodes on the left of Figure 5.5, are regarded as residing at the lowest semantic levels. Conversely, sink nodes, those without outgoing relationships like the rightmost node of Figure 5.5, are positioned at the highest semantic level within the pattern they represent. As a result, these highest-level rule nodes are often used to refer to their associated patterns.

Property *calcScope*

In addition to the mandatory property *changeType*, additional optional properties, such as *calcScope*, can be assigned. The property *calcScope* serves as a directive for the pattern matching process to calculate and evaluate the scope over the current *changeType* and its given properties.

For example, the directive *calcScope* over the value 'Δh' for the change type *RoofRaised* specifies that the pattern matching process must examine whether all instances of this type share the same height offset, and if it is the case, calculate the semantic and spatial extent (such as the bounding box) of all matching instances.

The results of the scope computation process can be local, clustered, or global scope, with a corresponding bounding box covering the scope's spatial extent, as modelled in Section 5.2. As scope calculation is computationally expensive, it is often applied only for top-level features in this research.

Property *join*

In Figure 5.5, the rightmost rule node requires all incoming conditions to be fulfilled. Each of these conditions can be vastly different and are evaluated based on its own propagated knowledge acquired thus far, such as v_T, D, and Δh of the rule nodes in the middle. However, as the rule relationships converge at the rightmost rule node, their knowledge store can also be shared, allowing for more comprehensive join conditions. Such are given in the property *join* of the rule node where rule relationships converge.

In addition, the join conditions also allow for the evaluation of more complex boolean expressions among converging rule relationships, such as $r_1 \land (\neg r_2 \lor r_3)$, where r_1, r_2, and r_3 are the relationship names representing their boolean values. These relationship names are described in Section 5.3.4.

5.3.4. Properties of Rule Relationships

Rule nodes are connected within the network using directed rule relationships. All rule relationships have the type *AGGREGATED_TO*, reflecting the aggregative nature of these pattern rules, and a number of properties. These properties are summarized in Table 5.2 and described in the following sections.

Property *nextContentType*

The property *nextContentType* guides the rule interpreter on where to navigate next within the content network. The direction of traversal is opposite to that of the content network, moving from lower semantic level content nodes to higher ones. For example, as shown in Figure 5.5, the target content types of the three rule nodes on the left are *RoofSurface*, *WallSurface*, and *Building*, while the target content type of all three rule nodes in the middle is *Building*.

This next target content type does not necessarily need to be the type of the next content node adjacent to the current one. It can be any node along the paths in the direction of traversal. One major advantage of this approach is that it does not require prior knowledge about the exact structure of the content network. As long as the type of the next 'check point' is known, the rule interpreter shall attempt to search for a path up to that node.

Table 5.2.: An overview of properties available in rule relationships.

Property	**Description**	**Example value**	**Required**
nextContentType	The type of the target content node used by the interpreter while navigating within the content network.	*Building*	Yes
searchLength	The maximum length the interpreter can traverse to reach the target node determined by *nextContentType*.	5	No
notContains	The content type that should not be encountered while traversing to the target content node.	*BuildingPart*	No
name	Assign a name to the start rule node of this relationship referenced by the *join* operations.	*ruleHeight*	No
propagate	Directive to propagate the properties of the start rule node of this relationship to the next one.	v_T	No
conditions	Boolean conditions required for the creation of the next interpreted change.	$\Delta h > 0$	No
scope	Examine whether the changes represented by the start rule node of this relationship belong to a scope.	*global*	No
weight	The number of changes represented by the start rule node required for the creation of the next change.	7	No

Property *searchLength*

When searching for the next content nodes based on their types, the additional information *searchLength* can be provided to limit the maximum number of relationships that can be traversed sequentially in a depth-first search before the target content nodes are reached. This is useful in cases where there are multiple content nodes of the same type exist along the path between two given nodes. By setting a limit on the search length, the targeted content node can be found. Notably, a search length value of 0 indicates that the next content node is the same as the current one, while the value 1 indicates that the next content node is the direct predecessor of the current content node within the content network.

Property *notContains*

During the traversal to the next content nodes, it is often required to exclude paths that contain a specific content type. This can be specified by the property *notContains*. If a content node of the type specified by *notContains* is encountered during traversal, the interpreter excludes the current content node and its change from the pattern. This approach is employed to handle cases where multiple possible paths may exist in the content network between the current content type and the target content type. For example, as allowed by the CityGML data model, a boundary surface may belong to a building or a building part. Relying solely on the property *targetContentType* is insufficient to differentiate boundary surfaces of a building or a building part. This problem can be solved by employing the additional property *notContains*.

Property *name*

In complex change patterns, a rule node may require multiple preceding rule nodes. For example, the rightmost rule node in Figure 5.5 requires three rule nodes in the middle. In such cases, the rule relationships outgoing from these middle rule nodes converge at the rule node on the right. To facilitate further assessment of complex patterns, such as evaluating all conditions for the creation of the next interpreted change, unique names can be assigned to the all converging relationships and their corresponding preceding rule nodes. All rule nodes with converging outgoing rule relationships can access the names and properties of each other. This is particularly useful when defining and evaluating join conditions, as described previously in Section 5.3.3.

Property *propagate*

The rule network allows for propagating variable and property values from lower-level rule nodes to the next higher-level node. This process ensures that the knowledge gained while interpreting the change patterns through the hierarchy is preserved, preventing the loss of important observations and findings while traversing through the rule network. For example, as illustrated in Figure 5.5, the knowledge variables v_T, D, and Δh are first obtained in the left rule nodes. These values are then further propagated to the subsequent rule nodes in the middle and on the right, as specified by the property *propagate* of the corresponding rule relationships.

Property *conditions*

The property *conditions* specifies the criteria that must be met for the creation of the next interpreted change. These conditions are logical expressions assessed against properties in the corresponding change nodes. Properties propagated from previous nodes, or those shared across converging rules for a *join* condition, can be used to formulate and evaluate these conditions. In the absence of any specific conditions, the default value *true* is used.

Property *scope*

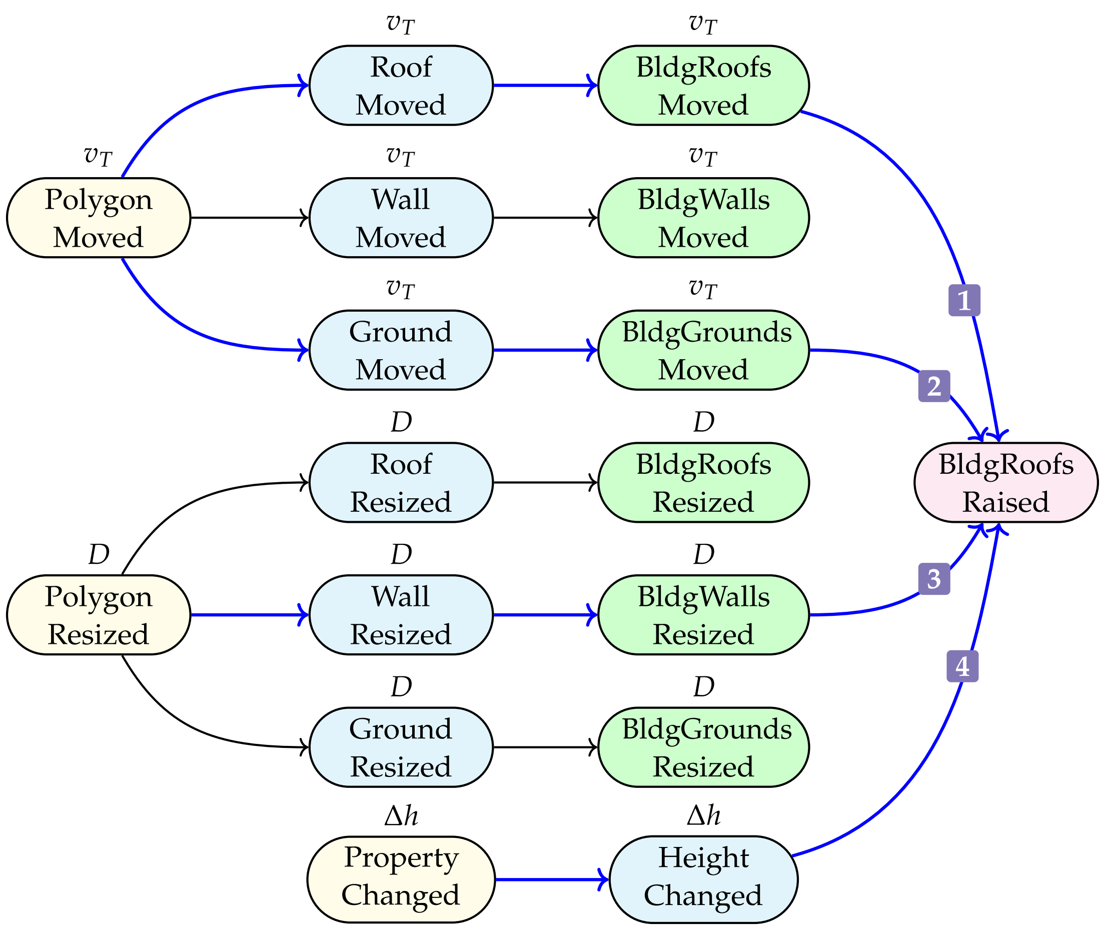
When the directive *calcScope* of the previous rule nodes is activated and a scope has been calculated across all corresponding changes, the resulting scope can be assessed using the property *scope*. For instance, the value *global* considers only global or systematic changes. Global changes related to top-level features are typically attached to the *CityModel* node, as it serves as the source node of the entire graph representation of the CityGML document. Other scope values include *clustered* and *local*.

Property *weight*

The weight or multiplicity of a rule relationship dictates the required number of change occurrences corresponding to the preceding rule node for the creation of the next interpretation node. A rule node with multiple incoming relationships can only be activated if all preceding rule nodes, when required by the *join* conditions, have accumulated a sufficient number of change instances. For example, as depicted in Figure 5.5, the rule node *RoofRaised* can only be activated if all required occurrences of *RoofMoved*, *WallResized*, and *HeightChanged* exist.

The weight can be assigned a specific value or a placeholder '$*$' if the value is not yet known. For instance, given that each building can have a different number of wall surfaces, the weight of the rule relationship between *WallResized* and *RoofRaised* is initially set to '$*$'. This placeholder is updated with a concrete value by the rule interpreter during runtime. The same can also be applied to other types of boundary surfaces or any other content types.

Figure 5.6 illustrates an extended rule network based on the rule network given in Figure 5.5. This enhanced network, utilizing many of the aforementioned node and relationship properties, can be employed to identify both translations and size changes of all boundary roof, wall, and ground surfaces of buildings. The same approach can be applied to define pattern rules for surfaces of building parts.



Rule Relationship Properties

ID	Next content type	Search Length	Conditions	Name	Weight
1	*Building*	0	$v_T.z > 0$	*roofs*	1
2	*Building*	0	$v_T.z \neq 0$	*grounds*	1
3	*Building*	0	$D.z > 0$	*walls*	1
4	*Building*	0	$\Delta h > 0$	*height*	1

Join at node *BldgRoofsRaised*: $height.\Delta h = walls.D.z = roofs.v_T.z - grounds.v_T.z$

Figure 5.6.: A rule network for detecting raised roofs of buildings, which is extended from that of Figure 5.5 by also considering translation and size changes of all boundary surfaces. The green rule nodes represent surface translation or size changes of the same v_T or D within a building.

5.3.5. Rule Notations in Cypher

The rule nodes and relationships, as described previously in Sections 5.3.3 and 5.3.4, can be combined and utilized by users to define their own rule networks. This section explains how such rule networks can be implemented in Cypher using the node and relationship definitions introduced above.

Table 5.3 gives an overview of rule nodes, relationships, along with their labellings and contents, expressed in Cypher notations. Node labels and relationship types are managed in Neo4j using thematic indexes, allowing for efficient retrieval of rule nodes and relationships based on their labelling, as described in Section 6.2.

Table 5.3.: Rule nodes and relationships in Cypher notations.

Rule Element	**Cypher Notation**
Labelling	
Node label	*RULE*
Relationship type	*AGGREGATED_TO*
Node Properties	
changeType	*change_type*
calcScope	*calc_scope*
join	*join*
Relationship Properties	
nextContentType	*next_content_type*
searchLength	*search_length*
notContains	*not_contains*
name	*name*
propagate	*propagate*
conditions	*conditions*
scope	*scope*
weight	*weight*

Listing 5.1 shows an example of a rule network for detecting change patterns on buildings' identifiers in Cypher. This simple network consists of a single path, containing three rule nodes that are sequentially connected by two rule relationships. It starts with the lowest-level change in an object's identifier property, then verifies if that object is a building, and finally checks if all buildings' identifiers have been updated. Thus, the last node is at the highest semantic level of this pattern.

Rule nodes and relationships can be described using the following Cypher notations:

1. **Rule node**: Denoted as (*n*:*RULE* {*properties*}), where *RULE* is the label of the node, *properties* represents a list of the node's properties, and *n* serves as a reference to the node for reusing.
2. **Rule relationship**: Denoted as ( )-[*r*:*AGGREGATED_TO* {*properties*}]->( ), where *AGGREGATED_TO* is the type of the relationship, *properties* represents a list of the relationship's properties, and *r* serves as a reference to the relationship (often not utilized).

```
// Pattern in id of buildings
MERGE (updated_property:RULE { // unique node names within the query
  change_type: 'UpdatedProperty' // unique change type within the database
})-[:AGGREGATED_TO {
  next_content_type: 'Building', // the label of the next content node
  search_length: 0, // the updated property is inside the content node
  conditions: 'NAME === "id"', // JavaScript syntax
  propagate: 'NAME', // propagate only property name
  weight: 1
}]->(updated_building_id:RULE {
  change_type: 'UpdatedBuildingId',
  calc_scope: 'NAME' // calculate scope over this property
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel', // attach scope nodes to CityModel node
  scope: 'global'
}]->(global_updated_building_ids:RULE {
  change_type: 'GlobalUpdatedBuildingIds'
})
```

Listing 5.1: An example Cypher query for defining pattern rules on the identifiers of buildings.

In Cypher, the clause *MERGE* functions as a combination of *MATCH* and *CREATE*. It determines whether a given graph pattern exists; if so, the graph entities are reused, otherwise, they are created. This is particularly useful for creating unique nodes and relationships that should exist only once, as in the case of the rule networks.

Additional guidelines and examples on constructing more complex rule networks using real-world datasets can be found throughout Chapter 7. A comprehensive list of all change patterns employed in the implementation of this thesis is available in Listing B.1.

5.4. Matching Change Patterns

During pattern matching, change patterns in a content network are matched against predefined rules in a rule network. Recognized patterns are represented as additional **interpretation nodes**, which act as interpreted changes linked to their content nodes, similar to any other changes in the change network. For instance, a *PolygonMoved* node is attached to a polygon node, while a *RoofRaised* is attached to a building. These interpretation nodes are central to the expansion of the change network.

Given an initial change network that contains edit nodes and other base changes, a content network, and a corresponding rule network with predefined pattern rules, the method *matchPatterns(changes, contents, rules)* for matching patterns using these components is outlined in Algorithm 8. Addressing Research Question RQC4 (Matching Change Patterns) and Research Question RQC5 (Managing Temporary Data), this method employs several strategies, which are explained in the following sections.

5.4.1. Successive Processing of Changes

Interdependence exists among change patterns, where a change can be both the outcome of previous lower-level changes and the input to subsequent higher-level changes. Therefore, the pattern matching process must avoid unnecessary repeated processing of these changes, while ensuring no changes are left unchecked.

To address this, the algorithm employs a First In First Out (FIFO) queue that functions like a conveyor belt in an assembly line. Initially, the queue is filled with all edit nodes and base changes (refer to Line 1). These are then sequentially removed from the queue for processing (refer to Line 3). When a pattern is identified and an interpreted change is created, it is added to the end of the queue (refer to Line 31). As a result, the changes stored in this queue follow an ascending semantic order, with newer changes possessing more semantic content.

Figure 5.7 illustrates the use of this queue during the pattern matching process. The algorithm successively assesses changes from the queue against a number of criteria: type check (Line 5), condition check (Line 11), origin check (Line 13), scope check (Line 14), and join check (Line 27), which are described as follows:

1. **Type Check**: Changes are typed and attributed. Only changes that are utilized in the rule network, indicating their role as a component of a pattern, are processed.
2. **Condition Check**: For a change to be aggregated, it must satisfy the logical conditions specified in the corresponding property in the rule relationships.
3. **Origin Check**: Even when changes are of the same type, they may originate from different, unrelated objects. The origin check ensures that only changes

Algorithm 8: Method for pattern matching *matchPatterns(changes, contents, rules)*

```
Input   : The networks changes, contents, and rules
Outcome: Interpretation nodes linked to content nodes for detected patterns
queue ← all edit nodes and base changes in changes
while queue is not empty do
    change ← queue.dequeue()
    rule ← rules.findRuleBy(change)
    if rule = null then continue
    if rule.hasProperty("calcScope") then
        scope ← create or retrieve a scope node based on change and rule
        attach change to scope, and update semantic and spatial extent of scope
    end
    foreach outgoing relationship rel of rule do
        if change does not satisfy rel.getProperty("conditions") then continue
        nextContent ← find next content node in contents based on
          rel.getProperties("nextContentType", "searchLength", "notContains")
        if nextContent = null then continue
        if rel.hasProperty("scope") but the scope does not satisfy then continue
        nextRule ← rel.getEndNode()
        nextChangeType ← nextRule.getProperty("changeType")
        memory ← find a memory node linked to nextContent with nextChangeType
        if memory = null then
            memory ← create a memory node for nextChangeType
            initiate a count of 1, and a capacity based on rel.getPropety("weight")
            attach memory to nextContent
        else
            increase the count value of memory by 1
        end
        attach change to memory
        if rel.hasProperty("propagate") then copy properties of change to memory
        if memory does not satisfy nextRule.getProperty("join") then continue
        nextChange ← create next change node and copy properties from memory
        attach all nodes linked from memory to nextChange
        attach nextChange to nextContent
        queue.enqueue(nextChange)
    end
end
```

sharing the same semantic context within the graphs are processed together. This is further explained in Section 5.4.2.

4. **Scope Check**: For changes that are part of a larger pattern, such as individual building translations that contribute to a systematic translation of the entire city model, their scope can be queried and analysed. A scope can be computed based on both the semantic and spatial information shared among changes. This is further detailed in Section 5.4.3.

5. **Join Check**: Lastly, when a pattern relies on several component rules, their conditions can be evaluated based on the join conditions specified at the pattern rule node, where the rule relationships of this pattern converge.

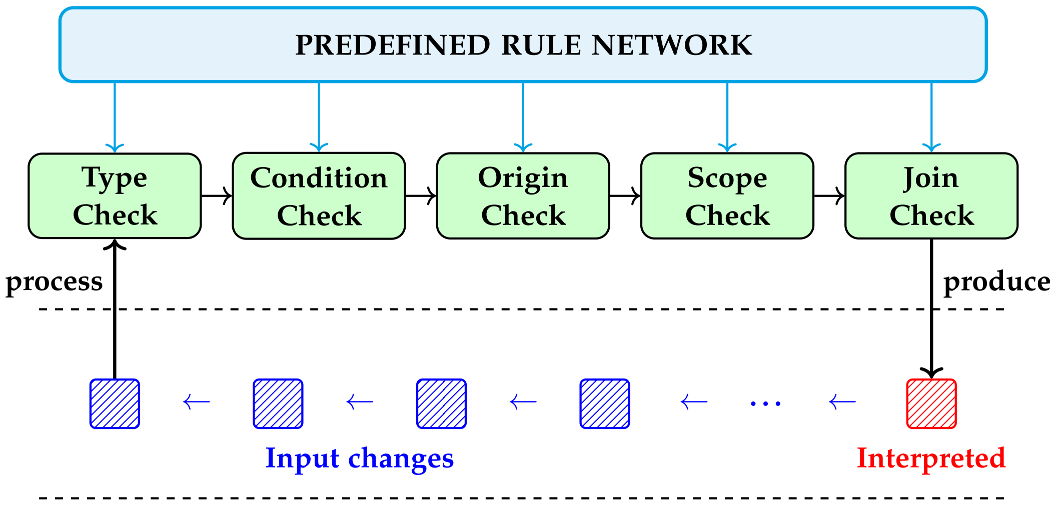


Figure 5.7.: An overview of the pattern matching algorithm. It employs a queue to process all edit nodes and base changes (blue). Changes that pass all checks are aggregated into a new interpreted change (red), which are added to the end of the queue for further processing.

These evaluations are conducted using the properties available in rule nodes and relationships, namely: *changeType*, *calcScope*, and *join*, as well as *nextContentType*, *search-Length*, *notContains*, *propagate*, *conditions*, *scope*, and *weight*, as introduced in Sections 5.3.3 and 5.3.4, respectively. Only changes that successfully pass all the aforementioned checks are aggregated. Once a sufficient number of changes have been aggregated, a new interpreted change is produced from these components. This process employs **aggregative memory nodes**, which are explained in Section 5.4.4. For Algorithm 8 to terminate, the given rule network must be acylic.

5.4.2. Handling of Semantic and Graph Origin of Changes

One of the key concepts of the pattern matching algorithm is its ability to differentiate changes based on the semantic context of their corresponding content node within the content network. The *origin* of a content node is a set that includes the node itself and all its ancestors in the content network. This origin information is essential to distinguish changes that are similarly typed and attributed. For example, in an input sequence of changes, such as $(r, w, w, r, r, h, \ldots)$, where the change types are *RoofMoved*, *WallResized* and *HeightChanged*, it is unknown whether these changes refer to the boundary surfaces of the same building or several different ones. Incorrectly aggregating parts from unrelated origins can lead to incorrect outcomes for the current rule and may prevent other rules from triggering due to missing required parts.

Therefore, the origin check is employed as an additional semantic safeguard, alongside the type check, to ensure accurate aggregation of changes. This is related to the problem of finding the Lowest Common Ancestor (LCA) (Aho et al., 1976; Harel & Tarjan, 1984) in graph theory, which involves searching for the closest shared ancestor of two given nodes in a directed acyclic graph. For instance, as shown in Figure 5.8, given the changes v_1, v_2, and v_3 and their corresponding content surface nodes s_1, s_2, and s_3. The LCA of s_1 and s_2 is the content building node b, while the LCA of b and s_3 is the content city model node. By leveraging the direct linkage between changes and content nodes (depicted in orange), the change v_3 can be distinguished from the changes v_1 and v_2, even though they all refer to the same type of changes.

Furthermore, the information about LCA indicates that the interpreted change C_1, which is associated with the building node b, is produced by aggregating v_1 and v_2. Similarly, the interpreted change C_2, which is associated with the city model node, is produced by aggregating C_1 and v_3. This applies to all LCAs for two given changes. The interpretation of changes at higher levels only considers interpreted changes at the next lower level, or edit nodes or base changes in the absence of such an interpretation.

5.4.3. Managing Scopes of Changes

In addition to patterns that rely on multiple types of component changes, such as the change *RoofRaised*, which is derived from *RoofMoved*, *WallResized*, and *HeightChanged*, a large number of patterns can also be derived from component changes of the same type. For instance, a systematic *RoofRaised* among buildings within the city model. To detect these patterns, the scopes of changes are computed.

A scope can be calculated based on the semantic information of changes, their spatial extent, or a combination of both. The semantic information used to compute scopes includes the types of changes, along with their thematic properties. On the other

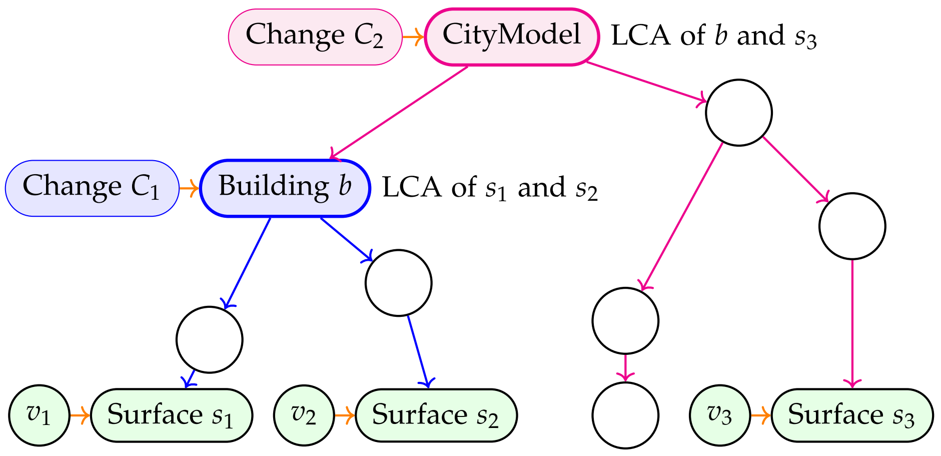


Figure 5.8.: An example of origin comparison among changes based on their corresponding content nodes. The old content network is used unless changes are attached to the new content nodes only. The change nodes v_1, v_2, and v_3 are associated with the content surface nodes s_1, s_2, and s_3 (green). The Lowest Common Ancestor (LCA) of s_1 and s_2 is located at the content building node b (blue), while the LCA of b and s_3 is located at the content city model node (red). For visual clarity, the relationships between changes v_1, v_2, v_3, C_1, and C_2 are not shown.

hand, the spatial information employed to compute scopes includes various geometric details, such as positions, orientations, footprints, and bounding boxes. For example, a systematic change on building identifiers has a semantic scope, as it involves only building objects and their thematic properties, while a systematic translation of all geometries in the city model has a spatial scope, as it involves spatial positions of geometric objects. Additionally, both semantic and spatial information can be combined to compute scopes. Examples include lifted buildings within a certain bounding box, or renovated buildings along a street.

When a change is part of a scope, the pattern matching process creates an explicit connection between the change and its scope node. This allows for efficient retrieval of changes within a scope, as well as effective removal and insertion of member changes within that scope. Scope nodes are attached to the content nodes that are the LCAs of all content nodes associated with these changes. These are often a top-level feature node (such as a building) for changes related to its sub-elements, or the city model node for changes related to the top-level features themselves. In this study, the computation for the majority of scopes is performed over top-level features. Therefore, the city model node is often used to retrieve all these scope nodes.

While a scope may cover a large number of changes, a single change may be assigned to multiple scopes. Scopes associated with a specific change are distinguished by the information they carry, including the type of the change, as well as semantic and spatial properties employed for the calculation of the scopes.

Scopes are evaluated based on their coverage. They are categorized as *global* if all changes are covered, *local* if only a single change is affected, or *clustered* for all other cases in between. The value *clustered* can be further specified to denote scopes of changes within a certain bounding box, polygon area, or radius. During the pattern matching process, as changes are being added to the existing scope, the semantic and spatial extent of these scopes must be updated to include the newly added change (refer to Line 8).

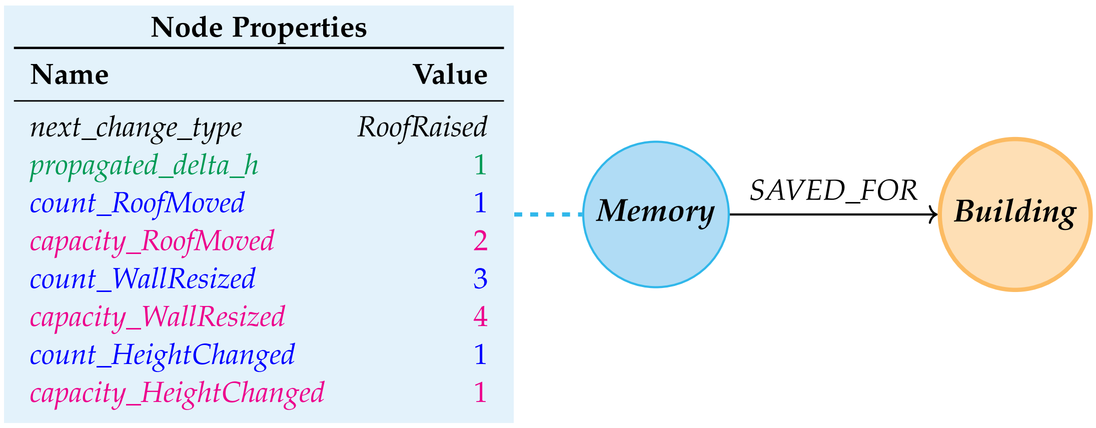
5.4.4. The Use of Aggregative Memory Nodes

The method *matchPatterns(changes, contents, rules)*, as outlined in Algorithm 8, relies greatly on the use of **aggregative memory nodes** for the implementation of many of its core concepts. A memory node serves as an auxiliary node, attached to a content node, and is responsible for storing temporary information required for the creation of the next interpreted changes.

Since a content node can be associated with many change patterns, multiple memory nodes can exist for one single content node. Therefore, memory nodes linked to the same content node are differentiated from each other by the type of the next changes they aim to create once sufficient components have been collected. For example, for the change pattern given in Figure 5.5, a memory node is created and attached to a building node, collecting the information necessary for the creation of the interpreted change *RoofRaised*.

In addition to the change type, memory modes also incorporate a counter that keeps track of the number of component changes per type collected thus far. Each time a change of a specific type is encountered and accepted, the counter is incremented by 1 (refer to Line 23). This counter continues to increase until it reaches a predetermined threshold, which is the capacity specified by the memory node for a specific change type. Once all component changes have reached their respective capacities, the next change node is then created. Figure 5.9 illustrates such a memory node utilized for matching the patterns defined in Figure 5.5.

The use of memory nodes is similar to that of the Rete networks (Forgy, 1982) and can eliminate repeated iteration by processing changes on the fly. However, in contrast to classical Rete networks, the proposed method does not store entire objects in its memory. Instead, the algorithm first distinguishes changes based on their types and attributes, then updates their counters accordingly. Moreover, at the start, the memory


Node Properties

Name	Value
next_change_type	*RoofRaised*
propagated_delta_h	1
count_RoofMoved	1
capacity_RoofMoved	2
count_WallResized	3
capacity_WallResized	4
count_HeightChanged	1
capacity_HeightChanged	1

Figure 5.9.: An illustration of a memory node used for matching the pattern rules for *RoofRaised*, as defined in Figure 5.5. Propagated properties from previous change nodes are shown in green, the number of acquired component changes thus far in blue, and their required number of occurrences in red. In this example, the memory node has acquired 50 % of *RoofMoved*, 75 % of *WallResized*, and 100 % of *HeightChanged* instances required.

is empty and only expanded as new rules and changes are encountered. This avoids the worst-case memory consumption of Rete networks, where the working memory could hold all input objects at runtime.

The capacity used in memory nodes is typically determined by the property *weight* found in rule relationships (refer to Line 20). However, when this weight is unknown, as indicated by its value '$*$', it is substituted with a concrete value during execution. To achieve this, the interpreter first searches 'upwards' in the content network for the next content node that matches the content type specified in the rule relationship. It then traverses all paths 'downwards' until a non-deleted content node specified by the previous rule node is found. The placeholder is subsequently replaced with the total number of paths reached. For instance, while processing the rule relationship between *WallResized* and *RoofRaised*, the interpreter searches for a building node, as its type is specified as the next content type of the rule relationship. From this building node, all paths to wall and roof nodes are counted.

As changes are being acquired by a memory node, they are temporarily linked to the memory node until all criteria are met to generate the next change, at which point these temporary connections are then replaced with relationships between the previous and next changes. Therefore, once interpreted, changes within a change network are interconnected, with lower-level changes pointing towards higher-level changes. Figure 5.10 shows an example of such a change network, which is the outcome of applying the pattern matching process to the content network depicted in Figure 5.4 and the rule network shown in Figure 5.5.

In this example, the algorithm starts with the changes *PolygonMoved*, *PolygonResized* and *PropertyChanged* at the lowest level and gradually propagates 'upwards' in the content network until a content node with a wanted type is encountered, such as the path *PolygonResized* $\rightarrow$ *Polygon* $\rightarrow$ *WallSurface*, where *WallSurface* is required by *WallResized*. Once all criteria have been fulfilled, a new interpretation node *WallResized* is created. The propagation proceeds until a building or a city model node (not shown) is reached, the latter of which is often associated with global or systematic change patterns. Thus, the pattern matching algorithm is an aggregation process, where changes of lower semantic levels are aggregated to produce new changes of higher semantic levels.

Additionally, memory nodes also serve as a repository for properties propagated from previous change nodes, which are required for evaluating the conditions for the creation of the next change. The decision on which properties are propagated and from which rule node is dictated by the corresponding property *propagate* in the rule relationships, as explained in Section 5.3.4. These properties remain in the memory node until the next change is created, at which point they are transferred to the new change node. For the rightmost rule node in the rule network given in Figure 5.5, its associated memory node stores the property Δh, which is propagated from the preceding rule node *HeightChanged*. Upon the creation of the next change node *Roof-Raised*, it will acquire the property Δh from the memory node. This ensures a consistent propagation of knowledge gained during the interpretation process.

A comparison of the methods for pattern matching and its rule network, as utilized in this thesis, with the concepts previously mentioned in Section 5.1 is summarized in Table 5.4.

Table 5.4.: A comparison between the proposed pattern matching methods and related concepts with respect to the key requirements described in Section 5.3.1.

	On-the-fly Typing	Origin Handling	Dynamic Aggregation	Memory Efficiency
Rete networks[1]	●	×	×	×
Petri nets[1]	◐	×	×	●
Graph transformation[1]	◐	●	×	×
Proposed pattern rules	●	●	●	●

× Not applicable ◐ Applicable if typing is enabled ● Applicable

[1] Original publication is considered. Some variants may differ.

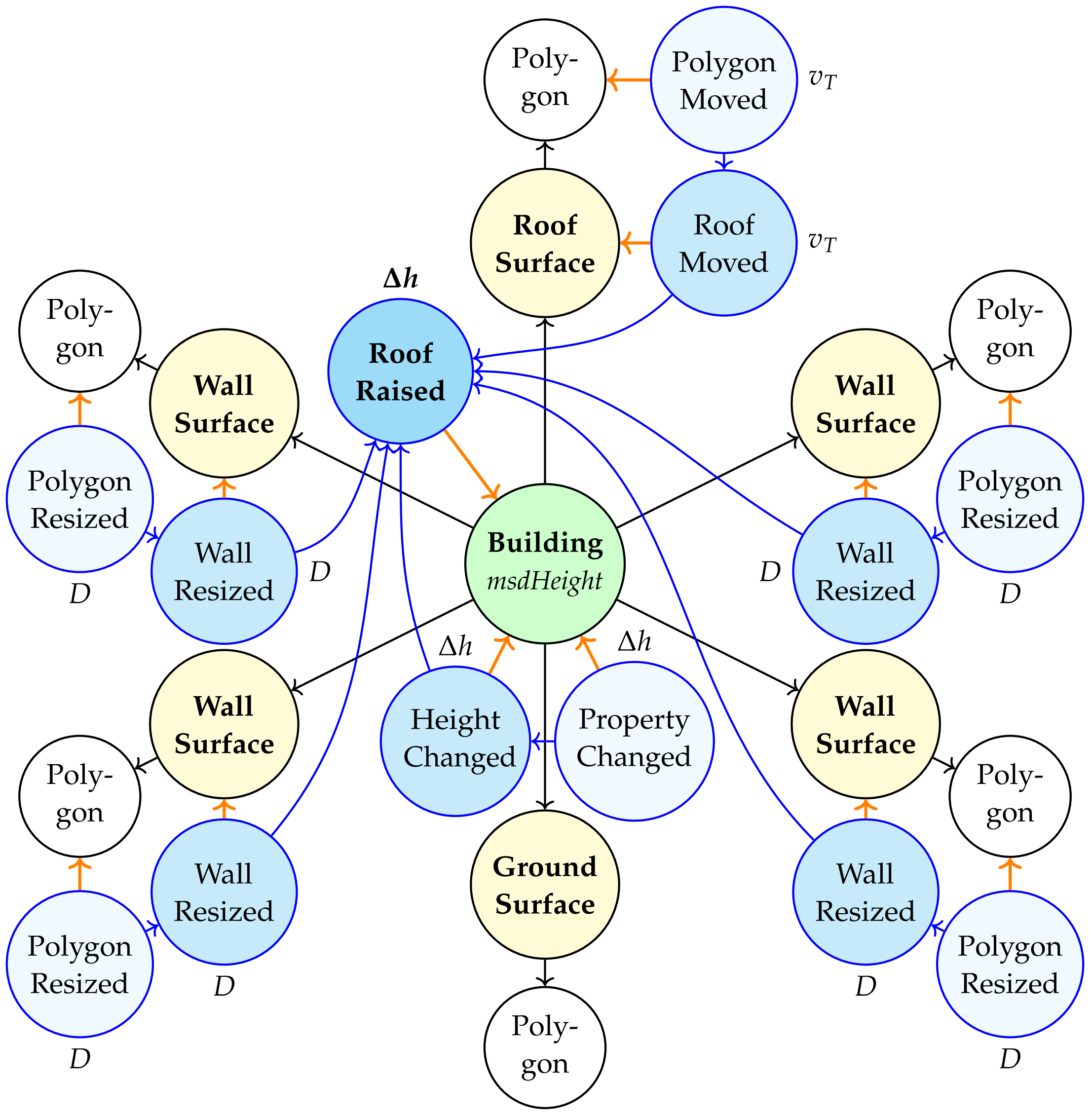


Figure 5.10.: An example of the results (blue) of the pattern matching process based on the content network shown in Figure 5.4 and the rule network shown in Figure 5.5. For visual clarity, *boundarySurface* nodes are omitted. Interpretation connections are shown in orange. The algorithm starts with the changes *PolygonMoved*, *PolygonResized*, and *PropertyChanged* at the lowest level and propagates 'upwards' in the content network until a content node with a wanted type is found, such as the path *PolygonResized* → *Polygon* → *WallSurface*. Once all criteria are met, a new interpretation node *WallResized* is created. Thus, the pattern matching algorithm is an aggregation process.

5.5. Change-Stakeholder Analysis

The interpretations derived from change patterns, as discussed in Section 5.4, provide crucial insights into the actual modifications across temporal versions of a city model. These interpretations, rich in semantic content, are both manageable and comprehensible to humans. However, as different groups of stakeholders perceive various types of changes differently, the next and final task is to examine which changes are relevant to which stakeholders, and vice versa, which stakeholders are interested in which types of changes. Addressing Research Question RQC6 (Change-Stakeholder Model) and Research Question RQC7 (Graph-based Change-Stakeholder Analysis), this section introduces a graph-based approach that allows for the dynamic modelling of the relevance relations between changes and stakeholders in a change-stakeholder network. These relations are then analysed using path-tracing techniques.

5.5.1. Requirements for Evaluating Change-Stakeholder Relations

The evaluation process of the relevance relations between changes and stakeholders presents several requirements and challenges, which are described as follows:

1. **Hidden Correlations**: Changes in semantic 3D city models are multifaceted and often correlated. They may all be caused by one single real-world action, or one change may induce other changes. Such correlation relations are complex and often hidden behind the data.

2. **Varying Interests**: Stakeholders have diverse and evolving interests in changes. A change that is highly relevant to a stakeholder now might lose its relevance in the future. This requires a flexible approach that allows for dynamic and adjustable modelling of changes and stakeholders.

3. **Stakeholder Roles**: In the context of Urban Digital Twins (UDTs), the term *stakeholder* can refer to real-world organisations, such as the city planning department. Each such stakeholder can have different *actor roles* and professions. For instance, a city mayor and a construction manager, both members of the city planning department, may have different responsibilities depending on the city's current policies and mandates, and hence, may be interested in different types of changes.

4. **Bidirectional Evaluation**: Many change analyses require not only the knowledge of which stakeholders are interested in the changes found in the city models, but also, conversely, a tailored list of specific types of changes that could be relevant to a given stakeholder.

Thus, these challenges are addressed in the following sections.

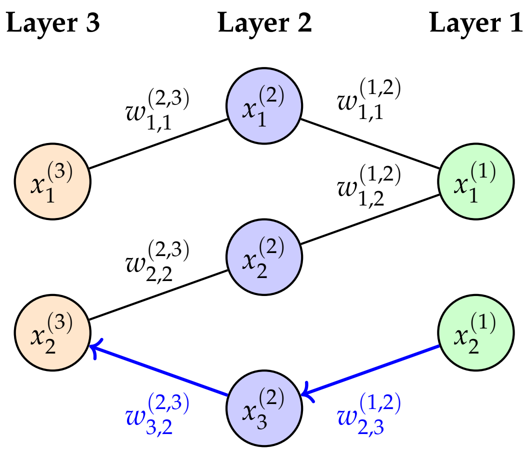
5.5.2. Change-Stakeholder Network Definition

This study introduces the concept of a **change-stakeholder network**, a multilayered semantic network that allows for the explicit description of the implicit interrelations, not only between changes and stakeholders, but also among changes and stakeholders themselves.

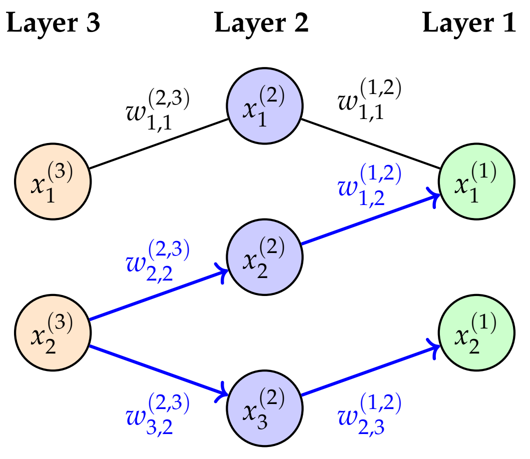
Formally, a change-stakeholder network is a graph or network $G = (V, E)$ consisting of a set V of nodes and a set E of relationships. The node set V is further divided into several partitions L_k called layers. At least two layers are required in a change-stakeholder network, namely an input and output layer that represent changes and stakeholders.

Layers are serially connected with a fixed direction, meaning a layer L_k can only have incoming connections from its previous layer L_{k-1} and outgoing connections to its subsequent layer L_{k+1}. The input layer L_1 does not have incoming relationships and the output layer L_n does not have outgoing relationships. The same can also be applied to the opposite direction.

Each node $v_i^{(k)}$ of layer L_k is assigned a weight $x_i^{(k)} \in \mathbb{R}$. A relationship $e_{i,j}^{(k,m)}$ is a directed or an undirected connection from node $v_i^{(k)}$ to node $v_j^{(m)}$ and is assigned with a weight $w_{i,j}^{(k,m)} \in \mathbb{R}$. Figure 5.11 illustrates an example of such network.



(a) Forward path tracing



(b) Backward path tracing

Figure 5.11.: An example of a bidirectional change-stakeholder network, where $x_i^{(k)}$ indicates the weight of node $v_i^{(k)}$ of layer L_k and $w_{i,j}^{(k,m)}$ indicates the weight of the relationship between nodes $v_i^{(k)}$ and $v_j^{(m)}$ of layer L_k and L_m, respectively.

In a change-stakeholder network, all nodes and relationships can be assigned with weights represented by real numbers. These values indicate the level of interest or relevance between adjacent nodes. In a directed network, a relationship weight $w_{i,j}^{(k,m)}$ denotes the relevance value of node $v_i^{(k)}$ from layer L_k towards the node $v_j^{(m)}$ of the subsequent layer L_m. In an undirected network, this weight applies to both directions. Thus, in case of different weights in each direction, directed relationships could be employed. Alternatively, positive and negative weights can also be assigned to undirected relationships to appoint a consistent traversal direction over the entire network. Infinite values can be utilized to explicitly prioritize or bypass certain nodes and relationships while traversing.

For example, relationship weights, with normalized values ranging between 0 and 1, can be used to qualitatively represent how relevant a change is to a stakeholder, with 0 signifying no interest and 1 indicating absolute interest. If the weight is 0, the relationship can be omitted, implying there exists no **semantic relation** between the nodes. In the case of *true* of *false* relations, such as whether a change is relevant to a stakeholder or not, the weight value 1 can be assigned to all existing relationships.

The change-stakeholder network is structurally similar to conventional Artificial Neural Networks (ANNs) (McCulloch & Pitts, 1943; Werbos, 1974). This similarity in structure is an intended design choice to facilitate the usage of change-stakeholder network in future deep learning applications. However, compared to common neural networks, a change-stakeholder network is additionally characterized as follows:

1. **Semantic Network**: The nodes and relationships of a change-stakeholder network are employed to model semantic concepts. While a node represents a concept or an object, such as a change or a stakeholder, a relationship describes a semantic meaning between its nodes. Thus, a node does not require incoming relationships from all nodes in the preceding layer, nor does it produce outgoing relationships to every node in the subsequent layer. A connection is set only if it represents a meaningful relation between nodes.

2. **Path-tracing Techniques**: A change-stakeholder network employs graph-based path-tracing techniques across the entire network to analyse the semantic meaning of its content. To enable forward and backward path tracing between the input and output layer, relationships must be bidirectionally traversable, i.e., by using an undirected relationship or two directed opposite relationships. These techniques are further discussed in Section 5.5.3.

3. **Layer Subdivision**: In a change-stakeholder network, a layer can be further divided into any number of sub-layers. This division allows for a more flexible modelling of semantic interrelations between concepts and objects.

Therefore, the change-stakeholder network is capable of handling data models rich in semantic information and complex interrelations. The use of a connected multilayered network provides not only an explicit representation of often implicit interactions between objects, but also an expressive and intuitive way to describe and capture the complex nature of objects and their relations.

A standard change-stakeholder network consists of four layers: Change Type Layer L_1, Reasoning Layer L_2, Actor Role Layer L_3, and Stakeholder Layer L_4. These layers are explained in the following sections. An illustration of such a network can be found in Figure 5.12. For visual simplicity, relationships between nodes within a layer are not shown in this figure.

Change Type Layer L_1

The Change Type Layer L_1 serves as the first layer of the change-stakeholder network. This layer is a type graph, where each node represents a distinct class of changes. These nodes can either represent the highest-level changes, as interpreted from the pattern matching process described in Section 5.4, or they can cover all changes within the entire change network, including edit nodes and other changes modelled in Sections 4.6 and 5.2. In the latter case, the rule network, as introduced in Section 5.3, can also act as the Change Type Layer L_1 due to them both being a type graph used to describe the interrelationships between different types of changes.

Within this layer, only nodes that capture change types of interest are connected to the next layer. Nodes that are not relevant to the subsequent layers, such as those representing auxiliary change types, are considered local only to the current layer and are not forwarded further within the change-stakeholder network. This can be achieved by marking or 'colouring' all such nodes and their corresponding relationships as local, to be excluded by the path-tracing process at a later stage. Alternatively, signed infinite weights can be assigned to these nodes and relationships to achieve the same outcomes.

Reasoning Layer L_2

Real-world objects are often interconnected through a variety of logical and physical processes. A modification to one object may also cause changes in others, or a single process can trigger changes to a number of connected objects. Thus, the Reasoning Layer L_2, the second layer of the change-stakeholder network, is introduced to help capture and model the implicit correlation and causal effects of changes in objects. The aim of this layer is not to provide a general-purpose model of correlation and causation relations between changes and real-world actions, which is challenging due to a multitude of factors to consider and thus out of the scope of this research. Instead,

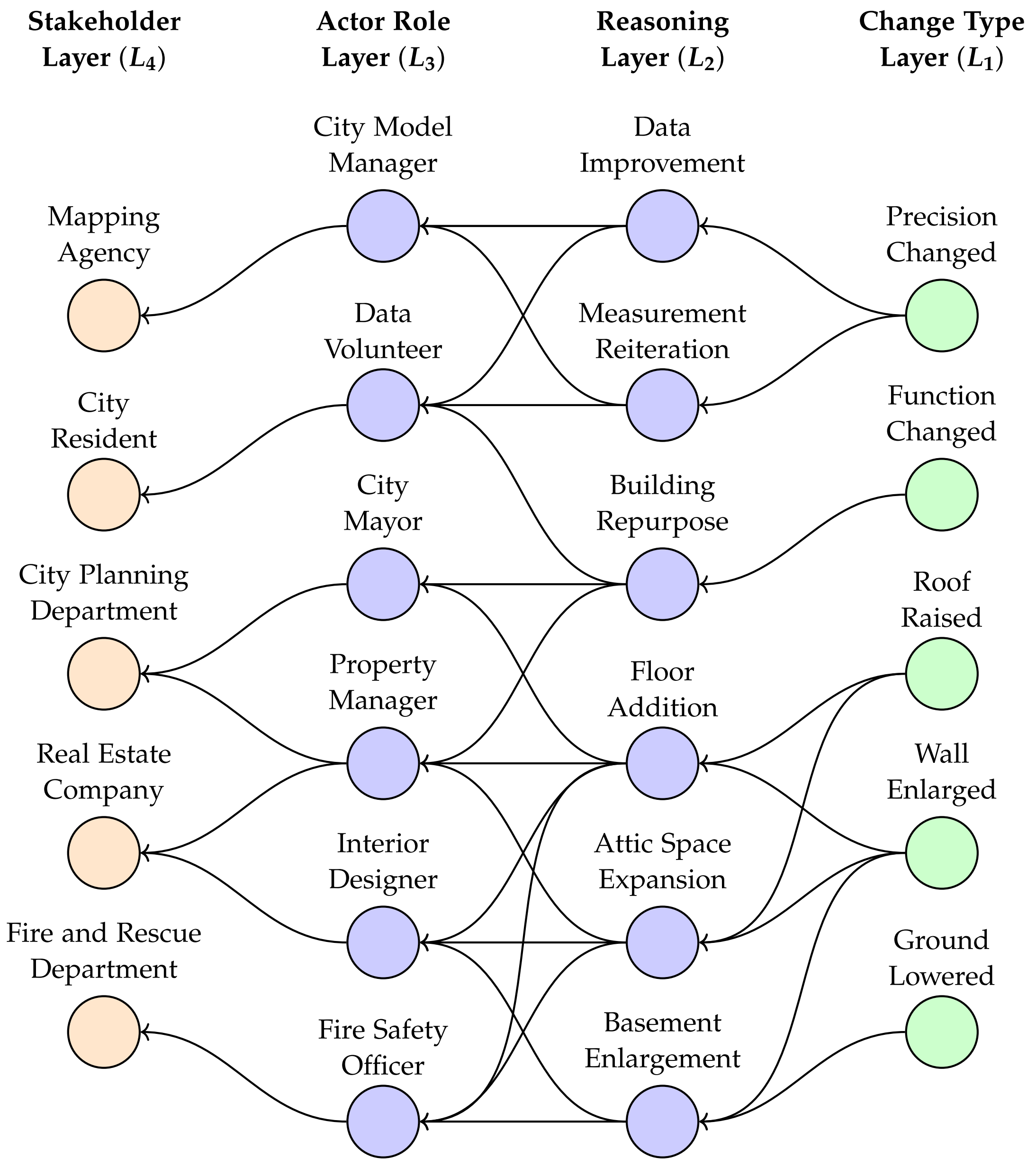


Figure 5.12.: An illustration of a change-stakeholder network employed for the modelling and analysis of changes, stakeholders, and their relations. Nodes represent semantic concepts and object classes, while relationships represent the relevance relations between nodes. The nodes and relationships in this figure are interpreted from right to left. However, in a bidirectional network, the interpretation can also proceed from left to right.

this layer provides the necessary means to explicitly capture such hidden relations in specific use cases.

In the Reasoning Layer L_2, each node represents an action that may have caused the observed changes in the data. Based on their impact within an UDT, these actions can be categorized as either changes to the physical reality or modifications to the digital representations. Similar to the types of changes in the Change Type Layer L_1, a conceptual model of these actions can be employed to fill the nodes of the Reasoning Layer L_2. Then, the correlation and causality between changes in layer L_1 and real-world actions in layer L_2 can be expressed explicitly using relationships in the semantic network.

Each relationship connecting a node in layer L_1 with a node in layer L_2 represents a causality between the two nodes. When outgoing relationships from two change nodes u and v in layer L_1 converge at an action node t in layer L_2, a correlation between the changes u and v is found, with the action t being their correlation effect. For example, as shown in Figure 5.12, the changes representing raised roofs and enlarged walls in layer L_1 are connected with the action of adding a new floor to the existing building. This implies that these changes may have been caused by the action. Similarly, since these changes have outgoing relationships that meet at the attic space expansion node, performing this action may trigger both of these changes.

On the other hand, when two actions result in the same changes, they may also be correlated. For instance, the action nodes representing the addition of a new floor and the expansion of the attic space are correlated, since they both share the same change nodes for raised roofs and enlarged walls.

Actor Role Layer L_3 and Stakeholder Layer L_4

Changes in layer L_1 are caused by actions in layer L_2, which are, in turn, initiated by humans. Previous studies have highlighted the complexity of the relationships between stakeholders and various types of changes in semantic 3D city models (Nguyen & Kolbe, 2020, 2021, 2022). Interest levels, like cities, evolve over time. Stakeholders may perceive different types of changes differently depending on their current professions, positions, and roles in the process. Therefore, to allow for meaningful analysis and interpretation of changes with respect to stakeholders, two additional layers are proposed to represent stakeholders in the change-stakeholder network: the Actor Role Layer L_3 and the output Stakeholder Layer L_4, as shown in Figure 5.12.

By decoupling roles from stakeholders as a separate layer, the network can establish direct connections between actions in layer L_2 and their actors in layer L_3. Roles provide a more expressive and robust means of describing the functional positions of a stakeholder in the process, especially when a stakeholder can be associated with many different roles at the same time. The Stakeholder Layer L_4, the last and output layer of the network, represents stakeholders based on their physical characteristics, such as individual beings (citizens), companies (private sectors), and organizations (non-governmental and governmental organizations). Similar to previous layers, a conceptual model for actors in layer L_3 and stakeholders in layer L_4 for specific use cases can be employed to populate the network (Nguyen & Kolbe, 2021).

5.5.3. Graph-based Path-tracing Analysis

This section introduces efficient traversal techniques for analysing the complex interrelations between changes and stakeholders based on a given change-stakeholder network.

Path-tracing Techniques

A significant advantage of the change-stakeholder network is its compatibility for efficient *path-tracing techniques*. These graph-based techniques allow for a comprehensive analysis of the interrelationships between changes and stakeholders simply based on the available modelled nodes and their connections.

The term 'path tracing' is used in this study to denote the traversal techniques employed between the layers of the change-stakeholder network. This name is inspired by the path-tracing technique used in the field of computer graphics, which simulates realistic global illumination of a 3D scene. It begins at the individual pixels on the objects' surface and follows along the many light rays bounced between objects until the light source is reached (Kajiya, 1986). Any light paths that did not reach the light source are discarded.

Using similar terminology, the one-way path-tracing process of the change-stakeholder network starts with the input layer L_1 and follows numerous graph paths between layers until the output layer L_n is reached. Paths that did not reach the target layer are discarded. The two-way variant of the path-tracing process can additionally start with the output layer L_n and end at the input layer L_1.

Forward and Backward Path Tracing

In this study, path tracing in the direction from the Change Type Layer L_1 to Stakeholder Layer L_4 is referred to as *forward path tracing*, while the reverse is referred to as *backward path tracing*. These are illustrated in Figures 5.11 and 5.13.

When applied to the entire change-stakeholder network, both the path-tracing directions enable efficient analyses of the often hidden interrelations between changes and stakeholders, such as:

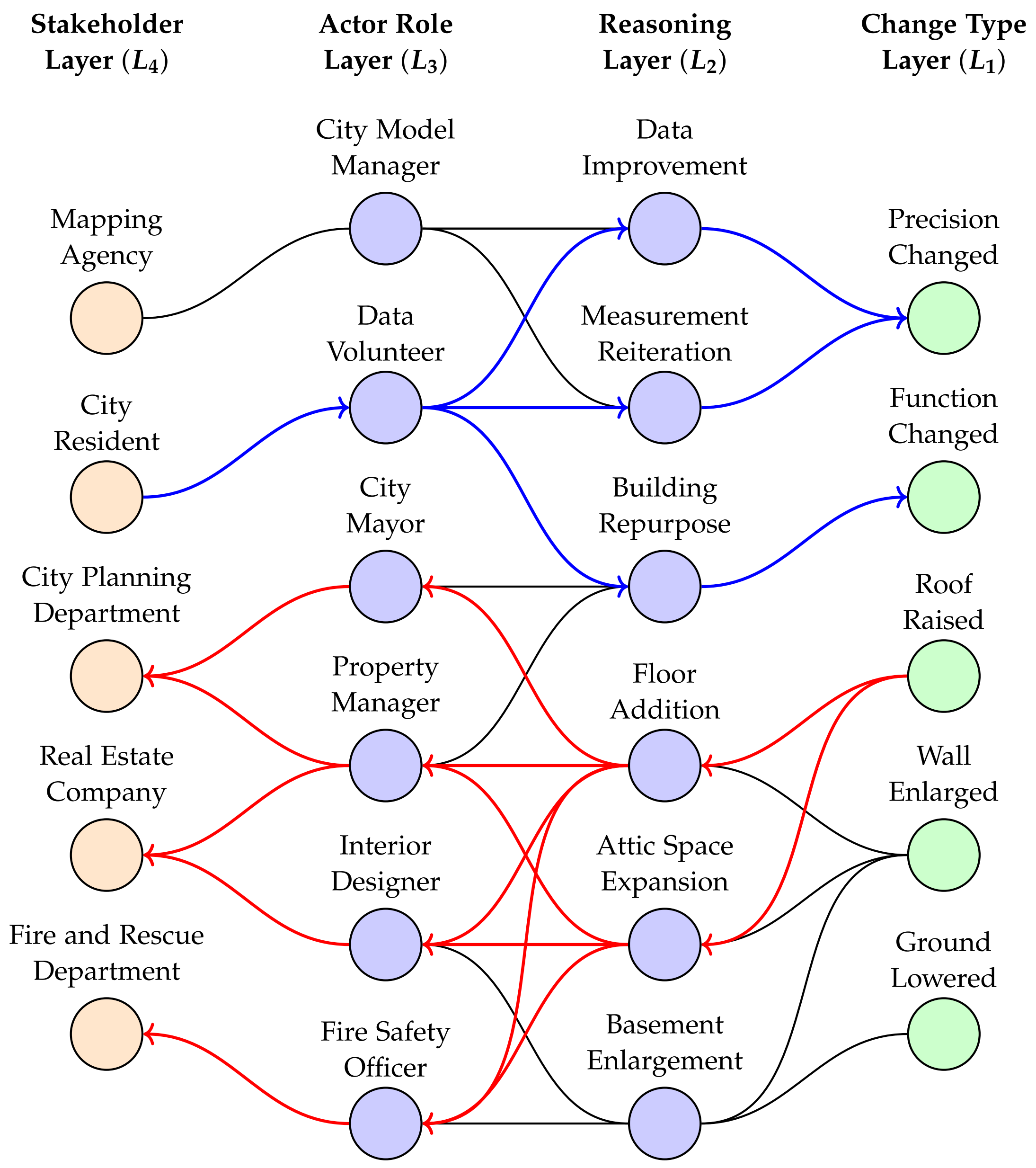


Figure 5.13.: An example of a two-way path-tracing analysis over changes, stakeholders, and their relevance within the change-stakeholder network given in Figure 5.12. The paths traced during the forward and backward path-tracing process are shown in red and blue, respectively.

1. **Change-Stakeholder Analysis:** This process determines how relevant a change is to different actions, actors, and ultimately stakeholders. Forward path tracing is employed in this case. For instance, the change *RoofRaised*, as introduced in Figure 5.1 and discussed in the example rule network in Figure 5.5, could suggest that a new floor has been added or the attic space has been expanded, thereby increasing the living space of the building. This information may be of interest to the city mayor, as well as property managers, interior designers, and fire safety officers. These actor roles are represented by the city planning department, real estate sector, and fire and rescue department, respectively. Thus, this analysis is employed when a number of changes have been detected and need to be evaluated with respect to stakeholders.

2. **Stakeholder-Change Analysis:** This process identifies which roles, actions, and ultimately types of changes are relevant to a specific stakeholder. Backward path tracing is employed in this case. For example, city residents, particularly volunteered data contributors, may be interested in processes such as data improvement, measurement reiteration, and building repurposing. As a result, they may be interested in changes concerning the quality of the data stored in the city model, as well as changes in the functions or types of a building. Thus, this analysis is essential in providing stakeholders with a more concise yet accurate list of potential changes of interest, apart from many other less relevant changes stored in the database.

As a result, despite the potentially large number of nodes and relationships stored in the change-stakeholder network, the relations between a given change and a stakeholder can be quickly identified and evaluated from both directions.

For the path-tracing process to terminate successfully, the network must either be free of cycles for each tracing direction, or if it does contain cycles, they should only be traversable for a finite number of times. To allow for both forward and backward path tracing, the given change-stakeholder network must be either undirected or bidirectionally traversable, as in the case of Neo4j graphs.

More examples and guidelines on the construction and path-tracing analysis of such networks are available in Section 7.7 and Listing C.1.

5.5.4. Evaluating Traced Paths

When the path-tracing techniques presented above result in multiple paths, such as when there are multiple changes reachable from a given stakeholder, it becomes necessary to establish a metric that enables comparison among these paths. This is done by evaluating the accumulated weights of all traced paths within the network in a specific direction. For forward path tracing, the accumulated weight w_P of a traced path $P = \left(v^{(1)}_{i_1}, v^{(2)}_{i_2}, \ldots, v^{(n)}_{i_n}\right)$ can be defined as follows:

$$w_P = \sum_{k=1}^{n-1} x^{(k)}_{i_k} w^{(k,k+1)}_{i_k,i_{k+1}} \tag{5.1}$$

where $v^{(k)}_{i_k}$ is the i_k-th node of layer L_k with weight $x^{(k)}_{i_k}$, $1 \leq i_k \leq |L_k|$ with $|L_k|$ as the number of employed nodes of layer L_k, and $w^{(k,k+1)}_{i_k,i_{k+1}}$ is the weight of the relationship connecting $v^{(k)}_{i_k}$ with $v^{(k+1)}_{i_{k+1}}$. Notably, the node weight $x^{(n)}_{i_n}$ of the target layer L_n is excluded from w_P. This can be included if needed.

If node weights are not utilized, or if all node weights are set to 1, Equation (5.1) simplifies to:

$$w_P = \sum_{k=1}^{n-1} w^{(k,k+1)}_{i_k,i_{k+1}} \tag{5.2}$$

Equations (5.1) and (5.2) allow for zero node and relationship weights, but the sums are not normalized. Multiplying all weights would normalize the value, but a single zero component would override others. Thus, the summation approach is used.

The path P has a maximum length of $n-1$ relationships, since the network has n layers. There exist at most $\prod_{k=1}^{n} |L_k|$ such paths. If the length of P is not less than $n-1$, it indicates that this path has not been fully traced, meaning that the target layer is not reachable. Otherwise, among the fully traced paths, the most fitting candidate for further analyses can be determined by calculating the maximum value of the accumulated weights, or their minimum value depending on the use cases and the sign of the weight values. This corresponds to solving the shortest or longest path problem in graph theory (Dijkstra, 1959).

However, it is often necessary to consider multiple candidate paths, since a stakeholder may have interest in not one, but a multitude of changes. As a result, instead of computing their minimum or maximum weight values, paths with weights below or above a certain threshold are considered. They can then be further sorted based on their weights for assessment. The value of this threshold is determined depending on the specific use cases.

For example, in the change-stakeholder and stakeholder-change analysis mentioned above, weights of nodes and relationships can be given as real, positive, normalized numbers to denote the relevance values between changes and stakeholders. Therefore, the most fitting candidates are the fully traced paths with the highest accumulated weights above a defined threshold.

Given that no relationships between nodes within the same layer exist in the change-stakeholder network (or such relationships can be ignored during the path-tracing process, as explained above), the adjacency properties of two consecutive layers L_k and L_{k+1} can be described using a modified submatrix A_k of their adjacency matrix.

This submatrix is defined as follows:

$$A_k = \left(x_i^{(k)} w_{i,j}^{(k,k+1)} \right) \in \mathbb{R}^{|L_k| \times |L_{k+1}|} \tag{5.3}$$

Thus, the adjacency matrix A of the entire network becomes:

$$A = \begin{array}{c} \\ \boldsymbol{L_1} \\ \boldsymbol{L_2} \\ \vdots \\ \boldsymbol{L_{n-1}} \\ \boldsymbol{L_n} \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{L_1} & \boldsymbol{L_2} & \boldsymbol{L_3} & \boldsymbol{...} & \boldsymbol{L_n} \end{array} \\ \begin{pmatrix} 0 & \boldsymbol{A_1} & 0 & \dots & 0 \\ 0 & 0 & \boldsymbol{A_2} & \dots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & \dots & \boldsymbol{A_{n-1}} \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix} \end{array} \tag{5.4}$$

An interesting property emerges when an adjacency matrix (filled with values 0 and 1) is multiplied by itself multiple times. Specifically, each matrix value a_{ij} of A^k, the matrix product of k copies of A, is equal to the number of paths of length k that can be traced between the layers L_i and L_j. Thus, if the distance between layers L_i and L_j is greater than k, the value a_{ij} of A^k becomes zero, as shown in Equation (5.4) with $k = 1$.

Given that layers are serially connected, only the 'diagonal' $(A_1 A_2 \dots A_{n-k})$ of A^k is populated with non-zero values. As k increases, this diagonal shifts upwards, leaving zeros behind. Thus, if n represents the number of layers in the change-stakeholder network, the number of all fully traced paths of length $n-1$ between layers L_1 and L_n can be found in the submatrix located in the top right corner, confined by the first $|L_1|$ rows and last $|L_n|$ columns of A^{n-1}. This can be utilized to reduce the size and computational complexity of A^k.

The aforementioned observations apply only when the values of the adjacency matrix are either 1 or 0. Leveraging this property, the matrix multiplication of the adjacency matrix can be modified to accommodate values other than 1 and 0, such as numeric values that represent the accumulated weights of respective paths.

In this modified matrix, its value $a_{ij} \in A \circ_f A \circ_f \dots \circ_f A = A^k$ denotes the accumulated weight of a path of length k between nodes v_i and v_j. Here, $\circ_f$ is a modified matrix multiplication such that, for each matrix $P = \left(p_{ij}\right) \in \mathbb{R}^{m \times l}$ and $Q = \left(q_{ij}\right) \in \mathbb{R}^{l \times s}$, their modified multiplication is $P \circ_f Q = \left(r_{ij}\right) \in \mathbb{R}^{m \times s}$, with $r_{ij} = \max\left(f\left(p_{it}, q_{tj}\right)\right) \forall t \in [1, l]$. Depending on the use cases, a minimum function may be preferred. The function f is defined for real values x and y such that $f(x, y) = 0$ if $xy = 0$, or $x + y$ otherwise. This ensures that the weights of each path can be accumulated correctly through successive multiplications of the modified adjacency matrix A with itself.

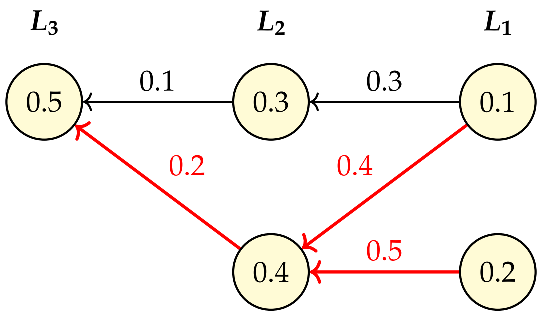


Figure 5.14.: An example of a directed network with three layers and five nodes. Nodes and relationships are depicted with their respective normalized weights.

To demonstrate, the simplified network illustrated in Figure 5.14 has the following adjacency matrix according to Equation (5.4):

$$A = 10^{-2} \cdot \left(\begin{array}{cc|cc|c} 0 & 0 & 3 & \mathbf{4} & 0 \\ 0 & 0 & 0 & \mathbf{10} & 0 \\ \hline 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & \mathbf{8} \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{5.5}$$

The adjacency matrix A contains the weights of all paths of length one between the layers of the given network. The weight of each path is calculated based on the node and relationship weights, as defined in Equation (5.1). For example, the value of a_{14} of A is a product of the node weight 0.1 and relationship weight 0.4.

The modified matrix product of A with itself is described as follows:

$$A^2 = A \circ_f A = 10^{-2} \cdot \left(\begin{array}{cc|cc|c} 0 & 0 & 0 & 0 & \mathbf{12} \\ 0 & 0 & 0 & 0 & \mathbf{18} \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{5.6}$$

Therefore, the matrix product A^2 shows the maximum accumulated weights of all traced paths of length two from layer L_1 to layer L_3. For example, the value a_{15} of A^2 is computed as $10^{-2} \cdot \max(3+3, 4+8) = 10^{-2} \cdot 12$, while a_{25} of A^2 has a value of $10^{-2} \cdot \max(0, 10+8) = 10^{-2} \cdot 18$. These values correspond to the accumulated weights of the red paths shown in Figure 5.14.

Thus, the accumulated weights of paths traced within the change-stakeholder network can be efficiently represented and evaluated using the network's adjacency matrix.

5.6. Summary and Discussion

This chapter first provides a classification of changes detected between the graph representations of CityGML documents. These changes are categorized into various classes within a hierarchy, including changes in appearance, semantic, geometric, topological, structural, LOD, syntactic, and scope information. To identify complex patterns among these changes, this chapter introduces a rule network, which is a type graph that can describe the typical behaviours and interrelationships of changes within patterns. It allows for the capture of all potential complex patterns among changes in a single network without redundancy, especially for patterns that depend on the outcomes of others. The rules within this network are aggregative, combining smaller changes with lower level of semantic content into changes with higher level of semantic content.

While the structure of the rule network allows for the definition of various change patterns, its nodes and relationships are assigned with properties that are essential for the subsequent pattern matching process, which is designed to detect patterns among changes attached to the content network. These properties can contain a range of information, from the type of the represented change and the type of the next content node, to the directive to calculate the scope of changes and boolean conditions for the creation of the next interpreted changes.

To detect change patterns based on the predefined rules, this study utilizes a First In First Out (FIFO) queue to manage all changes, including interpreted changes that may be part of bigger ones. The queue allows for successive handling of all changes without processing any change more than once. Each change removed from the queue undergoes a series of checks to ensure the creation of the next interpreted changes when a sufficient number of changes of correct types and origins within the graphs are present.

This aggregation of component changes is enabled by the use of aggregative memory nodes. These auxiliary nodes contain temporary information required for the creation of the next interpreted changes and are attached to the content nodes where next interpreted changes will be created. A memory node contains essential information about the type of the next change to be created, knowledge gained during the interpretation process, the number of change instances per type acquired thus far, and the number of occurrences of these changes required to create the next change.

This research further introduces the change-stakeholder network, a multilayered semantic network to represent, model, and evaluate the relevance relations between different types of changes and different groups of stakeholders. A standard change-stakeholder network consists of four layers: Change Type Layer L_1, Reasoning Layer L_2, Actor Role Layer L_3, and Stakeholder Layer L_4. While nodes represent concepts and objects involved in the process, relationships represent the relevance relations between nodes and are only set if a semantic relationship exists between the nodes. Nodes and relationships can be individually assigned with weights when necessary.

Lastly, this chapter presents path-tracing techniques as efficient traversal strategies for determining the relevance relations between every change and stakeholder. These techniques include the forward and backward path tracing to identify the stakeholders interested in a given change, and conversely, the changes a given stakeholder may be interested in. Their relevance relation is confirmed if there exists at least one path between them across layers. Mathematical methods, including a modified adjacency matrix, are provided to evaluate traced paths based on their accumulated weights.

Some notable observations and insights related to the concepts introduced in this chapter include:

1. **Significant Reduction of Number of Changes to Report**: As shown in Figure 5.5, all individual changes such as *RoofMoved*, *WallResized*, and *HeightChanged* can be represented by a single interpreted change *RoofRaised*. This effect is amplified in global change patterns, such as where all individual translations of geometric elements can be represented by a single interpretation, suggesting a systematic elevation change of the entire city model. The method employs aggregative rules to condense a large number of changes into a few interpretation nodes that are more comprehensible to stakeholders.

2. **Database-wide Application of Pattern Rules**: Pattern rules within a rule network are verified against all changes of corresponding types, regardless of the rules' size. For instance, a pattern rule with two rule nodes interpreting a translated polygon as a lifted polygon (for positive z component) is applied to all polygon translations found in the graphs. This implies that even a small rule network can have a significant impact on the interpretation of changes.

3. **Rule Networks for Consistency Check**: In addition to detecting change patterns, rule networks can also potentially be used to detect inconsistencies among changes in the data. For instance, pattern matching rules can examine whether a change in the number of storeys above ground of a building is consistent with the change in the building's measured heights and the vertical size of all wall surfaces. However, such verification process is out of the scope of this thesis.

4. **Expansion of the Change Network**: While interpreting, the change network is constantly expanded with new interpreted changes. These changes are connected to form a hierarchy, as dictated by the structure of both the employed rule and content network. New changes are attached to their corresponding content nodes, while the entire content network remains unchanged.

5. **Memory Nodes after Interpretation**: During pattern matching, memory nodes store temporary information needed for the creation of the next interpreted changes. They also serve as a temporary reference, linking all component changes of the next change. After the next change has been created and linked to its component changes, the memory node is left only connected to its content node. Depending on the implementation, memory nodes can then be removed. However, as the graph database Neo4j may reuse internal identifiers of deleted nodes for newly created ones, the implementation of this study removes the memory nodes in the graphs only after all processes are complete.

6. **Extended Change-Stakeholder Networks**: The change-stakeholder network introduced in this chapter contains four layers. However, the same principles can be applied for any number of layers larger than two, provided they care connected via relationships that reflect their semantic interrelations.

6. Optimization Strategies for Massive CityGML Datasets

While the methods proposed in previous chapters can effectively accomplish their designated objectives, their runtime complexity grows with the size of the input CityGML documents. Thus, this chapter introduces optimization strategies to enhance the efficiency of these methods when applied to large CityGML documents.

6.1. Chunk-wise Mapping

The methods detailed in Algorithms 1 to 3 are capable of generating a graph representation of an entire CityGML dataset. The input CityGML documents, along with their object-oriented representations, are entirely held in main memory. Consequently, an increase in the size of the input datasets leads to an increase in memory consumption. This, in turn, results in a slowdown of the program due to the time-consuming nature of the memory allocation and data structure construction processes. While this approach is feasible for small CityGML datasets, it becomes increasingly challenging or even impossible for larger ones, particularly those that reach gigabytes or even terabytes of data. Therefore, additional optimization strategies are necessary to process massive input CityGML datasets effectively, especially with restricted memory capacities. This section addresses Research Question RQD1 (Memory Reduction).

6.1.1. Splitting Massive CityGML Documents

There exist two primary methods for reading XML documents: the Document Object Model (DOM)-based approach and the event-based approach.

The DOM (Wood et al., 2000) is a standardized interface for reading, handling, and manipulating HyperText Markup Language (HTML) and XML documents. It represents each document as a hierarchical tree structure, where each element in the document is depicted as a node. Specifically, the DOM for HTML treats every element as a node, including all HTML elements, attributes, the text contents of HTML elements, comments, and even the document itself, which serves as the root node of the DOM tree. Each path in the tree begins at the root node and ends at a leaf node. Every node must have exactly one parent (except for the root node) and can contain an arbitrary number of children (with the exception of the leaf nodes). The entire DOM tree is loaded into the main memory, allowing for both direct and fast access, navigation, and manipulation of data from any point within the tree. As a result, the DOM and its variants are employed in most modern web browsers.

Contrary to DOM that operates on entire documents, event-based methods like the Simple API for XML (SAX) (Harold et al., 2000) operate specifically on individual segments of XML documents. SAX sequentially reads XML documents in a single pass, triggering events during parsing. When an event is detected, SAX reports it and then discards most of that information to accommodate the next event. As a result, the memory footprint of SAX is often much smaller than that of the DOM, especially for datasets that are larger than the capacity of the available main memory. Moreover, due to its event-driven nature, SAX generally outperforms DOM-based parsers in tasks that can be completed in a single start-to-end pass of XML documents, such as indexing, formatting, or converting XML elements. However, SAX is read-only and does not have the capability to change or write XML contents.

Many common event-based APIs like SAX employ a *push* model, in which the parser feeds the content of XML documents to the application immediately after an event has been detected. This approach, however, deprives the application of control over how and when XML elements are parsed, as the parser continues to iterate until it reaches the end of the document. This restrictive *observer design pattern* can be challenging for developers to adapt (Harold, 2003). In contrast, the *pull* model used in the Streaming API for XML (StAX) poses fewer adaptation challenges as it aligns with the more familiar *iterator design pattern*. In this model, the application dictates how and when the parser searches for the next piece of information from the document. StAX can thus be perceived as a bridge between the two opposing interfaces of DOM and SAX. It allows for reading, manipulating, and writing XML elements, while still retaining the ability to handle documents of arbitrary size with low memory consumption.

The concepts of DOM, SAX, and StAX are included in the Java API for XML Processing (JAXP). The library *citygml4j* utilizes JAXP, in conjunction with JAXB as introduced in Section 3.1.2, to generate in-memory Java objects of CityGML elements. To reduce memory usage while parsing large CityGML documents, *citygml4j* allows for dividing these documents into smaller pieces, often referred to as chunks. Each chunk can contain a CityGML feature, ranging from smaller elements such as boundary surfaces up to top-level features like entire buildings.

For example, a city model composed of six buildings can be segmented into seven chunks. Each of the first six chunks contains one building, while the seventh chunk includes the main content and semantic structure of the city model without the contents of its buildings. As these chunks are detached from their original form, *citygml4j* incorporates six *href* properties into the main seventh chunk to reference each of the six associated extracted chunks. These new XLink references, although not present in the original CityGML document, are necessary for the reconnection of all seven chunks after they have been mapped onto graphs. The processes for mapping and reconnecting CityGML chunks are explained in Sections 3.3 and 3.4 respectively.

6.1.2. Reconnecting Graph Representations of CityGML Chunks

Figure 6.1 provides an overview of the three-step process for generating cohesive, connected graph representations of large CityGML datasets: (1) segmenting a large CityGML document into chunks, (2) mapping these chunks onto graphs, and (3) resolving XLinks between these subgraphs. The same concepts illustrated in this example can be applied to accommodate any large-scale CityGML datasets.

The first two steps can be executed while chunks are being parsed from the input documents. However, the third step, resolving XLinks, can only be done after both the previous steps are complete for the entire dataset. This is because objects referenced by XLinks may not have been read yet while parsing the sequential chunks from the input dataset and thus are not yet available for reconnecting.

6.2. Leveraging Thematic Indexes

As mentioned previously, the resolution of XLinks requires the identification of graph nodes that contain either an *href* or *id* property. A brute-force approach to this would be to scan through all nodes and search among their property keys for the presence of *href* or *id*. However, this method can be time-consuming and inefficient, especially when dealing with large CityGML datasets that are represented by a vast number of nodes.

Furthermore, this issue is not limited solely to the resolution of XLinks. There exist other processes employed in this study that also require frequent access to nodes and relationships with specific labellings and properties, such as finding potential matches for a given subgraph.

In such cases, indexing can be applied. In databases, an index is an efficient data structure for storing and managing copies of selected data. It allows for faster retrieval of indexed data at the cost of additional write operations and storage space.

In the graph database Neo4j, thematic indexes can be applied to both nodes and relationships that have specific labels or types and properties. Neo4j provides support for several types of thematic indexes, including token lookup index, range index, text index, and full-text index. This section addresses the first half of Research Question RQD2 (Database Indexes).

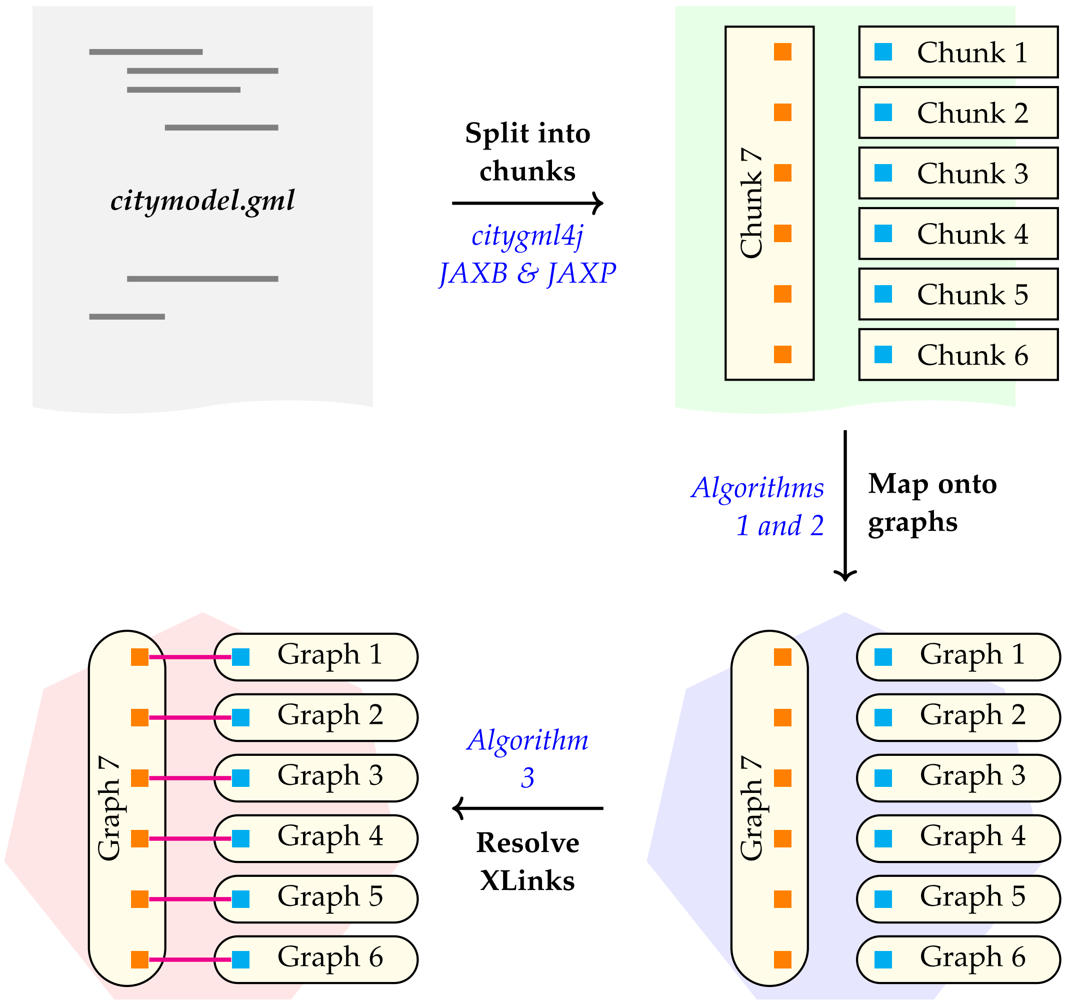


Figure 6.1.: Visualizing the processes: From segmenting a large CityGML document into chunks, to mapping these chunks onto graphs, and resolving XLinks between these subgraphs. The *href* properties are shown in orange, while the referenced identifiers are shown in blue. The tools and methods employed in these processes are *citygml4j* with JAXB and JAXP, as well as Algorithms 1 and 2 for mapping, and Algorithm 3 for resolving XLinks. Each of the blue chunks can contain a top-level feature (such as a building), while the orange chunk represents the city model. In this example, the city model has six buildings, but the same concepts can be applied to accommodate any large-scale CityGML datasets.

6.2.1. Token Lookup Index

Token lookup indexes are the only indexes in Neo4j that operate exclusively on node labels and relationship types, independent of their properties. They are enabled by default for all nodes and relationships (Neo4j, 2023). The token lookup indexes play a central role in Neo4j's indexing system, as they accelerate the population of other indexes and are extensively utilized in many database operations and queries. Listing 6.1 provides some examples of such database queries in Cypher, Neo4j's query language. They showcase the implicit use of the token lookup indexes in Cypher queries.

```
// The token lookup indexes are implicitly utilized in Cypher

// Using node label predicate
MATCH (n:Building)
RETURN n
// Alternative node label predicate
MATCH (n)
WHERE n:Building
RETURN n

// Using relationship type predicate
MATCH ()-[r:boundedBy]->()
RETURN r
// Alternative relationship type predicate
MATCH ()-[r]->()
WHERE r:boundedBy
RETURN r
```

Listing 6.1: Querying by node labels and relationship types in Cypher.

6.2.2. Range Index

Range indexes are the most commonly used indexes for node and relationship properties in Neo4j. When used in conjunction with the token lookup index, a range index enables the search for nodes and relationships based on a wide range of operations on their properties. These operations, which are based on boolean logic, include exact comparisons, checks for membership and existence, as well as inequality and prefix searches. Range indexes can be applied to multiple properties of the same node, forming a **composite index**.

Listing 6.2 illustrates the use of range indexes on node properties in Cypher queries. The same applies to relationship properties.

```
// Range indexes on node properties

// Exact comparison
MATCH (n:Building)
WHERE n.id = "bdlg1"
RETURN n

// List membership test
MATCH (n:Building)
WHERE n.id IN ["bldg1", "bldg2", "bldg3"]
RETURN n

// Existence test
MATCH (n:Building)
WHERE n.id IS NOT NULL
RETURN n

// Inequality search
MATCH (n:Building)
WHERE n.storeysAboveGround > 5
RETURN n

// Prefix search
MATCH (n: Building)
WHERE n.id STARTS WITH "bldg"
RETURN n
```

Listing 6.2: Using range indexes on node properties in Cypher.

6.2.3. Text Index

Range indexes can be employed for both numeric and character-based properties, as shown previously. In contrast, a text index is applicable exclusively to strings. When used in conjunction with the token lookup index, it allows for efficient retrieval of nodes and relationships based on a variety of string operations on their properties. These operations cover exact string comparisons, checks for membership in a string list, as well as prefix, suffix, and substring searches.

Text indexes are particularly efficient when dealing with suffix and substring searches, providing them an advantage over range indexes (Neo4j, 2023). Similarly to the use of *STARTS WITH* for prefix searches, the key words *ENDS WITH* and *CONTAINS* are employed for suffix and substring searches, respectively.

6.2.4. Full-text Index

The previously mentioned range and text indexes offer limited operations on string properties, such as exact matching, as well as prefix, suffix, and substring searches. In contrast, a full-text index provides more flexibility, as it divides string values into smaller tokens, thus allowing for matching from any point within the strings. This is achieved by tokenizing the indexed string values (Neo4j, 2023).

Full-text indexes are provided by Apache Lucene[1], a powerful library developed for indexing and searching. Full-text indexes are equipped with various analysers that dictate the tokenization process of strings. Analysers can be provided for different natural languages, such as English and German, but also in various structures, such as an email analyser for parsing email communication.

Unlike range and text indexes, a full-text index can be applied to multiple node labels and relationship types. It can also be applied to multiple string properties at once. In contrast to composite indexes, where only nodes with the exact labels and all specified properties are indexed, a full-text index is performed on nodes that have any of the specified node labels and relationship types, as well as any of the specified properties.

Listing 6.3 illustrates how full-text indexes can be employed in Cypher.

```
// Query nodes using a full-text index on street names
CALL db.index.fulltext.queryNodes(
  "indexStreetNames", // unique index name
  'Arcisstr. OR Boltzmannstr.' // logical operation
) YIELD node
RETURN node

// Query relationships using a full-text index on wasXLink attributes
CALL db.index.fulltext.queryRelationships(
  "indexWasXLink",
  "true"
) YIELD relationship
RETURN relationship
```

Listing 6.3: Using full-text indexes on node and relationship properties in Cypher.

[1]https://lucene.apache.org/

6.2.5. Combining Thematic Indexes

This thesis leverages different types of indexes available in Neo4j for different types of nodes and their properties. The goal is to enhance the retrieval speed of these elements within a large-scale graph database instance. While indexes can also be applied to relationships, the majority of queries used in this research make extensive use of indexes on nodes and their properties. These indexes play a crucial role not only in the mapping process but also in the subsequent processes of matching graphs and interpreting their changes. Table 6.1 showcases some of the most prominent examples of thematic indexes used in these processes.

Table 6.1.: The use of thematic database indexes to enhance query time of specific node labels and their properties.

Node Labels and Properties	**Index Type**	**In Process**		
All labelled nodes	Token lookup index	①	②	③
Nodes containing *id*	Text index	①	②	③
Nodes containing *href*	Text index	①		
Code nodes (for *function*, *usage*, *roofType*)	Text index			③
Length nodes (for *measuredHeight*)	Range index			③
Building nodes, on *storeysAboveGround*	Range index			③
Building nodes, on *storeysBelowGround*	Range index			③
Top-level features, on *relativeToTerrain*	Text index			③
Top-level features, on *relativeToWater*	Text index			③
Top-level features, on *creationDate*	Full-text index			③
Top-level features, on *terminationDate*	Full-text index			③
Top-level features, on *yearOfConstruction*	Full-text index			③
Top-level features, on *yearOfDemolition*	Full-text index			③

① Graph mapping ② Change detection ③ Change interpretation

Both the token lookup index for all labelled nodes and the text index on nodes with the property *id* are utilized in all processes. In contrast, the text index on nodes with the property *href* is primarily used during the mapping process to resolve XLinks. Once the XLinks are resolved, the property *href* becomes redundant, and thus, its index is no longer needed in the subsequent processes.

The majority of the indexes are reserved for the change interpretation process, as it attempts to detect patterns among various types of content and change nodes, particularly those associated with top-level features such as buildings.

The full-text index on top-level features with a date property, such as *creationDate* and *terminationDate*, can be performed using a date analyser. On the other hand, nodes with numeric values like *measuredHeight* can be found using a range index.

The creation and removal of some of these indexes are depicted in Listing 6.4.

```
// Create a range index
CREATE RANGE INDEX indexLengths
FOR (n:Length)
ON (n.value)

// Create a text index
CREATE TEXT INDEX indexIds
FOR (n:Building)
ON (n.id)

// Create a full-text index
CREATE FULLTEXT INDEX indexDates
FOR (n:Building)
ON EACH [
  n.creationDate,
  n.terminationDate,
  n.yearOfConstruction,
  n.yearOfDemolition
]

// Drop an existing index
DROP INDEX indexDates
```

Listing 6.4: Creating and dropping indexes in Cypher.

In Neo4j, indexes can be established either before or after the creation of nodes and relationships. Once initiated, indexes are maintained in the background and are automatically updated when any changes are made to the associated nodes and relationships in the database. When processing queries, Neo4j automatically leverages the available indexes to optimize runtime, thereby eliminating the need for explicit usage or invoking of indexes in each Cypher query.

However, like node labels, the number of thematic indexes employed in the database should be kept small to avoid excessive amount of additional disk storage and decreased runtime efficiency while managing these indexes.

6.3. Spatial Indexing

As depicted in Table 6.1, matching graph representations of city objects leverages thematic indexes on node labels and their identifiers. This is particularly useful when matching objects with unchanged identifiers over time. However, as explained in Section 2.2.2, this is not always guaranteed, leading to multiple different identifiers of the same objects. As a result, further optimization strategies are required for matching objects in one-to-many and many-to-many relationships of large city models.

In addition to the semantic information, CityGML documents are also rich in geometric contents, which can be utilized to efficiently retrieve and differentiate objects from one another. Database indexes that leverage this source of information are referred to as spatial indexes. This section addresses the last half of Research Question RQD2 (Database Indexes).

6.3.1. Point Index

Neo4j natively supports a basic spatial index known as the point index. This index is capable of handling spatial queries involving points, such as determining the distance between two given points or verifying whether a point is located within a certain bounding box. However, its capabilities are limited, as it can only operate on a single property of a node or relationship, and that property must be of the type *point* specified by Neo4j. Moreover, exact comparison is the only non-spatial query supported by this index. Listing 6.5 illustrates how this point index can be created and utilized in Cypher.

```
// Create a point index
CREATE POINT INDEX indexPoints
ON (n:Point3D)
FOR (n.location) // e.g., n.location = point({x: 100, y: 100, z: 100})

// Calculate distance between two point nodes n and m
point.distance(n, m)

// Verify whether a point node n is in a bounding box
// bounded by two point nodes lowerLeft and upperRight
point.withinBBox(n, lowerLeft, upperRight)

// Drop the point index
DROP INDEX indexPoints
```

Listing 6.5: Applying a point index in Cypher.

Neo4j's point index supports both 2D and 3D points. However, it only recognizes point locations given in either the Cartesian coordinate system (with coordinates x, y, and z) or the World Geodetic System (WGS) 84 CRS (with latitude, longitude, and height). For spatial querying, Neo4j provides several built-in spatial functions, such as *point.distance(n, m)* and *point.withinBBox(n, lowerLeft, upperRight)*, as shown in Listing 6.5. The resulting values and units of these functions vary depending on the dimensionality and the CRS of the given points (Neo4j, 2023). For example, the distance (both 2D and 3D) between two points provided in the Cartesian coordinate system is calculated using Pythagoras' theorem. On the other hand, the 2D distance of points provided in the WGS 84 CRS is calculated using the haversine formula, while the 3D distance is calculated using a combination of the haversine formula and Pythagoras' theorem.

6.3.2. R-trees for Indexing More Complex Geometries

The point index can be utilized during the change detection and interpretation process to efficiently locate city objects based on their spatial properties. The storage and management of indexed points are handled automatically by Neo4j in the background. However, this index only applies to points, while many queries in the change detection and interpretation process involve more complex geometries in higher dimensions, such as 2D surfaces in 3D space. Moreover, the locations of these points must be provided using a built-in constructor specified by Neo4j. In addition, the point index only supports coordinates given in the Cartesian or WGS 84 system.

In this context, a Bounding Volume Hierarchy (BVH) is often used for indexing a large number of complex geometries. A BVH is a tree data structure constructed using a bottom-up approach, where each leaf node represents an individual input geometry. Geometries in close proximity are grouped together, forming their minimum bounding volume. This volume is then stored as a parent node encompassing the wrapped geometries. These minimum bounding volumes, in turn, are bundled together based on their proximity to each other, creating a larger minimum bounding volume at the next level. This process of spatial aggregation repeats 'upwards' until all geometries are encapsulated within a single minimum bounding volume, which serves as the root node of the tree. BVHs are particularly useful in applications such as in ray tracing and overlapping or collision detection.

The term BVH is a broad classification over various types of spatial hierarchies. Each hierarchy is distinguished by the shape of the minimum bounding volumes used, which can range from 2D geometries such as rectangles and circles to 3D shapes such as cubes and spheres. However, most studies and applications tend to limit these shapes to simpler forms that best fit their input geometries, while being able to enhance query efficiency. In this context, **R-trees** (Guttman, 1984) are a special class of BVHs that employs 2D Axis-aligned Minimum Bounding Boxes (AABBs) as minimum bounding volumes (Haverkort, 2004). The use of AABBs simplifies the execution of many spatial operations such as intersection queries, as only comparisons and subtractions of the coordinates at the vertices of the bounding boxes are required. Although the 'R' in the name 'R-tree' indicates the use of rectangles, R-trees can also be applied to three-dimensional bounding boxes. Moreover, each node in an R-tree can be allocated a maximum number of regions, known as the tree's block size. This implies that the leaf nodes of an R-tree are no longer atomic entities, as they contain input data rectangles which are the actual atomic elements. Figure 6.2 illustrates the structure of an R-tree with a block size of three used for organizing 2D surfaces.

R-trees can be utilized for a variety of spatial queries, such as determining the nearest geometry to a given location or finding all geometries within a certain radius of a specific point. However, they are exceptionally efficient in coverage and overlap queries. Since every non-leaf node in the tree represents a bounding box that contains its member elements, if a given geometry does not intersect with this bounding box, it can be concluded that it does not overlap with any of the enclosed geometries. This essentially allows for the majority of the tree hierarchy to be disregarded during queries. Similar to B-trees (Bayer & McCreight, 1970), R-trees are balanced search trees, thus resulting in the *logarithmic runtime complexity* for coverage or overlap queries on average.

Several variants of R-trees exist. A standard R-tree may contain large overlap or empty areas between their minimum bounding boxes during construction. The **R$^+$-tree** (Sellis et al., 1987) addresses these issues by allowing geometric objects to be stored redundantly in multiple leaf nodes. As a result, however, an R$^+$-tree may become larger than its corresponding R-tree for the same input dataset. On the other hand, the **R*-tree** (Beckmann et al., 1990) avoids excessive overlap between minimum bounding boxes in the original R-trees by splitting elements from overfilled nodes and reinserting them into the tree. While this increases the construction cost of R*-trees compared to R-trees, it allows for better performance for coverage and overlap queries in general.

6.3.3. Employing R-trees in a Graph Database

As of this writing, the most recent release of Neo4j does not officially support spatial indexing for geometries more complex than points in higher dimensions. However, several community-supported plug-ins and extensions are available, such as the Neo4j Spatial utility library[2]. This tool enables indexing of already existing spatial data in the graph database with a new layer of R-tree nodes, allowing for efficient spatial operations and queries on 2D regions and surfaces. The R-tree hierarchy is maintained in the same persistent storage as its associated spatial data. This library has been successfully

[2]https://neo4j-contrib.github.io/spatial

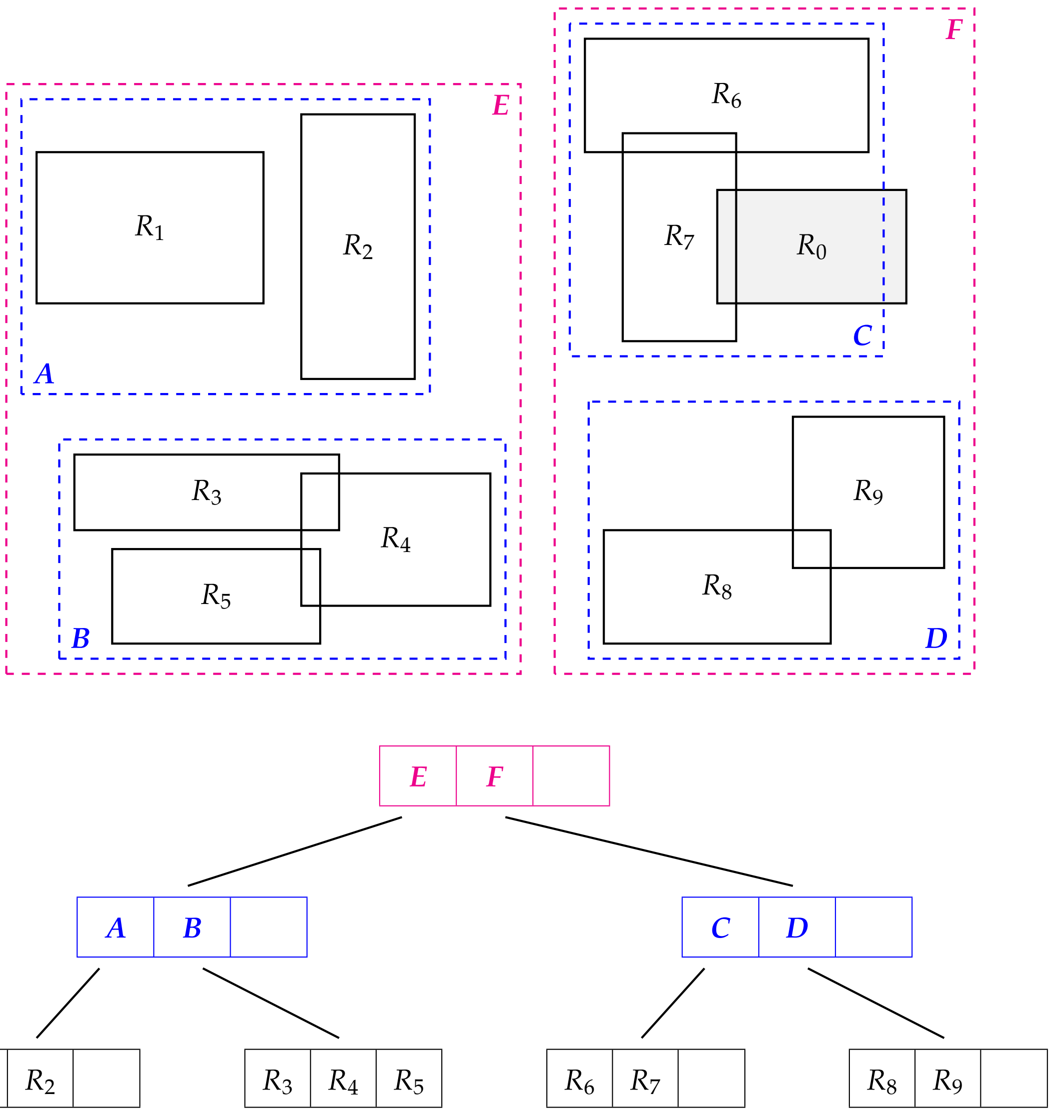


Figure 6.2.: An example of an R-tree with a block size of three that contains nine rectangles labelled from R_1 to R_9. The minimum bounding boxes of these rectangles are A, B, C, and D, as shown in blue. These boxes are further enclosed by larger minimum bounding boxes, E and F, depicted in red, which together form the root node of the R-tree. The grey rectangle R_0 does not intersect with E, and therefore does not intersect with any of the enclosed rectangles R_1 to R_5. Similarly, R_0 does not intersect with D, and thus does not intersect with R_8 and R_9. However, R_0 does intersect with C, indicating a potential intersection with R_6 or R_7.

employed in several past studies (Nguyen et al., 2017; Nguyen, 2017; Nguyen et al., 2018), mostly with earlier versions of Neo4j. However, as a community-driven project, it does not receive updates as frequently as Neo4j, leading to compatibility issues with the latest version of Neo4j employed in this thesis.

Alternatively, in-memory representations of R-trees can be used for spatial indexing. These representations allow for faster access to R-tree nodes, as well as faster writes and updates in the tree compared to its persistent representation. By storing only a single reference, such as a unique identifier or a single relationship, to each associated spatial element of specific types of CityGML objects, like top-level features, the memory required by the in-memory R-tree is kept at a comparatively low level. This thesis makes use of such in-memory representations of R-trees, including their variants such as R*-trees, and leverages their open-source and up-to-date implementations[3]. During execution, the R-tree hierarchy is maintained in the main memory. Prior to program termination, this hierarchy is stored as persistent nodes using the same methods used to map CityGML objects onto graphs as outlined in Algorithms 1 and 2, since these methods can be applied to any in-memory objects. In subsequent executions, these R-tree nodes can be used to reconstruct the entire R-tree hierarchy in main memory using the reverse mapping methods outlined in Algorithms 4 and 5.

6.3.4. Linking R-tree Nodes with Geometric Content

Nevertheless, both the persistent and in-memory representation of R-trees require a method to link each element of a leaf node with a content node that provides the geometric information. Is this research, two-dimensional footprints of top-level features serve as the source of geometric content for this coupling. Each link is established through a relationship connecting a geometry node and its corresponding node in the R-tree. These one-to-one relationships between the R-tree nodes and footprints of top-level features, as well as between the footprints and the top-level features themselves, allow for efficient and unique identification of top-level features that intersect with a given test region. Such queries are frequently employed during the change detection and interpretation process. This coupling between the R-tree nodes and geometric nodes, as illustrated in Figure 6.3, is performed during the mapping process.

Most CityGML datasets already contain bounding shapes for the entire city model and top-level feature, such as buildings. These bounding boxes, being AABBs, can be directly linked to an R-tree, which also employ AABBs. However, if such bounding shapes are not available, they can be quickly calculated for each top-level feature by extracting its boundary surfaces and comparing the coordinates of their vertices. Bounding shapes of implicit geometries are computed based on their anchor points.

[3]A commonly used library for R-trees in Java can be found at https://github.com/davidmoten/rtree.

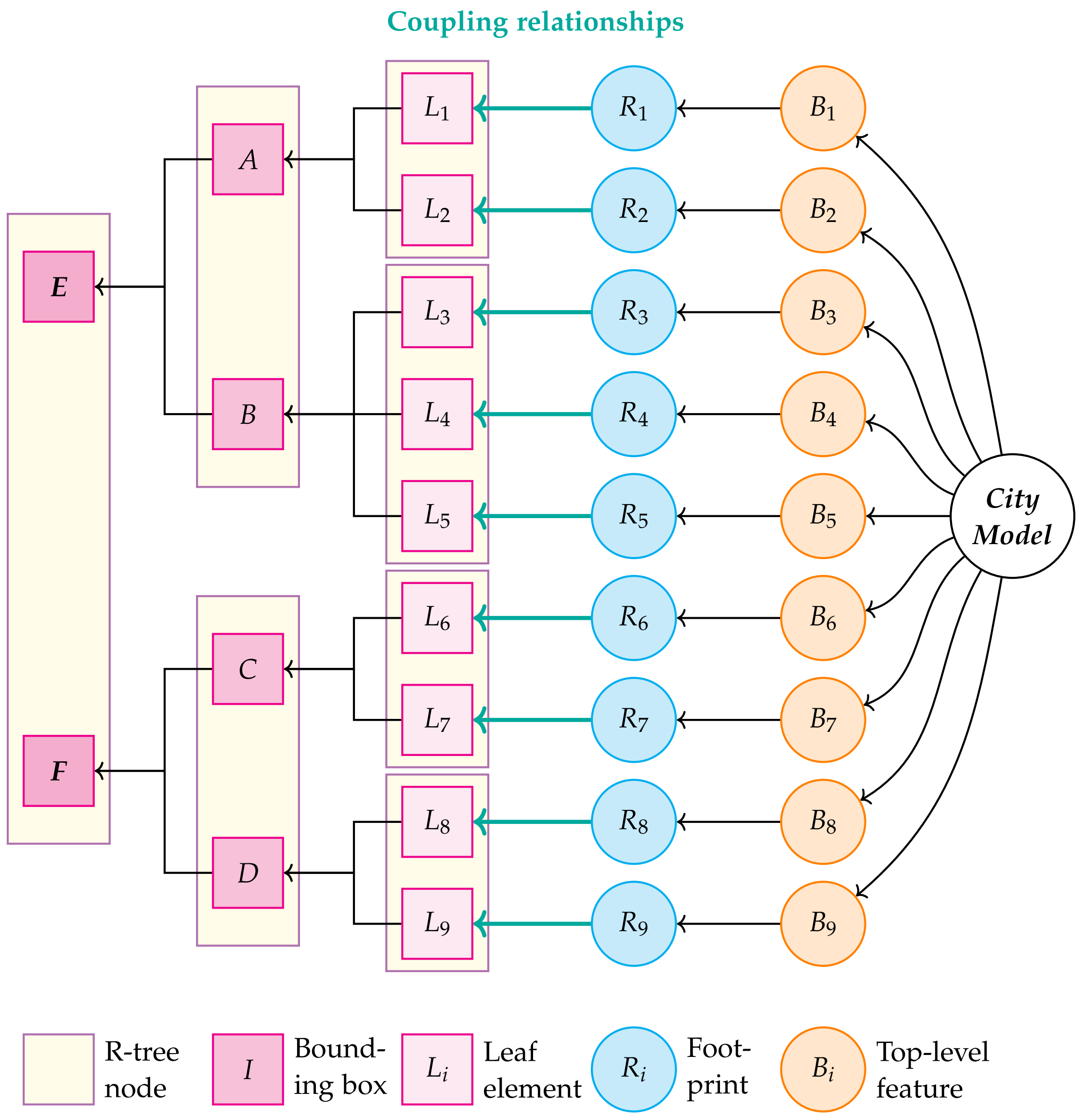


Figure 6.3.: An illustration of the coupling mechanism between the footprints (blue) of all top-level features (orange) of a given city model (white) and the leaf elements (marked as red L_i nodes) of an R-tree. These links (displayed as green connections) provide the geometric data (on the right-hand side) necessary for constructing the R-tree (on the left-hand side). The results of queries performed on this R-tree can thus be traced back to these geometries, as well as their corresponding top-level features. The root node (E and F) of the R-tree covers the spatial extent of the entire city model.

6.4. Transaction Management

In the graph database Neo4j, operations on the schema, indexes, or the graph entities are performed within **transactions**. Neo4j's support for full ACID properties ensures:

1. **Atomicity**: Failure on any part of a transaction leads to failure of the entire transaction, causing all changes made by the transaction to be reverted back to a prior state. This process is known as *rollback*. Changes can only be *committed* to the database if the transaction executes without errors.

2. **Consistency**: The graph database is always in a consistent state between transactions. Only transactions, whose effects comply with the existing rules and constraints of the database, are accepted.

3. **Isolation**: Each graph entity can be modified by at most one transaction at a time. A modifying transaction obtains a write lock on its elements, preventing other operations from accessing them at the same time.

4. **Durability**: Once a transaction is successfully performed, its committed changes are stored persistently in the graph database. This, combined with the automatic transaction logging in Neo4j, allows for tracing and recovering database states between committed transactions.

This section addresses Research Question RQD3 (Transaction Management) and Research Question RQD4 (Concurrency Control).

6.4.1. Memory Management and Batch Transactions

To facilitate full ACID properties in Neo4j, all uncommitted data and intermediate states of each transaction are first stored in main memory. They are only written to the database once the transaction has been successfully executed. As a result, more complex transactions may require more memory. Hence, large transactions that involve a significant portion of a large graph database should generally be avoided. Instead, smaller transactions are preferred. However, given that each transaction must be committed, an excessive number of small transactions can lead to inefficiency due to the high cost of write operations to the persistent data storage. Thus, finding a balance between the size of transactions and the frequency of their commits is crucial.

In this study, when possible, methods and transactions are applied to top-level features in CityGML. For example, the mapping of each building onto its graph representation or the comparison of two buildings can be performed within a transaction. This approach significantly reduces the number of commits needed in the database, while simultaneously maintaining a small memory footprint for each transaction.

Alternatively, a predetermined number n of small and repetitive operations can be grouped into a single transaction. This means that a commit is only performed for every n successful operations. Such transactions are known as **batch transactions**. For instance, the mapping process outlined in Algorithm 1 can be applied to $n = 10$ CityGML top-level features, such as buildings, before their graph representations are written to the database. This further reduces the write frequency to the database at the expense of increased memory usage. Listing 6.6 shows an example of how such batch transactions can be implemented in Java. However, batch transactions can only be applied in cases where the member operations do not share their intermediate results with each other, as newer versions of Neo4j prohibit any sharing of intermediate graph data across transactions.

```java
// Initialize citygml4j reader
CityGMLInputFactory in = initCityGMLInputFactory();
in.setProperty(CityGMLInputFactory.FEATURE_READ_MODE,
                      FeatureReadMode.SPLIT_PER_COLLECTION_MEMBER);
CityGMLReader reader = in.createCityGMLReader(citygmlFile));

// Read CityGML top-level features in batches
final int batchSize = 10;
List<CityGML> batch = new ArrayList<>();
while (reader.hasNext()) {
  CityGML chunk = reader.nextChunk().unmarshal();
  batch.add(chunk);
  if (batch.size() == batchSize || !reader.hasNext()) {
    // Perform batch transaction in auto-closable block
    try (Transaction tx = graphDb.beginTx()) {
      for (CityGML feature : batch) {
        // Map each top-level feature chunk onto graphs
        map(feature);
      }
      tx.commit();
    }
    // Reset batch
    batch = new ArrayList<>();
  }
}
```

Listing 6.6: Implementing a batch transaction for Neo4j in Java 7 or newer.

6.4.2. Concurrency Control

Neo4j's full support for ACID properties enables the execution of transactions in a multi-threaded manner, as they are isolated and independent from each other. Parallelization can enhance the performance of various processes employed in this thesis significantly. However, transactions in Neo4j are single-threaded and cannot be distributed across parallel threads. Consequently, the execution of each transaction must be completely encapsulated within a single thread. As a result, transactions on top-level features or batch transactions, as explained in Section 6.4.1, are ideally suited for parallelization.

While transactions are contained within individual threads, they can still access and influence the same elements stored in the graph database. Allowing parallel methods to modify the same node or relationship at the same time could lead to incorrect results or unforeseen effects. Thus, concurrency control is crucial to prevent such scenarios.

Locking Mechanism

In Neo4j, **locks** serve as a mechanism to ensure the isolation of transactions and consistency of data. A lock is automatically created and assigned to a graph entity when it is being accessed by a transaction for the first time. Therefore, locks are associated with the resources but can be granted to the transactions that access them. Neo4j supports two main types of locks based on their isolation levels: **read locks** (read-committed isolation level) and **write locks** (serializable isolation level) (Neo4j, 2023). These are described as follows:

1. **Read locks**: A read lock is activated when a transaction attempts to access a graph entity without changing it. Read locks are less restrictive than write locks, as read-only operations do not prevent others from reading or modifying the associated resources. This means that multiple read locks can be granted to parallel transactions that operate on the same graph node or relationship, offering a significant speed advantage over write locks. As a result, read locks are sufficient and employed in the majority of cases.

2. **Write locks**: A write lock is enabled when a transaction attempts to modify a graph entity. In contrast to read locks, only one write lock can be assigned to each graph node or relationship at a time, allowing only one write transaction to execute and blocking other write transactions. Once the transaction holding the lock has ended, the lock is released and can be granted to other pending transactions. This explicit locking mechanism allows for the serial execution of write transactions for the same graph elements. Thus, write locks are more restrictive and slower than read locks.

In Neo4j, the types of locks are determined by both the nature of the database transactions and the structure of their associated resources, as summarized in Table 6.2. Database transactions include operations such as retrieval, creation, deletion, and updating of nodes, relationships, and their properties. Structurally, a node can be categorized as 'sparse' or 'dense' (Neo4j, 2023). Neo4j considers a node as *dense* if it is linked with more than a certain number of relationships (for example, 50 relationships) at any point during runtime. In contrast, a *sparse* node never exceeds that number of relationships throughout the entire execution. This distinction between dense and sparse nodes is crucial for enhancing efficiency. While sparse nodes can be exclusively locked by a single transaction without a significant impact on the overall performance, applying the same to a dense node may prevent a large number of concurrent transactions from accessing it, causing a decline in the performance. Thus, write locks on dense nodes are allowed to be shared internally among concurrent transactions, forming a **prevention lock** that prevents the dense nodes from being deleted during the lock's duration.

Table 6.2.: An overview of database operations, resources, and their associated locks in Neo4j. Extended and adapted from (Neo4j, 2023).

Operation	**On Resources**	**Lock Types**
Read	Node	**Read lock** on the node
	Relationship	**Read lock** on the relationship
Update	Node label or property	**Write lock** on the node
	Relationship property	**Write lock** on the relationship
Create	Node	**No lock**
	Relationship (sparse nodes[1])	**Write lock** on the nodes
	Relationship (dense nodes[2])	**Prevention lock** on nodes' deletion
Delete	Node	**Write lock** on the node
	Relationship (sparse nodes[1])	**Write lock** on the relationship **Write lock** on the nodes
	Relationship (dense nodes[2])	**Write lock** on the relationship **Prevention lock** on nodes' deletion

[1] This applies to both start and end node of the relationship, if they are sparse.
[2] This applies to both start and end node of the relationship, if they are dense.

Deadlock Handling

The use of write locks and synchronization is crucial to prevent lost updates caused by concurrent transactions that attempt to modify the same node or relationship. However, this may lead to cases where two or more transactions mutually obstruct each other's access to the resources they require. Such transactions are in a state of deadlock.

A deadlock occurs in a system with single-instance resources if all of the following conditions, known as the *Coffman conditions* (Coffman et al., 1971), are met:

1. **'Mutual Exclusion' Condition**: Transactions are granted exclusive control over the resources they require, thereby blocking others from accessing them.

2. **'Wait For' Condition**: Transactions hold resources allocated to them while simultaneously requesting for additional resources held by other transactions.

3. **'No Pre-emption' Condition**: Resources can only be released once their associated transaction is complete.

4. **'Circular Wait' Condition**: Two or more transactions form a dependency cycle, where each transaction is holding resources requested by the next transaction in the cycle.

Major strategies on handling deadlocks can be divided into two main categories: reactive and proactive approach. The **reactive approach** allows deadlocks to occur but aims to mitigate their effects. The *detection technique* is a prime example of this approach. In systems like Neo4j that employ the detection techniques, resources and transactions are periodically examined for deadlocks. If a deadlock is detected, its locking transactions are aborted, thereby rolling back all changes made by these transactions and releasing the resources until the deadlock is resolved. This technique is used in systems where deadlocks cannot be completely ruled out, such as in the presence of (directed) cycles in the underlying graph structure of the graph database. This approach allows for graceful handling of deadlocks during runtime at the cost of increased computational demands and reduced efficiency.

On the other hand, the **proactive approach** aims to prevent deadlocks from occurring in the first place. The *deadlock prevention* and *deadlock avoidance* technique are two prominent examples of this approach. The deadlock prevention technique eliminates deadlocks by ensuring that at least one of the Coffman conditions cannot occur. On the other hand, the deadlock avoidance accomplishes this by thoroughly analysing all resources required by each transaction and determining whether a deadlock could potentially occur once these resources are distributed. However, this is difficult to achieve, as it requires full insight into the entire resource consumption profile of each transaction.

Therefore, this study employs the deadlock prevention technique to prevent deadlocks from occurring. However, due to the fully supported ACID properties in Neo4j, the 'mutual exclusion' and 'no pre-emption' condition, which are essential for the presence of a deadlock, are always in effect for write transactions. As a result, the deadlock prevention technique often relies on invalidating the 'wait for' or 'circular wait' condition. In most cases in this study, this is achieved as follows:

1. **Read-only Transactions**: Transactions that require either no lock or only read locks on the resources, such as creating a new node or retrieving an existing node as shown in Table 6.2, can be executed in multi-threaded mode without causing deadlocks. In these cases, no further actions are required.

2. **Isolated Write Transactions**: Transactions that require a write or prevention lock can still be performed concurrently without causing any deadlock, if:
   a) Their resources are isolated and do not share any nodes, and
   b) Their results are not used as input for other concurrent transactions.

3. **Single-threaded Transactions**: If neither of the above conditions applies, transactions are executed in single-threaded mode to ensure no deadlock can occur.

Table 6.3 gives an overview of the methods employed in this study and their corresponding concurrency properties.

Table 6.3.: An overview of the methods introduced in this study and their corresponding concurrency properties. Deadlock prevention techniques are applied.

Method	Lock	In-/Output	Mode
map(source)	R W	II IO	MT
resolveXLinks(graph)	R W P	II IO	MT
toObject(node)	R	II IO	MT
compare(left, right)	R W	II IO	MT
findBest(left, right, ref)	R W	II IO	MT
matchPatterns(changes, contents, rules)[1]	R W P	II IO	MT
matchPatterns(changes, contents, rules)[2]	R W P	SI RO	ST

R Read lock W Write lock P Prevention lock ST Single-threaded MT Multi-threaded
II Isolated input SI Shared input IO Isolated output RO Reused output
[1] When applied to sub-elements up to top-level features
[2] When applied to top-level features up to city model

The use of single-threading and multi-threading across the processes presented in this thesis is summarized as follows:

1. **The method *map(source)***: Large input CityGML documents are divided into smaller chunks, each of which is mapped independently onto graphs within its own separate database transaction. This implementation enables the processing of the input and output of the method *map(source)* in isolation, allowing for full multi-threading support.

2. **The method *resolveXLinks(graph)***: All graph nodes containing an XLink or an identifier are retrieved from the database. The list of XLink nodes is then divided into smaller distinct parts of equal length, with each part processed in a separate batch transaction. When multiple XLinks reference the same identifier, temporary locks are introduced to handle concurrent writes from multiple XLink nodes, as well as the deletion of the identifier node. Therefore, this implementation provides multi-threading support for the method *resolveXLinks(graph)*.

3. **The method *toObject(node)***: Unlike other methods, the reconstruction of CityGML objects requires only read access to the graph contents. As a result, both the input and output of the method *toObject(node)* are processed in isolation, allowing for full multi-threading support.

4. **The methods *compare(left, right)* and *findBest(left, right, ref)***: Both methods *compare(left, right)* and *findBest(left, right, ref)* primarily require only read access to the graph contents for matching. Write locks are introduced only when auxiliary nodes, such as indicators for already matched geometries, are created and attached to the content nodes in the database. A separate transaction is initiated for each top-level feature from the old city model, in which the matching process takes place. Thus, multi-threading support is implemented.

5. **The method *matchPatterns(changes, contents, rules)***: When applied to top-level features or their sub-elements, the method *matchPatterns(changes, contents, rules)* operates in multi-threaded mode, with each transaction processing the changes and contents within a top-level feature. Write locks are employed to handle concurrent attachment of interpreted changes to their corresponding content nodes, while prevention locks are introduced to prevent the deletion of content and memory nodes while interpreting rules. However, when applied to elements higher in the semantic hierarchy, such as the city model, the method *matchPatterns(changes, contents, rules)* transitions to single-threaded mode. This is due to the significantly smaller number of content and memory nodes at this stage, indicated by their high level of semantic concentration.

7. Application Results

This chapter showcases the orchestrated application of all methods proposed in this research. The process begins with mapping CityGML documents onto graphs, followed by comparing these graph representations. It then proceeds to create a rule network for pattern definitions and match these among the detected changes. The ultimate objective is to provide a comprehensible and meaningful insights into the changes between the input original CityGML documents, tailored to specific stakeholders such as those introduced in Section 2.4.

7.1. Implementation: An Overview

The open-source implementation of this thesis is available online as a repository[1] on GitHub, a popular platform that enables developers to build, publish, and share their work. At the time of this writing, the project has over 6,000 lines of code spanning over more than 50 classes and helper files. In addition, it leverages more than 15 third-party libraries, the majority of which are open-source or free for non-commercial use. Given the extensive nature of the project, this section only provides a concise overview of the implementation.

All methods proposed in this study are implemented using Java SE 17 (LTS), leveraging the Neo4j Java Core API version 5.11.0. Cypher, Neo4j's declarative graph query language, is employed to construct rule networks, which contain rules for identifying and matching patterns among changes.

To enhance usability and ease deployment, the entire program is available as a **Docker container**. Docker is both a tool and a platform for developing, packaging, and executing applications within lightweight, isolated environments known as containers (Docker, Inc., 2024). The use of Docker containers offers a major advantage, as they contain all information necessary for the deployment and execution of an application, ensuring consistent performance across all host systems that are equipped with a Docker engine.

While all compiled source codes are packaged within a container, persistent resources consumed or produced by the application can be stored outside of the container, within

[1]https://github.com/tum-gis/citymodel-compare

the host system. These resources include input CityGML datasets, output Neo4j database instance, R-tree footprints, execution logs, as well as program configurations and other settings. Figure 7.1 provides an overview of the directory structure of such information. To enable the exchange of these data, a bridge is established between the container and its host system using *bind mounts* or *volumes*.

7.1.1. Implementation of the Mapping Process

The mapping process employs the library *citygml4j*[2], an open-source Java API for CityGML, for reading and parsing CityGML documents, transforming them into in-memory object-oriented representations. These in-memory objects are then mapped onto graphs using the methods previously introduced in Chapter 3.

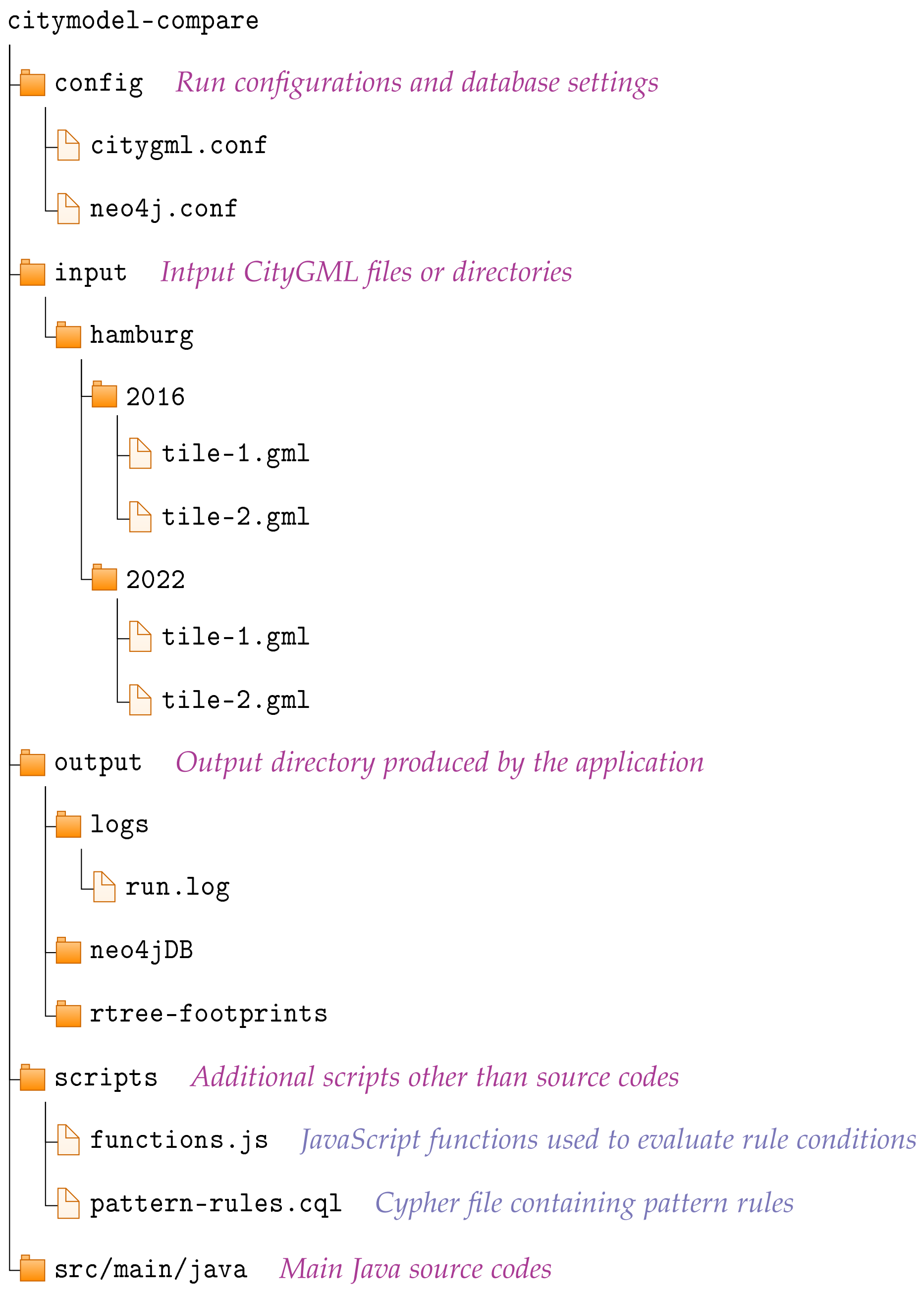
The tool *citygml4j* supports both CityGML versions 2.0 and 3.0, as well as CityJSON, a JSON-based encoding for 3D city objects and a subset of the CityGML data model (Ledoux et al., 2019). Moreover, the mapping process proposed in this thesis is capable of mapping any in-memory object onto its corresponding graphs. As a result, documents encoded in CityGML versions 2.0 or 3.0, as well as in CityJSON, can all be mapped onto graphs using the same mapping techniques provided in this thesis.

As explained in Chapter 6, to deal with CityGML datasets that contain a vast number of buildings, the *citygml4j* library enables the division of input CityGML documents into smaller, more manageable chunks. Each chunk can be configured to contain either a single CityGML feature, such as a boundary surface, or a single CityGML top-level feature, such as a building, bridge, and tunnel.

In this research, each chunk is processed within a database transaction. On one hand, initiating and terminating a transaction are time-consuming operations. On the other hand, a transaction that holds a large amount of data can prevent other transactions from accessing shared resources for its entire duration. To balance these aspects, this study divides large input CityGML documents by each top-level feature, then groups and processes these objects in batch transactions. This strategy effectively reduces the number of transactions that need to be started and committed, while maintaining relatively low memory consumption, thereby increasing the overall efficiency.

While small CityGML documents can be processed individually, larger datasets, such as those covering an entire region or country, may exist as a collection of multiple smaller CityGML documents, each storing city objects located within a spatial segment, referred to as a tile. To allow for the mapping of such tiled datasets, they can first be merged into a single document, such as using the 3DCityDB Importer/Exporter (Yao et al., 2018), prior to their mapping onto graphs. However, this approach may introduce numerous syntactic changes to the original datasets, such as alterations in

[2]https://github.com/citygml4j/citygml4j



```
citymodel-compare
├── config                  Run configurations and database settings
│   ├── citygml.conf
│   └── neo4j.conf
├── input                   Intput CityGML files or directories
│   └── hamburg
│       ├── 2016
│       │   ├── tile-1.gml
│       │   └── tile-2.gml
│       └── 2022
│           ├── tile-1.gml
│           └── tile-2.gml
├── output                  Output directory produced by the application
│   ├── logs
│   │   └── run.log
│   ├── neo4jDB
│   └── rtree-footprints
├── scripts                 Additional scripts other than source codes
│   ├── functions.js        JavaScript functions used to evaluate rule conditions
│   └── pattern-rules.cql   Cypher file containing pattern rules
└── src/main/java           Main Java source codes
```

Figure 7.1.: An overview of the persistent data used and produced by the implementation of this thesis. The core application is provided as a Docker container.

the representation of a geometry, as permitted by the encoding standard. Moreover, some thematic properties, such as *creationDate*, may also be overwritten during export.

To address these issues, the implementation of the mapping process has been extended to support the parsing and mapping of all CityGML documents given within a tiled dataset. Once all member CityGML documents have been mapped, their respective top-level features are all connected to one common node representing the single combined city model of all tiles.

7.1.2. Implementation of the Matching Process

For efficient differentiation of graph elements representing city objects in the old and new CityGML documents, each graph element is assigned with an additional label. This label, which is stored along with their actual type, indicates their originating CityGML document. This label can be any identifier, such as partition index values 0 and 1 indicating the old and new CityGML document, respectively.

As explained in the mapping process above, it is crucial to maintain a balance between the number of required transactions and their memory consumption. To achieve this, the matching process minimizes the number of transactions by matching CityGML documents based on their top-level features. It begins by retrieving all top-level feature nodes from both old and new graphs. For each reference node in the old graph, the process searches for the best match in the new graph. If a match is found, the comparison is performed further within their subgraphs. If no match exists, it suggests that the reference top-level feature may have been deleted. Once all top-level features of the old graph have been processed, any remaining top-level features of the new graphs are considered to have been recently inserted.

When matching geometries, especially 2D surfaces in 3D space, a number of geometric and spatial libraries written in Java are employed, such as Apache Commons Geometry[3]. This library is part of the open-source initiative Apache Commons and provides powerful tools for geometric processing. It supports Euclidean space in 1D, 2D, and 3D, as well as spherical space in 1D and 2D. The core concept revolves around the use of hyperplanes, which are geometric constructs with one dimension less than that of the space they are located in. For instance, a hyperplane in 2D space is a 1D line, while a hyperplane in a 3D space is a 2D plane. Hyperplanes divide elements in its surrounding space into three groups: those ‘above’ it, ‘below’ it, or directly on it.

These hyperplanes form the foundation for Binary Space Partitioning (BSP) trees, a powerful and flexible means to represent spatial partitioning of regions in space. BSP trees allow for a variety of geometric and topological operations, such as area and volume calculation, as well as union and intersection of complex shapes.

[3]https://commons.apache.org/proper/commons-geometry

An alternative to the library Apache Commons Geometry is the JTS Topology Suite (JTS)[4], an open-source Java library for processing vector geometries. However, the library does not provide full support for 3D geometries. Moreover, for the execution of simple geometric operations, such as computing the Axis-aligned Minimum Bounding Boxes (AABBs) or the centroids of top-level features, the basic mathematical operations and data structures already available in Java, like arrays, are sufficient.

As Neo4j provides limited spatial support for points, this study requires support for more complex geometries in higher dimensions. For earlier versions of Neo4j, the community-driven *Neo4j Spatial*[5] was often used to incorporate a spatial index, such as an R-tree, onto existing graphs and perform spatial operations on them. A major advantage was that the entire R-tree was stored as persistent nodes and relationships in the graphs, allowing for the reuse of the index in subsequent deployments. However, as of this writing, this library has not been updated to support newer versions of Neo4j, rendering it incompatible with the implementation of this research.

Therefore, for the spatial indexing during the matching process, this study employs *rtree*[6], an open-source Java library that provides efficient implementation of immutable R-trees and R*-trees. Listing 7.1 illustrates the instantiation and application of such an R-tree for performing geometric and topological operations. The construction and utilization of a spatial index using such an R-tree is described previously in Section 6.3.

```java
// Create an R-tree with min. and max. number of children per node
RTree<String, Geometry> rtree
        = RTree.minChildren(2).maxChildren(4).create();

try (Transaction tx = graphDb.beginTx()) {
  // Add a building node and its 2D footprint to the R-tree
  String nodeId = buildingNode.getElementId();
  Rectangle bbox = getBoundingBox(buildingNode);
  rtree = rtree.add(nodeId, bbox);
  // Find buildings that match a reference footprint
  List<Node> buildings = rtree.search(refFootprint)
        .map(id -> tx.getNodeByElementId(id)).toList();
  // Commit transaction
  tx.commit();
}
```

Listing 7.1: An example of constructing and applying an R-tree for spatial indexing.

[4]https://github.com/locationtech/jts
[5]https://github.com/neo4j-contrib/spatial
[6]https://github.com/davidmoten/rtree

7.1.3. Implementation of the Interpretation Process

Like the mapping and matching process, the implementation of the interpretation process revolves around the top-level features of CityGML documents. Initially, it assesses changes within each top-level feature. Upon finding a matching pattern, it aggregates these changes, then generates and attaches the interpretation nodes to the corresponding content nodes, extending up to the representation node of this top-level feature. Once all sub-elements have been processed, changes associated with the top-level changes can be evaluated for global patterns. When these global patterns match, their respective interpretation nodes are attached to the city model node.

This implies that the rules for identifying change patterns must allow for the propagation and concentration of knowledge acquired from interpreting changes within sub-elements up to the encompassing top-level feature. This way, each top-level feature can serve as an interpretation 'hub', containing and providing crucial information of all changes that have occurred in its sub-elements. Moreover, to enhance efficiency, the pattern rules should restrict scope calculation to top-level features only.

For instance, to detect whether a building has been shifted by a certain amount (without changing its size and shape), and if all buildings within the same city model have also been shifted by the same amount, the rules for identifying these patterns can be summarized as follows. In this simplified textual description, only the change types of rule nodes are shown. The arrows '$\rightarrow$' represent a rule relationship. For simplicity purposes, node and relationship properties are omitted, while some redundancies are introduced for ease of reading, such as the repeated mention of some change types. A complete and redundancy-free graph representation of such rules can be found in Section 5.3:

1. *Polygon moved by v* $\rightarrow$ *Roof, wall, or ground moved by v*
2. *All roofs of a building have this same change* $\rightarrow$ *All building roofs moved by v*
3. *All walls of a building have this same change* $\rightarrow$ *All building walls moved by v*
4. *All grounds of a building have this same change* $\rightarrow$ *All building grounds moved by v*
5. *All building roofs, walls, and grounds moved by v* $\rightarrow$ *(Top-level) Building moved by v*
6. *All buildings in the city have this same change* $\rightarrow$ *(Global) All buildings moved by v*

In this example, the interpretations for the translation of all building roof, wall, or ground surfaces (green) are aggregated from the sub-elements within a building. These interpretations are only attached to that building when the corresponding patterns matched. This propagation and concentration of all relevant interpretations within the building allows for the retrieval of all necessary information from that building and its sub-elements, without the need to traverse beyond the building node.

The rules for matching patterns among changes can be provided as Cypher queries stored in an external script file. This file is then parsed and executed by the interpretation process to generate corresponding rule nodes and relationships in the graph database Neo4j. An example Java source code to parse and execute such Cypher queries from a script file is provided in Listing 7.2.

```java
// Read Cypher script file into a string
String query = null;
try {
  query = Files.readString(Path.of("scripts/pattern-rules.cql"));
} catch (IOException e) {
  logger.error("Could not read Cypher script file {}.", e.getMessage());
}

// Execute the Cypher script in a transaction
try (Transaction tx = graphDb.beginTx();
     Result result = tx.execute(query)) {
  // Process result
  ...
  // Check whether all rule nodes have been created
  int count = tx.findNodes(NodeLabels.RULE).stream().count();
  logger.info("Created {} rule nodes from Cypher file {}", count, file);
  // Commit transaction
  tx.commit();
}
```

Listing 7.2: Generating a rule network from a Cypher script file in Java.

When also considering rule node and relationship properties, the complexity of the interpretation process increases significantly, particularly for the evaluation of join conditions across all preceding changes required for the creation of the next higher-level change. In real-world scenarios, these conditions can quickly become complex, such as in cases where not all criteria need to be satisfied, but only a select few (a combination of AND, OR, or even XOR operations). Additionally, complex spatial operations, such as the computation of the overlapping volume of two 3D bounding boxes, may be required at runtime. In the example above of interpreting a building's translation, the join conditions dictate that all building components, i.e., roof, wall, and ground boundary surfaces, must have been shifted by the same translation vector.

To address this problem, this study employs a JavaScript engine, such as the open-source Nashorn JavaScript Engine[7], for parsing and evaluating rule conditions during interpretation. With this JavaScript engine, all valid boolean expressions can be evaluated, regardless of their length or complexity. However, these conditions often contain variables, whose values must be known prior to the evaluation.

To provide the JavaScript engine with all information necessary for the evaluation of rule conditions, the following steps are performed prior to the evaluation process:

1. **Creation of JSON Objects**: JSON objects containing required information from all preceding rule nodes and relationships are created. The names of these JSON objects are the same as those of their corresponding rule relationships, and the JSON contents are extracted from the properties propagated from the preceding rule nodes, as described in Sections 5.3.3 and 5.3.4. This allows for sharing all variables among converging rules.

2. **Loading of Predefined Functions**: All predefined JavaScript functions are loaded into the engine's context. These functions, provided in a separate script file, are independent from the source codes and allow for specialized operations, such as comparison of two 3D vectors while accounting or error tolerances. These functions are highly customizable and adaptable for specific use cases. Such helper functions are typically defined and utilized on an ad-hoc basis by the same users that also define the rule network.

For example, the rule node representing a translated building requires three incoming rule relationships, corresponding to three preceding rule nodes representing all moved roof, wall, and ground surfaces. The content of these rule nodes are summarized below. For simplicity purposes, only properties relevant in this context are mentioned:

1. ***All building roofs moved***: This rule node contains a translation vector v, indicating that all the roof surfaces of this building have been moved by the same vector. The rule relationship outgoing from this rule node is named 'roofs' and has a weight value of 1.

2. ***All building walls moved***: This rule node contains a translation vector v, indicating that all the wall surfaces of this building have been moved by the same vector. The rule relationship outgoing from this rule node is named 'walls' and has a weight value of 1.

3. ***All building grounds moved***: This rule node contains a translation vector v, indicating that all the ground surfaces of this building have been moved by

[7]https://github.com/openjdk/nashorn

the same vector. The rule relationship outgoing from this rule node is named 'grounds' and has a weight value of 1.

4. ***Building moved***: This final rule node contains a join condition as follows:

   ```
   vectEq(roofs.v, walls.v, 0.001) && vectEq(walls.v, grounds.v, 0.001)
   ```

   The names 'roofs', 'walls', and 'grounds' refer to the preceding rule relationships. The function *vectEq(. . .)* determines whether two given vectors are equal, with an error tolerance of 0.001 length unit taken into account.

Generally, the vector *v* of these rule nodes may refer to different translation vectors for roof, wall, and ground surfaces. However, the pattern of moved buildings requires them to be equal. Listing 7.3 shows an example of the content of the JSON objects representing these vectors. These objects are then provided to the JavaScript engine.

```
roofs = { v: [ 1.0, 0.0, 0.0 ] }
walls = { v: [ 0.999, 0.0, 0.0 ] }
grounds = { v: [ 1.0, 0.001, 0.0 ] }
```

Listing 7.3: Example JSON objects required for the evaluation of the join conditions used for detecting translated buildings.

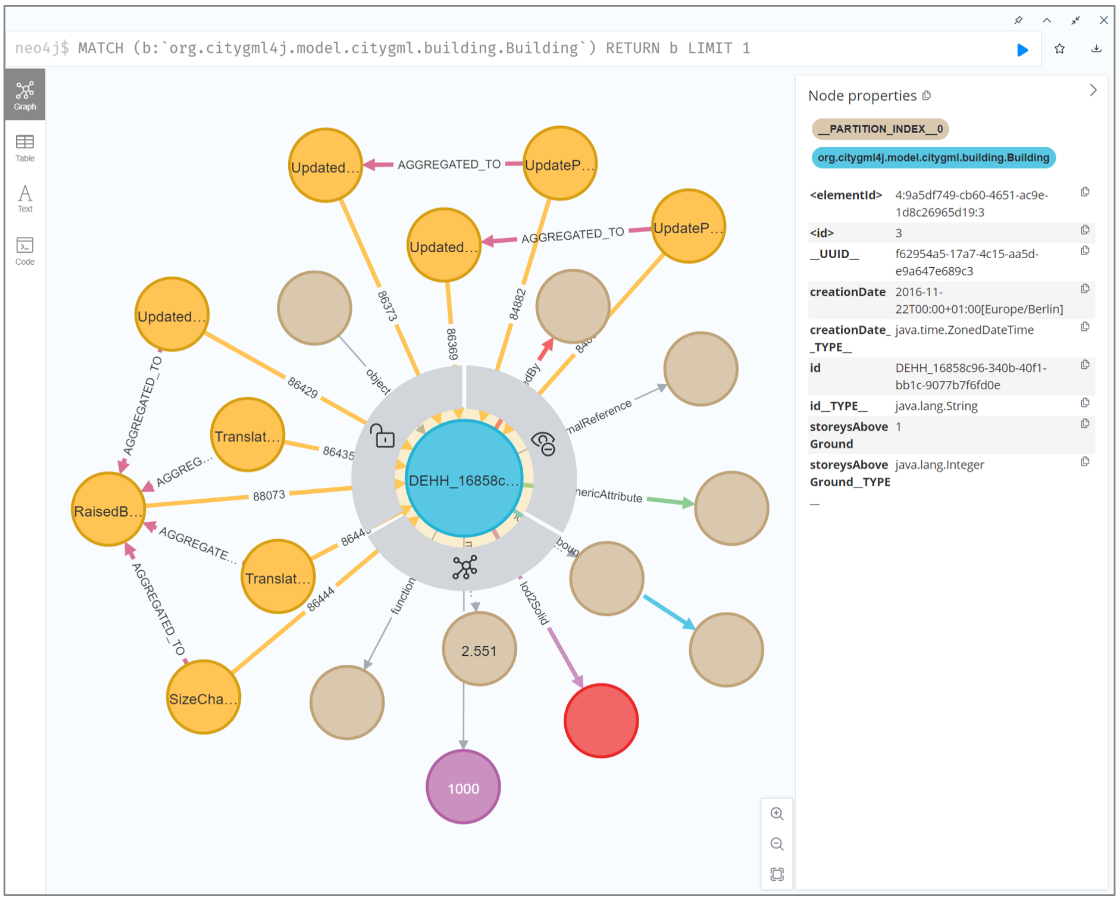
The function *vectEq(. . .)*, along with any additional helper functions required for the rule evaluation, can be provided by users in a single separate JavaScript file, such as the file *functions.js* depicted in Figure 7.1. An example of the content of this file is presented in Listing 7.4.

```
// File funtions.js: Helper functions in JavaScript

// Compare two vectors with error tolerance
function vectEq(v1, v2, errorTolerance) {
  if (v1.length !== v2.length) return false;
  for (let i = 0; i < v1.length; i++) {
    if (Math.abs(v1[i] - v2[i]) >= errorTolerance) return false;
  }
  return true;
}

// Other helper functions
...
```

Listing 7.4: Example helper functions needed for the evaluation of rule conditions.

Upon completion of all processes, the Neo4j server is activated, allowing remote clients outside of the host system of the Docker container to access its data. This is achieved by establishing a secure Bolt connection between the client and the Neo4j server (Neo4j, 2023).

This, in conjunction with the Docker container, allows for convenient deployment, as well as comprehensive visual and interactive analysis of the graph content and structure. Figure 7.2 shows the Graphical User Interface (GUI) of Neo4j Browser, an interface for executing Cypher queries and visualizing results (Neo4j, 2023). Leveraging the Bolt protocol, the Neo4j Browser acts as a remote client and can thus also be deployed outside of the Docker container.


```
neo4j$ MATCH (b:`org.citygml4j.model.citygml.building.Building`) RETURN b LIMIT 1
```

Node properties

__PARTITION_INDEX__0

org.citygml4j.model.citygml.building.Building

<elementId>	4:9a5df749-cb60-4651-ac9e-1d8c26965d19:3
<id>	3
__UUID__	f62954a5-17a7-4c15-aa5d-e9a647e689c3
creationDate	2016-11-22T00:00+01:00[Europe/Berlin]
creationDate__TYPE__	java.time.ZonedDateTime
id	DEHH_16858c96-340b-40f1-bb1c-9077b7f6fd0e
id__TYPE__	java.lang.String
storeysAboveGround	1
storeysAboveGround__TYPE__	java.lang.Integer

Figure 7.2.: The Graphical User Interface (GUI) of Neo4j Browser, an interface for executing Cypher queries and visualizing results. Using the Bolt protocol, the Neo4j Browser acts as a client that can establish a connection to a remote Neo4j server.

7.2. Test Environment and Datasets

The experiments in this chapter are conducted on a dedicated machine equipped with the following hardware and software specifications:

1. **Operating System**: Ubuntu 22.04.3 LTS
2. **Processor**: Intel® Xeon® CPU E5-2667 v3 at 3.20 GHz (16 cores, 32 threads)
3. **Main Memory**: 1 TB of Random-access Memory (RAM) in total

The methods proposed in this thesis are evaluated using the real-world CityGML datasets publicly provided by the German state of Hamburg[8]. These datasets, produced in the years 2016 and 2022 and presented in LOD2, contain approximately 760 thousand buildings in total, covering the entire area of Hamburg of circa 750 km^2. The spatial extent of the employed CityGML datasets is shown in Figure 7.3.

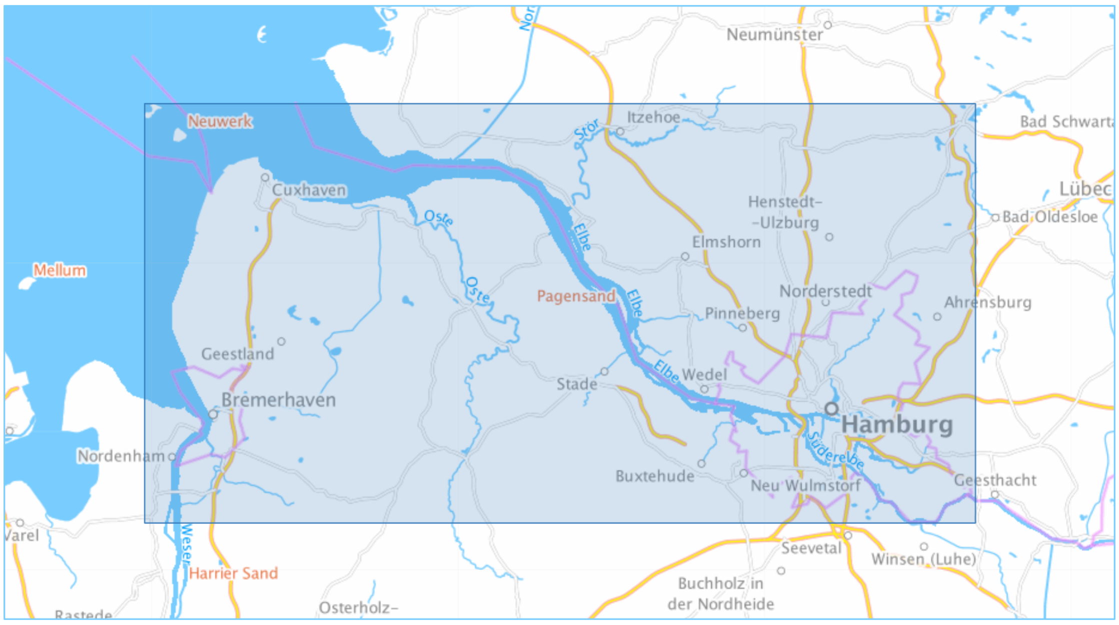


Figure 7.3.: The spatial coverage of Hamburg's CityGML LOD2 datasets. Source: The German Federal Agency for Cartography and Geodesy, 2022.

In addition, the datasets are provided as a collection of smaller, tiled files. The 2016 dataset consists of 788 tiles, while the 2022 dataset contains 887 tiles. The total size of the old and new datasets are 7.61 GB and 8.25 GB, respectively.

[8]https://metaver.de

In all these datasets, standardized roof shapes for buildings are automatically generated from airborne laser scanning, 3D building measurements from Amtliches Liegenschaftskatasterinformationssystem (ALKIS) (the official real estate cadastre information system for Germany) (AdV, 2008), or the aerial image-based digital surface model (LGV Hamburg, 2024). These shapes are then assigned to buildings based on their alignment with the available roof ridges. On the other hand, the building footprints are extracted from ALKIS and serve as the basis for the model's positional accuracy. The vertical accuracy is approximately ± 1 metre.

In this application scenario, the Hamburg CityGML datasets from 2016 and 2022 are first mapped onto corresponding graph representations, which are then matched to identify changes. Comprehensive interpretations are derived based on these found changes. The results of each of these three processes are presented in the following sections.

7.3. Results of the Mapping Process

Section 7.3 provides a summary of the distribution of nodes per label created in the graph database after the entire Hamburg CityGML datasets from 2016 and 2022 have been mapped onto graphs.

Table 7.1.: An overview of the node distribution per label in the graph database after the entire Hamburg CityGML datasets have been mapped onto graphs. CityGML and geometry class names are displayed in orange and green, respectively.

Node Label	**Quantity**	**Percentage**
ArrayList	114,811,007	21.582 %
DirectPosition	114,811,007	21.582 %
PosOrPointPropertyOrPointRep	55,605,096	10.453 %
ChildList	39,704,538	7.464 %
PosOrPointPropertyOrPointRepOrCoord	33,722,237	6.339 %
SurfaceProperty	23,402,240	4.399 %
StringAttribute	15,902,775	2.989 %
BoundingShape	12,741,837	2.395 %
Envelope	12,741,837	2.395 %
Exterior	10,977,064	2.063 %
LinearRing	10,977,064	2.063 %
Polygon	10,977,064	2.063 %
BoundarySurfaceProperty	10,974,366	2.063 %

MultiSurface	10,974,366	2.063 %
MultiSurfaceProperty	10,974,366	2.063 %
WallSurface	7,445,964	1.400 %
CurveProperty	6,628,977	1.246 %
LineString	6,628,977	1.246 %
Code	2,212,186	0.416 %
RoofSurface	2,078,017	0.391 %
CompositeSurface	1,450,810	0.273 %
Length	1,450,810	0.273 %
Solid	1,450,810	0.273 %
SolidProperty	1,450,810	0.273 %
MultiCurve	1,450,689	0.273 %
MultiCurveProperty	1,450,689	0.273 %
GroundSurface	1,450,385	0.273 %
BuildingPart	1,007,367	0.189 %
BuildingPartProperty	1,007,367	0.189 %
Building	758,429	0.143 %
CityObjectMember	758,429	0.143 %
ExternalObject	758,429	0.143 %
ExternalReference	758,429	0.143 %
Address	173,027	0.033 %
AddressDetails	173,027	0.033 %
AddressProperty	173,027	0.033 %
Country	173,027	0.033 %
CountryName	173,027	0.033 %
Locality	173,027	0.033 %
LocalityName	173,027	0.033 %
PostalCode	173,027	0.033 %
PostalCodeNumber	173,027	0.033 %
Thoroughfare	173,027	0.033 %
ThoroughfareName	173,027	0.033 %
ThoroughfareNumber	173,027	0.033 %
ThoroughfareNumberOrRange	173,027	0.033 %
XalAddressProperty	173,027	0.033 %
ThoroughfareNumberSuffix	56,727	0.011 %
StringOrRef	1,675	0.000 %
CityModel	2	0.000 %
Total number of nodes	531,975,220	100.000 %

The graph representations of the old and new Hamburg datasets contain approximately 532 million nodes in total. This number does not include auxiliary nodes utilized during the mapping process or in subsequent processes, such as nodes representing the R-tree used for spatial indexing, which accounts for another 80 million nodes.

Among these nodes, 758,429 are labelled as buildings and additional 1,007,367 as building parts. These objects contain 10,974,366 boundary surfaces in total, divided into 2,078,017 roof surfaces, 7,445,964 wall surfaces, and 1,450,385 ground surfaces.

The graph database for the Hamburg datasets occupies approximately 97 GB of disk space, excluding transaction logs. This space is allocated for the entire graph representations of the CityGML datasets, all graph elements resulting from the change and interpretation process, as well as database indexes, which allow for faster retrieval of thematic and spatial information among the graphs at the cost of additional storage space, as explained in Chapter 6.

7.4. Results of the Matching Process

An overview of more than 9 million edit nodes and base changes detected between the graph representations of the entire Hamburg CityGML datasets can be found in Table 7.2. More than half of these changes (56.07 %) are edit operations, including inserted nodes, deleted nodes, inserted properties, deleted properties, and updated properties, as introduced in Section 4.6. The remaining changes are geometric changes, such as translations and size changes of 2D surfaces. In addition, the matching process is also capable of detecting more complex geometric changes, such as a building being divided into smaller, adjacent buildings with an equivalent footprint. All interpretations in the subsequent processes are produced based on these edit operations and geometric changes.

7.4.1. Detected Edit Nodes

Figure 7.4 provides a summary of the distribution of the detected edit nodes among various city object types and attributes. The edit operations, represented by these edit nodes, include inserted nodes, deleted nodes, inserted properties, deleted properties, and updated properties. The CityGML objects affected by these changes are detailed as follows:

1. **Inserted Nodes**: The majority (79 %) of node insertions are detected among buildings and building parts, followed by thematic contents (11 %), such as object names, roof types, addresses, and measured heights, and geometric objects (10 %), which include terrain intersections, boundary surfaces, and solids.

Table 7.2.: An overview of the edit nodes and base changes detected in the Hamburg datasets.

Detected Change	**Quantity**	**Percentage**
Inserted Node	39,507	0.428 %
Deleted Node	1,780,953	19.279 %
Inserted Property	711	0.008 %
Deleted Property	768	0.008 %
Updated Property	3,357,679	36.347 %
Translation	1,143,966	12.383 %
Size Change	2,911,548	31.518 %
Top-level Split	2,692	0.029 %
Total number of changes	9,237,824	100.000 %

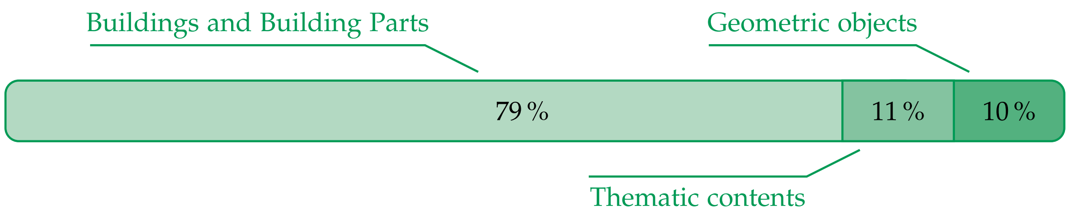
2. **Deleted Nodes**: Geometric objects like curves, surfaces, and solids account for 98 % of 1.8 million detected node removals. The remaining 2 % is found not only in thematic elements, such as generic attributes[9], roof types, addresses, and measured heights, but also in complex features like buildings and building parts.

3. **Inserted Properties**: All identified property insertions refer to a single attribute: the number of storeys above ground for buildings (99 %) and building parts (1 %). The newly added values span from 1 to 15 storeys, with the majority (72 %) of inserted numbers of storeys being 1 or 2.

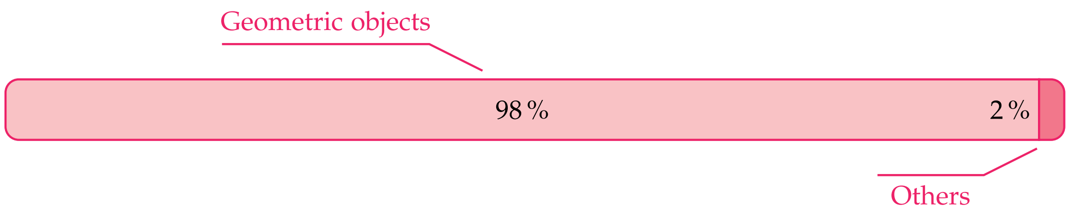
4. **Deleted Properties**: Similarly, all property removals exclusively relate to the number of storeys above ground for buildings (99 %) and building parts (1 %). The deleted values range from 1 to 16 storeys, with the majority (82 %) of the removed number of storeys being 1 or 2.

5. **Updated Properties**: In contrast to the similar composition of inserted and deleted properties, the distribution of more than 3 million detected updated properties is more diverse. Generic attributes, identifiers, and *creationDate* values each account for approximately a quarter of all updated properties, while updated measured heights make up another fifth. The remaining 6 % of all updated properties include roof type values and addresses, as well as the number of storeys above ground for both buildings and building parts.

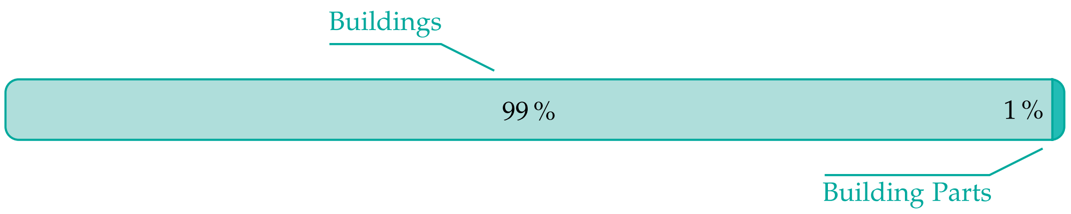
[9]Node insertions and removals refer to how city objects are represented. For example, while generic attributes are properties in the CityGML data model, they are stored as nodes in this thesis.



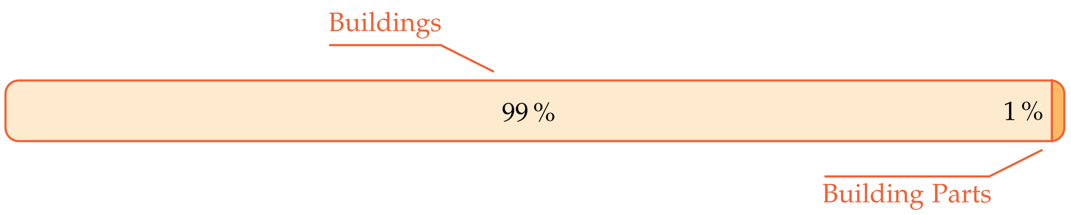
(a) Distribution of inserted nodes



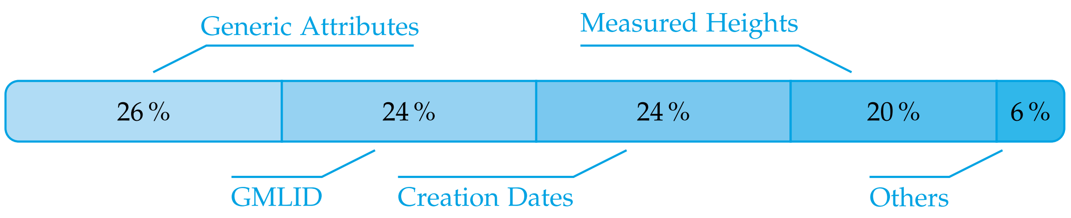
(b) Distribution of deleted nodes



(c) Distribution of inserted properties



(d) Distribution of deleted properties



(e) Distribution of updated properties

Figure 7.4.: The distribution of the detected edit nodes in the Hamburg datasets.

Figure 7.5 visualizes the spatial distribution of top-level insertions and removals over the entire area of Hamburg, where the 2D bounding boxes of inserted buildings are displayed in green, while those of deleted buildings are shown in red. The figure shows significantly more green bounding boxes than red ones, as there are approximately 30 thousand inserted buildings compared to 23 thousand deleted buildings. Leveraging such spatial visualization, 'hotspots' with a strong increase in new inserted buildings, or vice versa, can be visually identified.

Figure 7.6 illustrates one such hotspot with a significant increase in new buildings. The deletion and insertion of buildings may indicate an improvement in the building locations or actual demolition of old buildings and construction of newer ones.

Figure 7.7 shows the satellite images from 2015 and 2021 of the same area of Hamburg given in Figure 7.6. The buildings displayed in the 2015 image align with the old buildings (red) deleted from the datasets, while the buildings depicted in the 2021 images align with the new buildings (green) inserted to the datasets. Therefore, this shows a rapid urban development of the area, where all old buildings have been demolished and replaced with newer, more modern ones.

All updated generic attributes are observed to be string attributes. In the Hamburg datasets, these generic string attributes provide additional thematic information for buildings, building parts, and roof surfaces, as summarized in Table 7.3. Unlike others, the attributes for source ground elevation and source roof height (*DatenquelleBodenhoehe* and *DatenquelleDachhoehe*, respectively) can belong to both buildings and building parts. The internal composition of these updated attributes and the coverage of their source types in the datasets are summarized in Table 7.4.

Table 7.3.: The distribution of updated generic string attributes by name in the Hamburg CityGML datasets.

Attribute Name	**English Translation**	**Quantity**	**Percentage**
Gemeindeschluessel	Municipality key	347,574	39.667 %
DatenquelleBodenhoehe	Source ground elevation	318,179	36.312 %
DatenquelleDachhoehe	Source roof height	167,800	19.150 %
Flaechengroesse	Surface area	39,033	4.455 %
Flaechenneigung	Surface inclination	3,619	0.413 %
Flaechenrichtung	Surface orientation	23	0.003 %
	Total number of changes	876,228	100.000 %

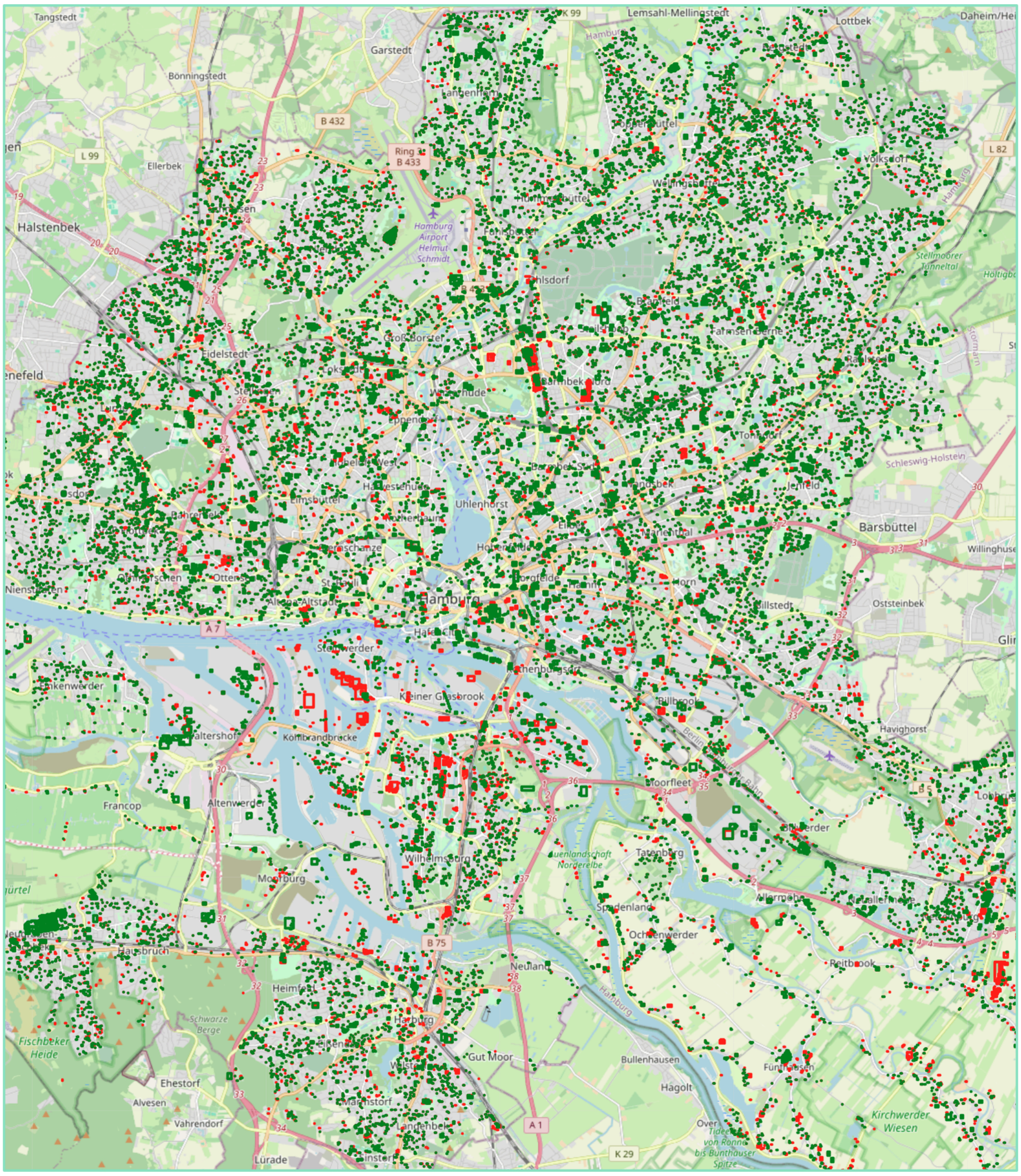


Figure 7.5.: An illustration of inserted (green) and deleted (red) buildings detected in the Hamburg datasets between 2016 and 2022. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

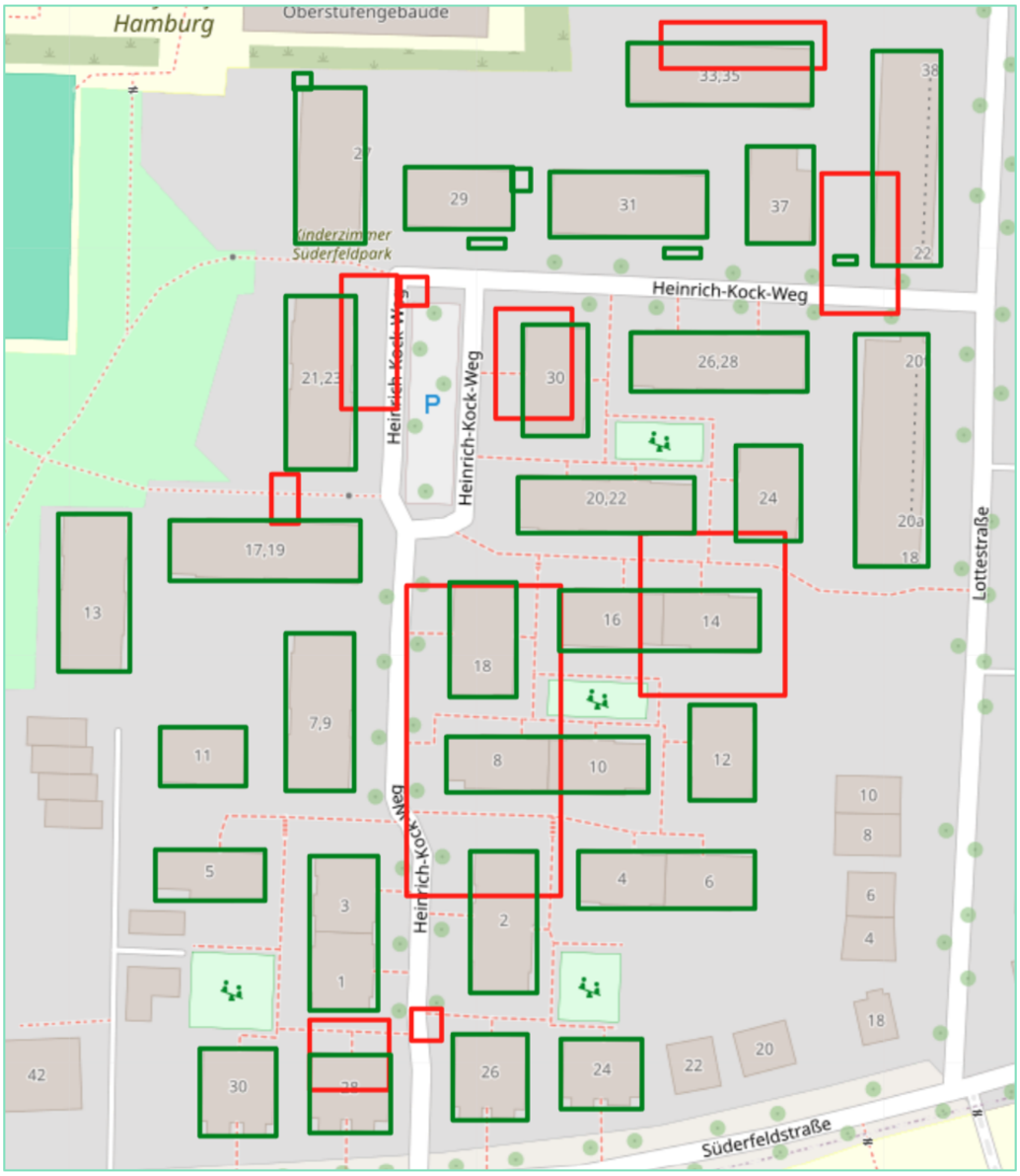
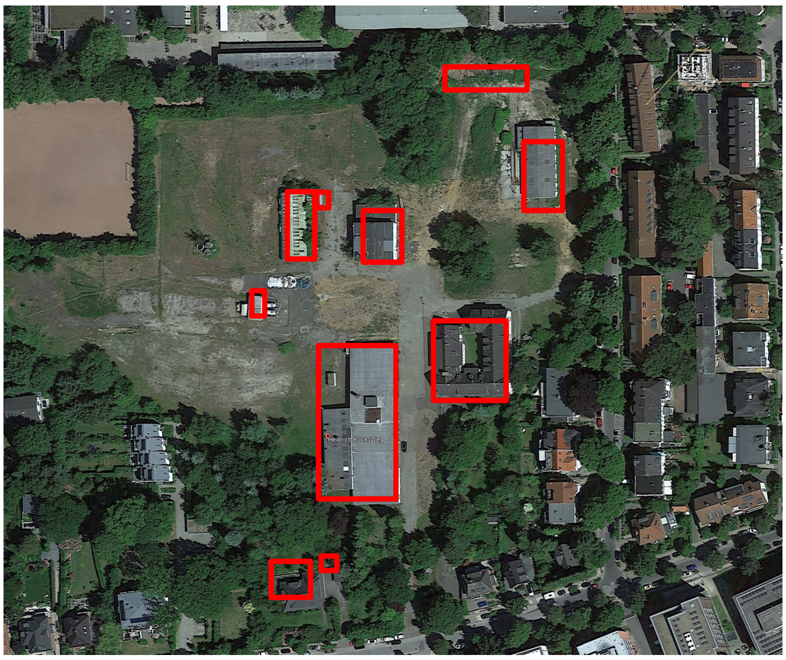


Figure 7.6.: An example of inserted (green) and deleted (red) buildings of an excerpt area from the Hamburg datasets. The deletion and insertion of buildings may indicate an improvement in the building locations or actual demolition of old buildings and construction of newer ones. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).


(a) From 2015

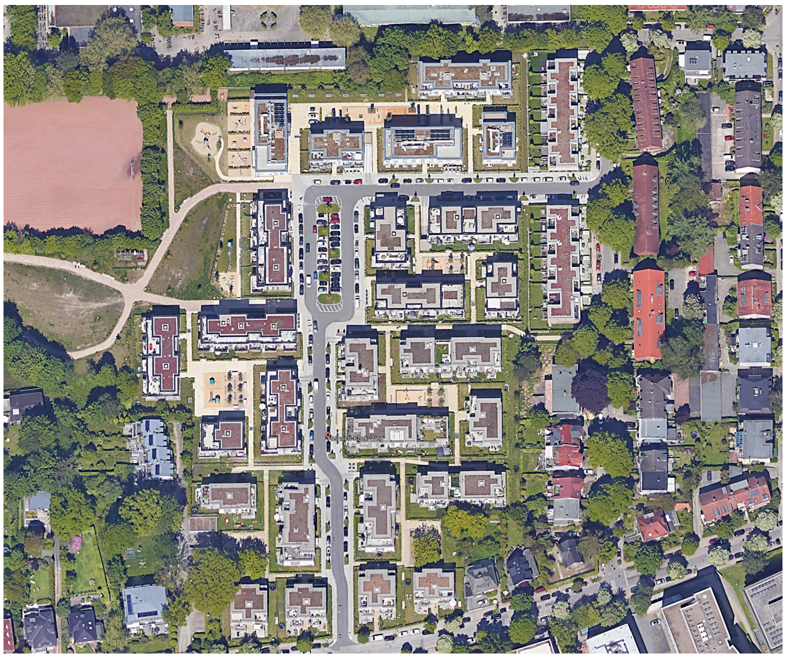

(b) From 2021

Figure 7.7.: Satellite images between 2015 and 2021 of the area given in Figure 7.6. Visualized with historical data from 2015 and 2021 in Google Earth Pro.

Table 7.4.: Internal composition and coverage of updated generic string attributes by source type in the Hamburg CityGML datasets. The middle column shows the percentage of each source type to which the updated generic string attributes belong, while the right column provides the coverage of those source types within the entire datasets.

Attribute Name	**Internal Composition**		**Dataset Coverage**	
Gemeindeschluessel	B 100.000 %		B 99.598 %	
DatenquelleBodenhoehe	B 27.048 %	P 72.952 %	B 24.660 %	P 47.630 %
DatenquelleDachhoehe	B 27.296 %	P 72.703 %	B 13.125 %	P 25.033 %
Flaechengroesse	R 100.000 %		R 8.082 %	
Flaechenneigung	R 100.000 %		R 0.749 %	
Flaechenrichtung	R 100.000 %		R 0.005 %	

B Buildings P Building Parts R Roof Surfaces

7.4.2. Detected Geometric Changes

The matching process detected more than 1 million translations and approximately 3 million size changes in the geometric surfaces of the Hamburg datasets. Each translation and size change contains the information about the translation offsets or change in sizes across all three dimensions. For instance, a translation vector of $(1, 1, 0)$ indicates a shift of 1 m along each horizontal dimension x and y. On the other hand, a resize vector of $(0, 0, 1)$ represents a vertical change in size, where the surface's height has been increased upwards by 1 m.

An overview of all detected surface translations can be found in Figure 7.8. The top-left figure visualizes the distribution of these translations by their total lengths, which are calculated as the Euclidean lengths of their respective translation vectors. As visualized, the majority of these translations fall between 0 m and 0.5 m, with decreasing number of occurrences as the translation length increases. However, even at the farthest value interval of between 2.5 m and 3 m empirically selected for the Hamburg datasets, approximately 10 thousand surfaces are still affected by this change.

The bottom-right figure displays a similar distribution of these surface translations by their vertical offsets, with positive and negative values indicating an increase and a decrease in height, respectively. As explained in Section 4.5.4, the current implementation allows for the matching of two surfaces that are located within a maximum distance of 3 m. Beyond this threshold, they are considered as unrelated. This is to reduce the number of false matches among unrelated, parallel surfaces.

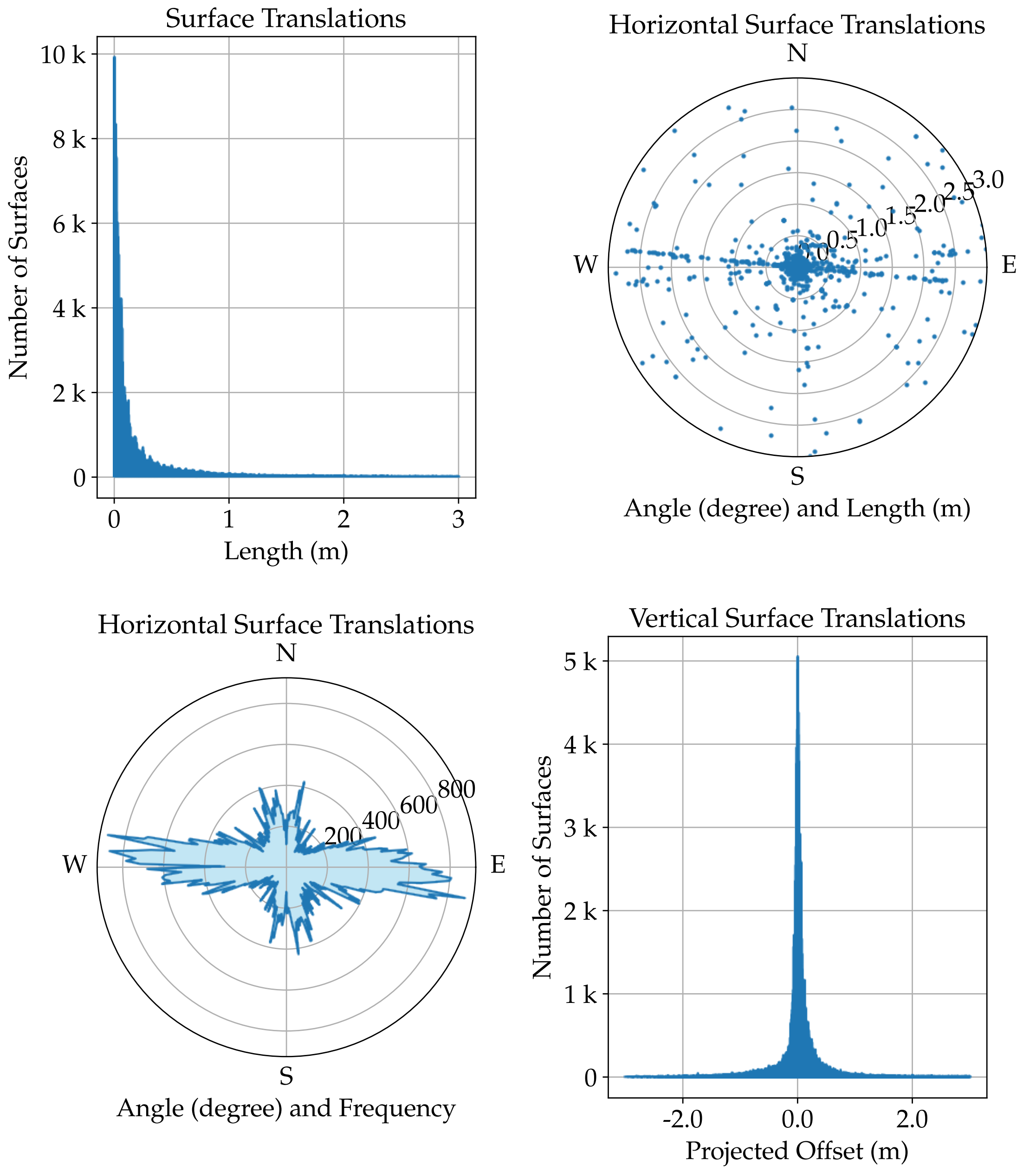


Figure 7.8.: An overview of more than 1 million surface translations detected in the Hamburg datasets. Their distributions are visualized by total lengths (top), horizontal orientations and projected lengths on *xy*-plane (top-right), horizontal orientations and frequency (bottom-left), and vertical offsets (bottom-right). The current implementation allows for matching two surfaces with a maximum distance of 3 m.

A notable observation emerges from the top-right and bottom-left figure of Figure 7.8. The bottom-left figure shows that the majority of horizontal translations occur in the East-West directions. More specifically, the top-right figure shows the consistent movement of more than 13 thousand surfaces approximately 10° clockwise from both the East and West directions, indicated by the alignment of the points in the shape of a straight line plotted in the figure. This could imply an adjustment or improvement in the positions of buildings and building parts in the city model relative to the East-West axis.

On the other hand, Figure 7.9 provides an overview of approximately 3 million surface size changes detected in the Hamburg datasets. While most of size changes along the x and y-dimension occur simultaneously, as shown in the top-left figure, the majority of vertical size changes occur independently of changes in the other dimensions, as shown in the top-right and bottom-left figure. These observations coincide with the facts that (1) most size changes in ground and roof surfaces take place in the xy-plane, thus resulting in changes in both the x and y-dimension, and (2) wall surfaces, which account for the majority of surfaces in the city model, are typically resized vertically as they are often directed perpendicularly to the ground. A size change in a wall surface that also involves changes in the x and y-dimension could suggest an additional horizontal enlargement or downsizing. The Venn diagram at the bottom-right shows 470 such size changes with non-zero values in all dimensions.

Since (1) geometric changes are computed based on the axis-aligned bounding boxes of surfaces, and (2) the increased complexity when also including rotations on top of translations and size changes, which could require the calculation of transformation matrices, the implementation of this study omits rotations between surfaces.

In addition to the translations and size changes mentioned above, the matching process is also capable of detecting split changes in buildings, which result from a division of a larger building into smaller, adjacent buildings, with total bounding boxes corresponding to that of the original one. Figure 7.10 gives an overview of such split changes identified in the Hamburg datasets. Figures 7.11 and 7.12 then provide some example use cases of these split changes within the datasets.

As logical components of buildings, building parts can also be detached from their original buildings to become separate buildings. This effect is similar to that of the observed changes in split buildings. Thus, to investigate whether the new, smaller buildings in a split change were originally building parts, all old buildings attached with a split change are analysed. As a result, nearly half of all detected split buildings do not have any building parts. Among the remaining split buildings, not all their building parts spatially coincide with the new, smaller buildings, as illustrated in Figures 7.13 and 7.14. Therefore, the detected split changes in building were not solely due to the logical division of building parts from their original buildings.

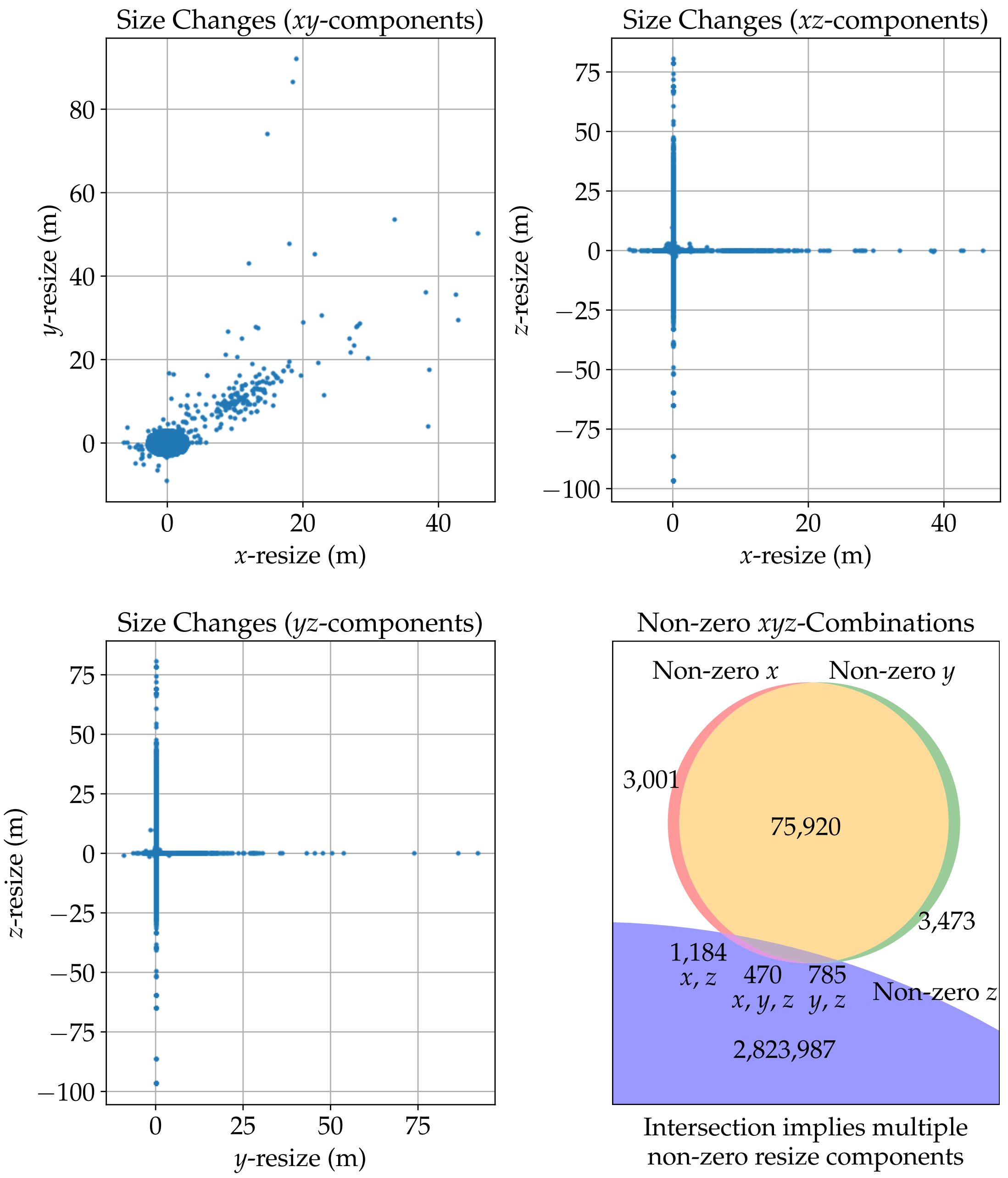


Figure 7.9.: An overview of approximately 3 million surface size changes detected in the Hamburg datasets. They are visualized by their components *xy* (top-left), *xz* (top-right), and *yz* (bottom-left). All combinations of these three resize components are illustrated in a Venn diagram (bottom-right).

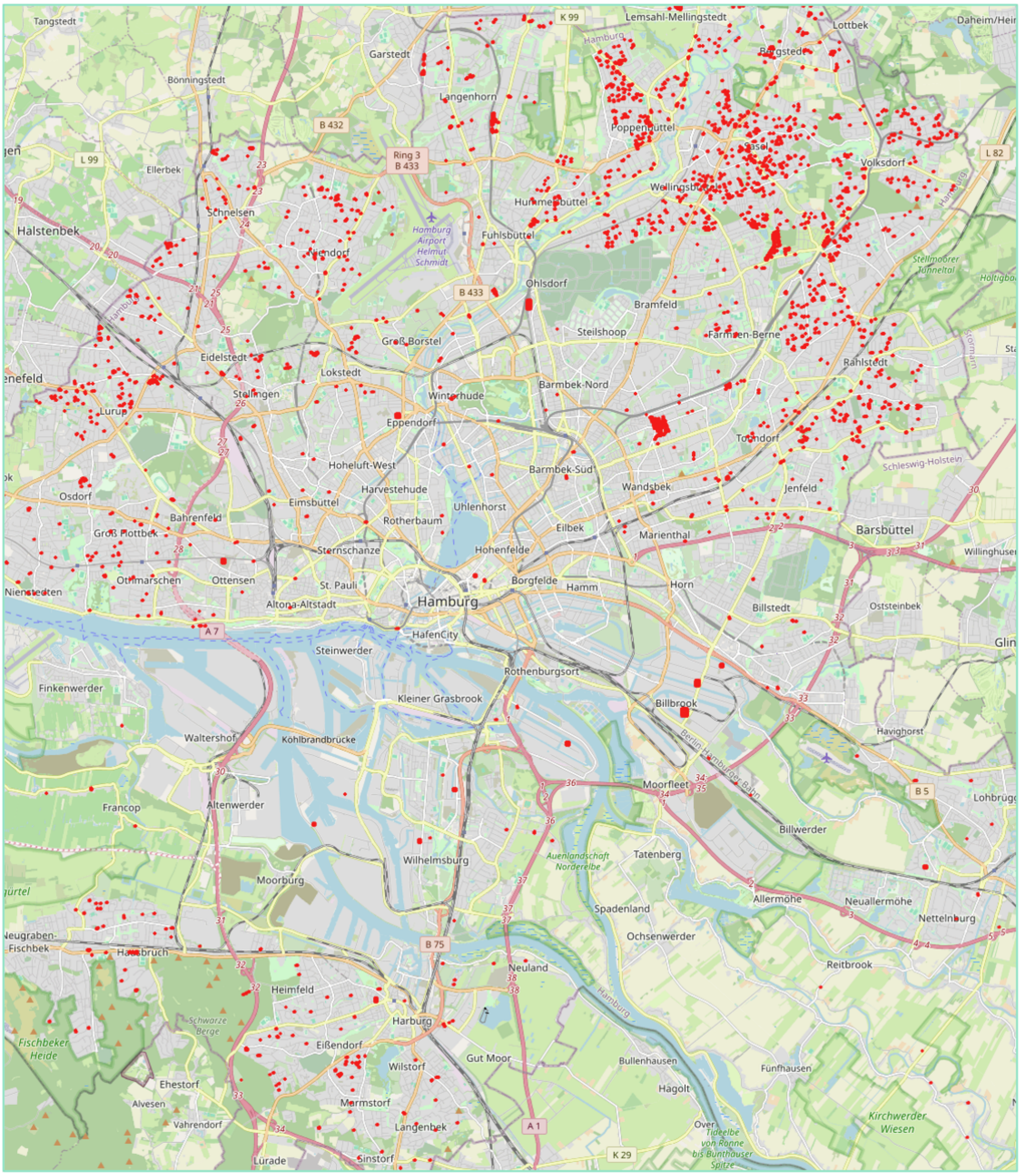


Figure 7.10.: An illustration of split changes among buildings (red) in the Hamburg datasets between 2016 and 2022. A split change is observed when an existing building is replaced by smaller, adjacent buildings with equivalent footprint. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

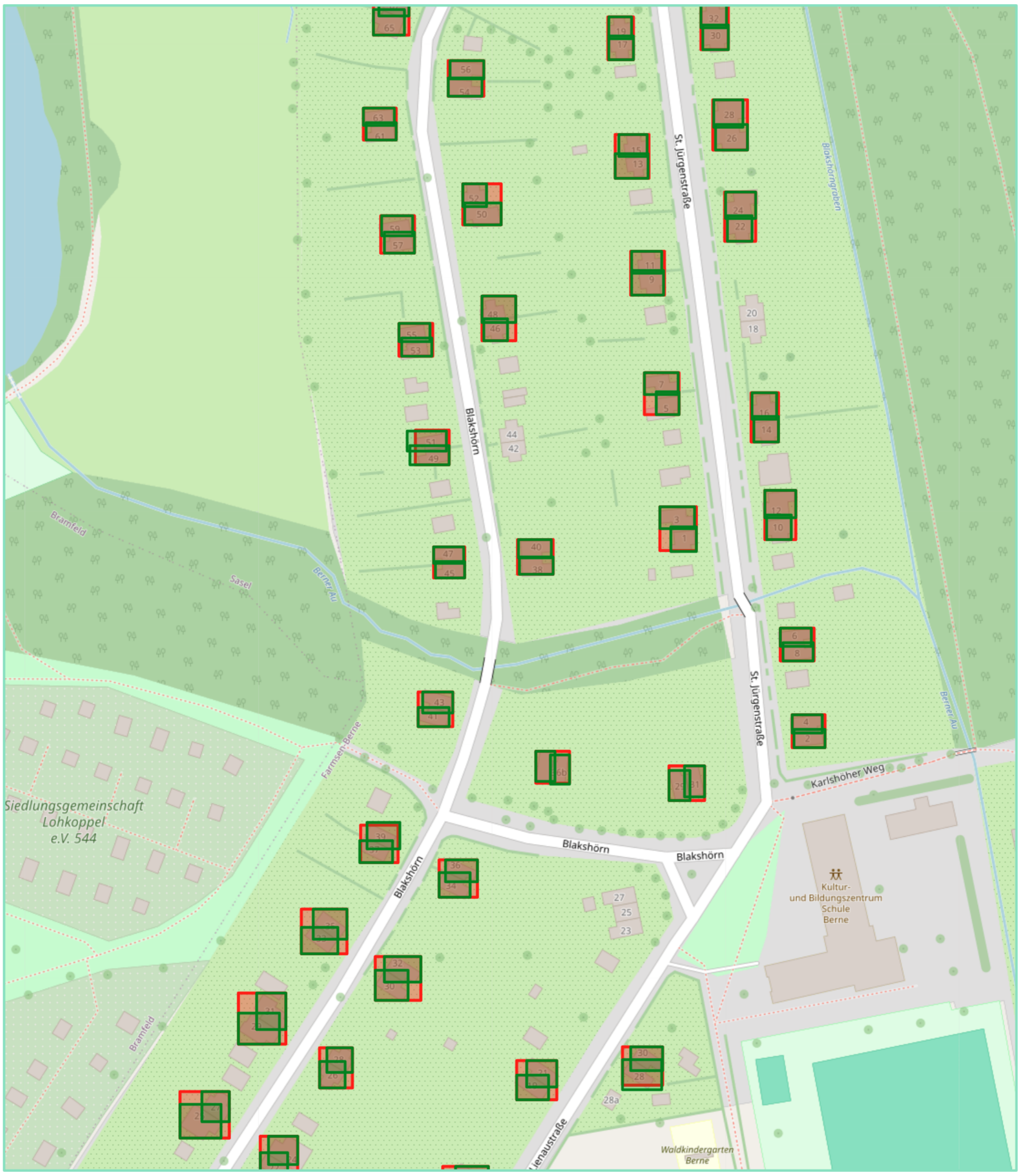


Figure 7.11.: An example of building split changes of an excerpt area (Blakshörn) from the Hamburg datasets. In this illustration, each old building (red) has been replaced by two smaller, adjacent buildings (green) with equivalent footprint. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

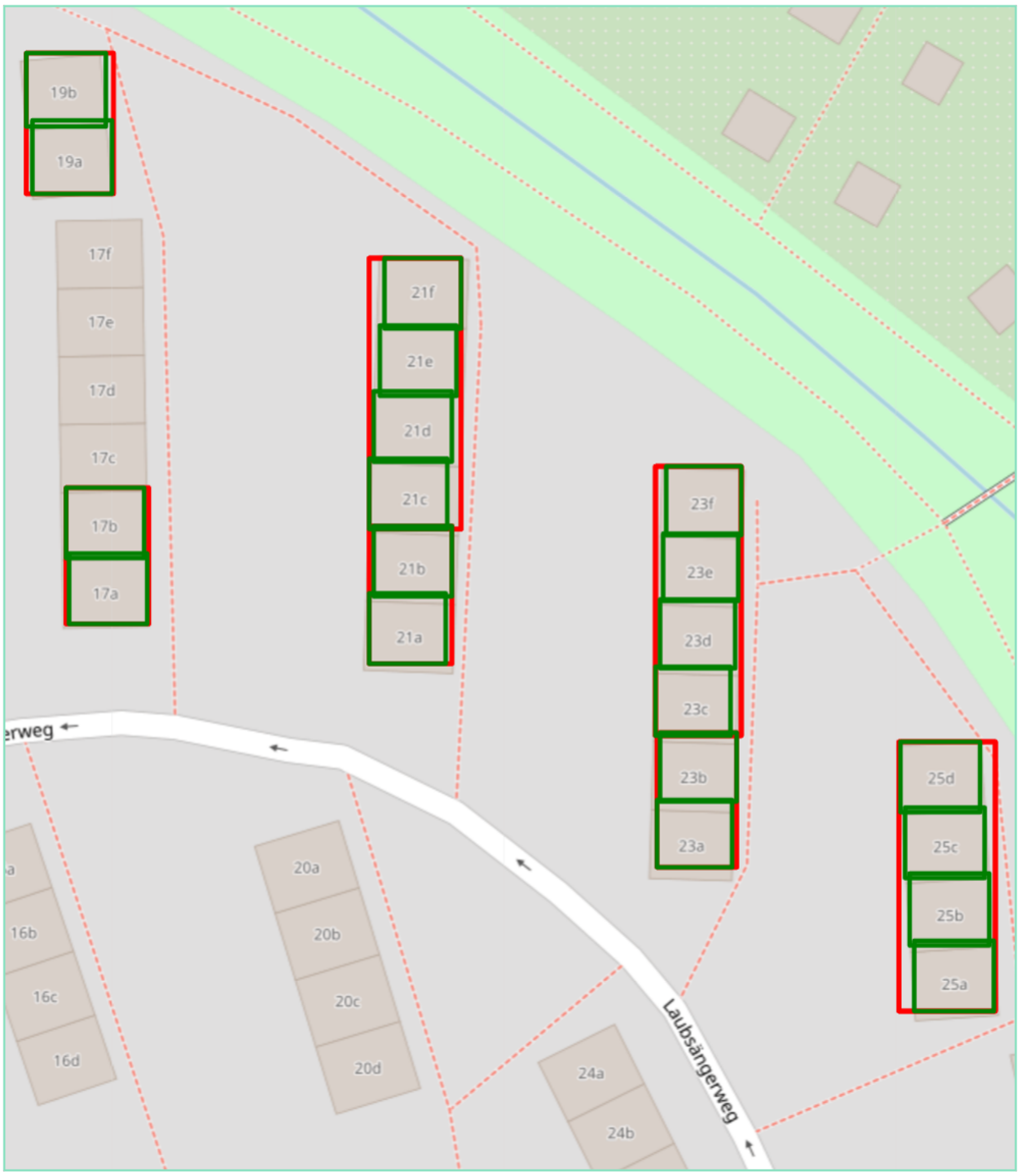


Figure 7.12.: An example of building split changes of an excerpt area (Laubsängerweg) from the Hamburg datasets. In this use case, the split changes (green) among buildings (red) align with the creation of new addresses, such as the new house numbers 17a-b, 19a-b, 21a-b, 21c-f, 23a-b, 23c-f, as well as 25a-d. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

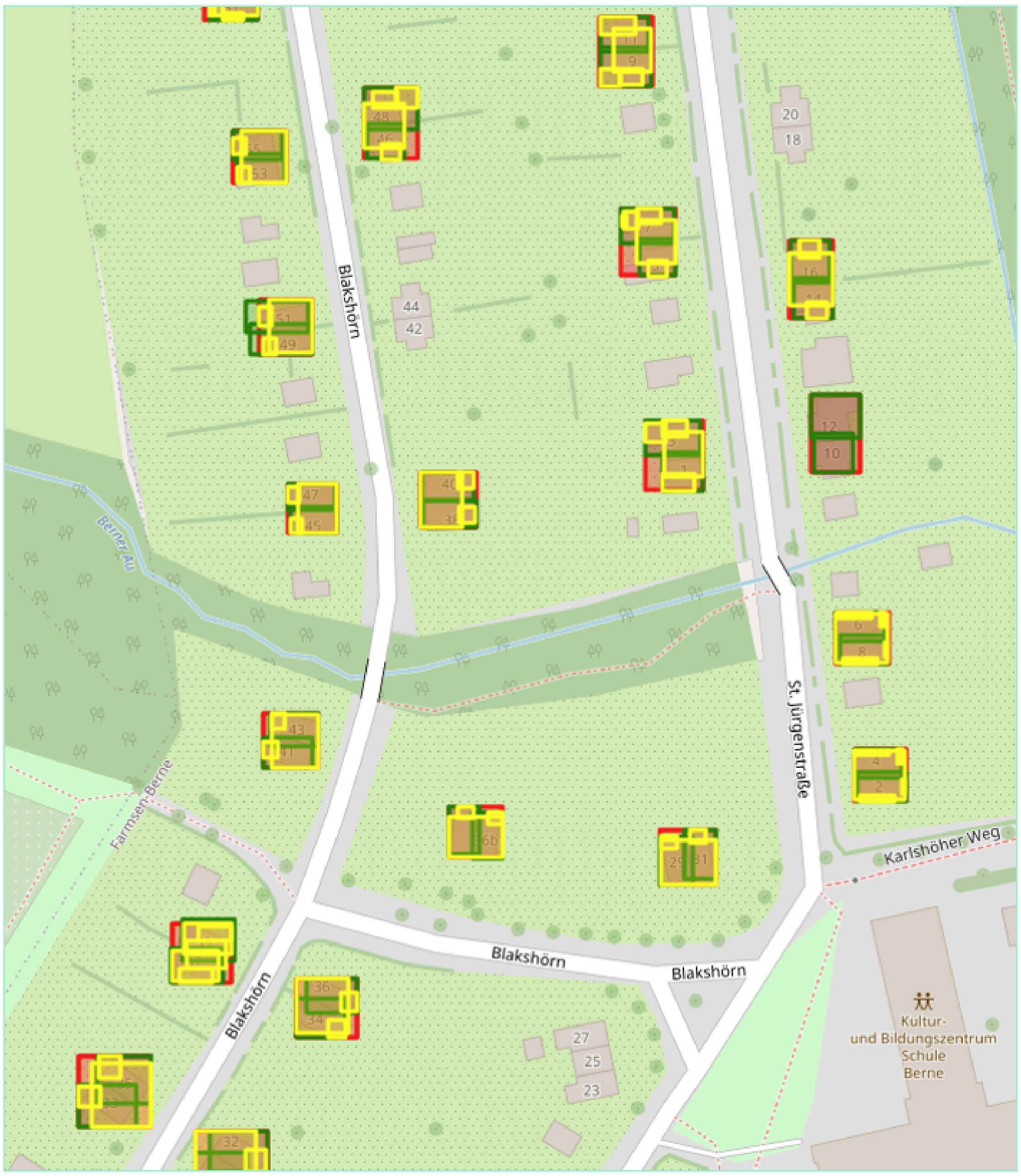
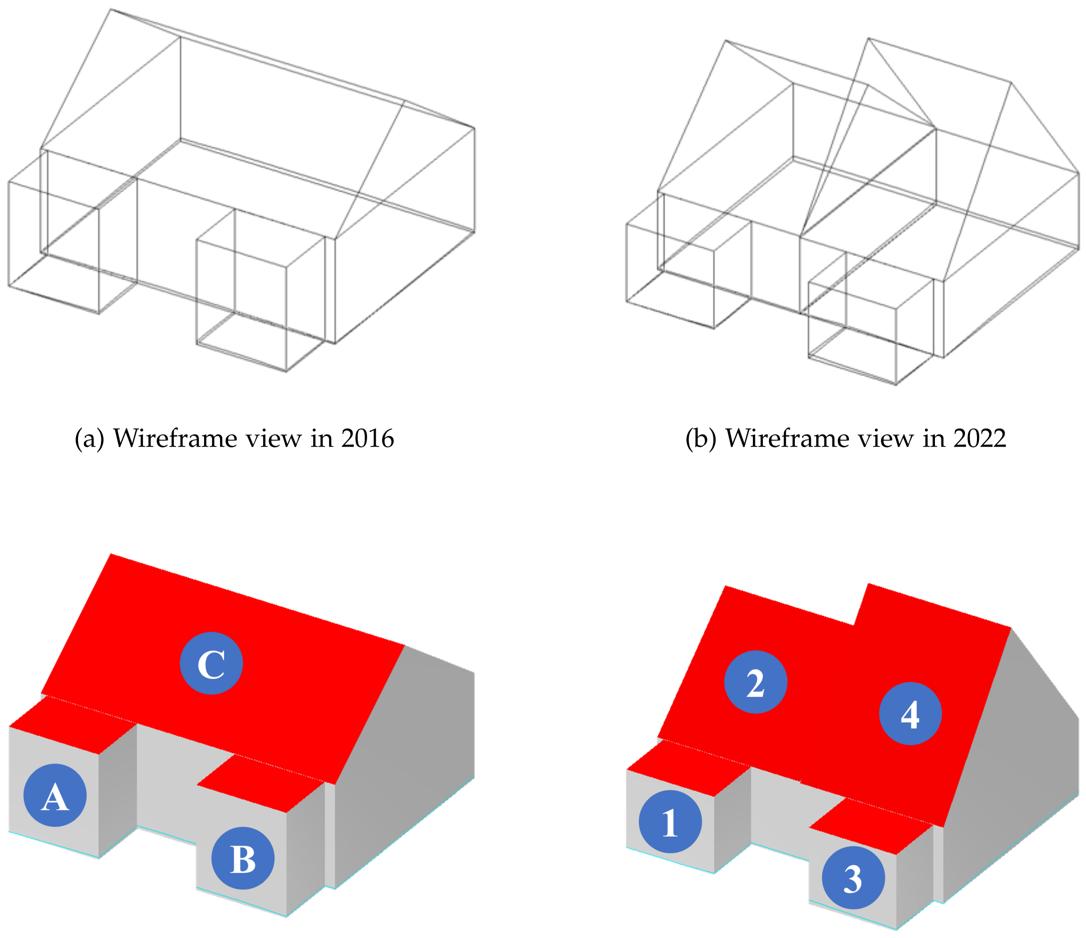


Figure 7.13.: An example of building split changes based on the area displayed in Figure 7.11, additionally visualized with building parts (yellow) from the old datasets (red). As shown, not all building parts spatially coincide with the new split buildings (green). A more detailed comparison can be achieved by examining their precise 3D geometries (not depicted). Visualized with Folium, Leaflet, and OpenStreetMap (OSM).



(a) Wireframe view in 2016

(b) Wireframe view in 2022

(c) Visualization in 2016

(d) Visualization in 2022

Figure 7.14.: Visualization of 3D geometries of a split building and its building parts, as shown in Figure 7.13. The building is located at the intersection of Blakshörn and St. Jürgernstraße, with all views depicted from the north-west perspective. In the 2016 dataset (top left and bottom left), this single building is composed of three building parts. However, in the 2022 dataset (top right and bottom right), this building is divided into two smaller buildings, each consisting of two building parts. This example illustrates that not all building parts spatially coincide with the new split buildings. Visualized with the KITModelViewer (KIT IAI, 2024).

Figures 7.15 and 7.16 illustrate an area in Hamburg where buildings have been split into between two and four new, smaller buildings. All these split changes of top-level features are classified as the changes *ObjectSplit*, as discussed in Section 4.5.6 and modelled in Section 5.2. Without these, more edit nodes would have been generated: one top-level change *DeletedNode* for the old building, and at least two top-level changes *InsertedNode* for the new buildings.

Therefore, based on the detected insertions and deletions of building objects, the following interpretations can be derived:

1. **Building Replacement**: If a building from the old datasets is deleted and a corresponding building from the new datasets is inserted, this typically indicates a building replacement operation. This can be further categorized into:
    a) **Improvement in Buildings' Locations**: A replaced building object may suggest an improvement in its spatial extent during the given time period, possibly due to more accurate measurements.
    b) **Building Division**: An older, larger building may be replaced by several smaller buildings with equivalent combined footprints. This may indicate a logical or physical division of the old buildings in the real world.
2. **Building Construction and Demolition**: These insertions and deletions may be the actual result of newly constructed or demolished buildings in the city.

Like building division, a similar approach can be applied to detect merge changes, where two or more buildings from the older datasets are merged into a new, larger one. The current implementation only includes the handling of building split changes.

7.5. Results of the Interpretation Process

To provide a deeper understanding of the thematic and geometric changes detected in the Hamburg datasets between 2016 and 2022, rules for detecting change patterns among them must first be defined. The interpreter then applies these rules to all changes and contents stored in the graph database. Both the intermediate and final results of the interpretation process are stored in the same graph database. They are either represented as new interpretation nodes if the patterns have been successfully identified, or as memory nodes for patterns that have not yet been fully matched. These memory nodes serve as a working memory created to keep track of the component changes collected so far that are required for the creation of the next interpreted changes, as dictated by the given change pattern rules. Lastly, these matched change patterns and created interpretation nodes are then utilized and analysed, so that only the most relevant changes are provided to specific groups of stakeholders.

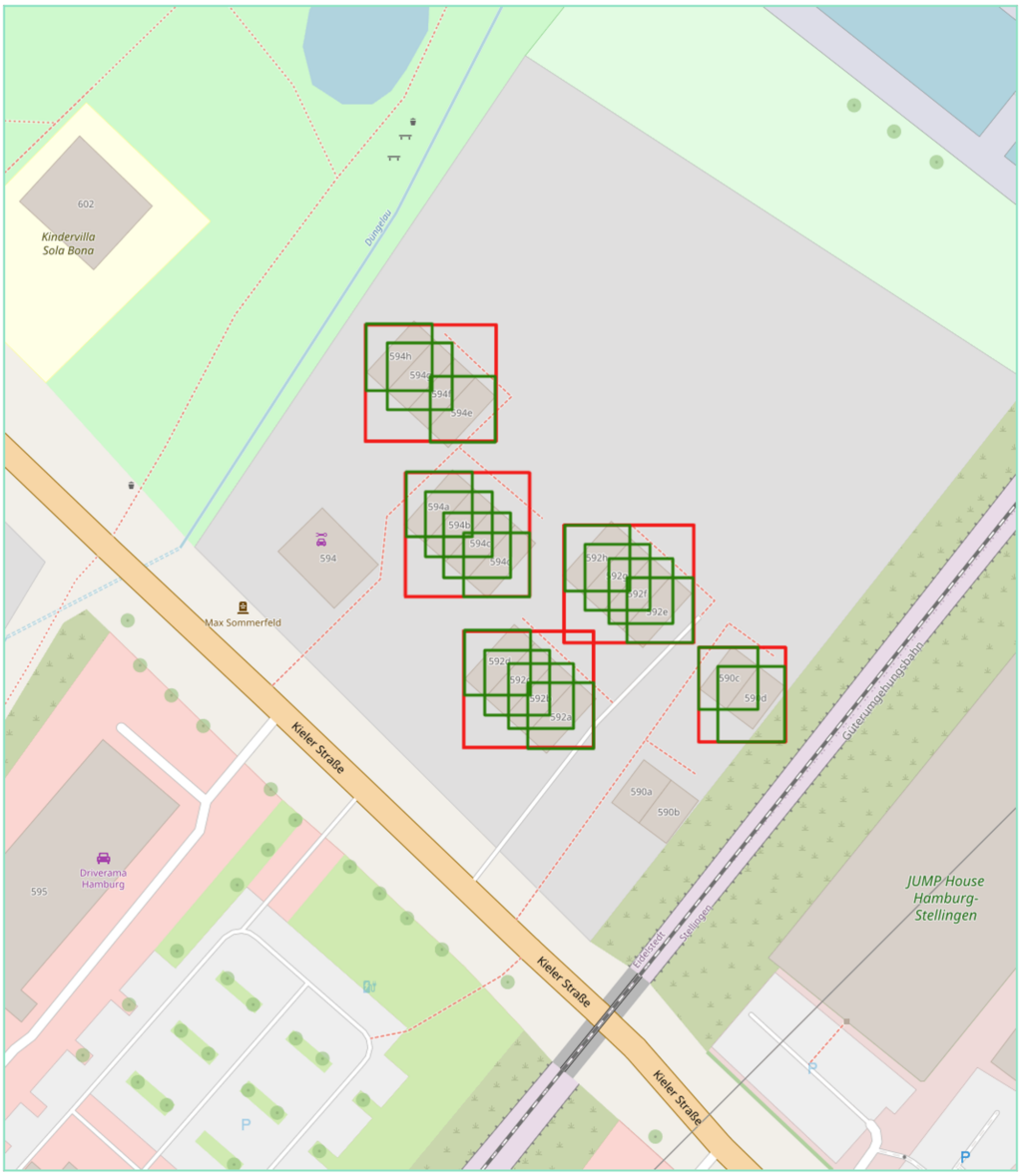


Figure 7.15.: An example of building split changes of an excerpt area (Kieler Straße) from the Hamburg datasets. Despite the total area of the green bounding boxes not equating to that of the original red ones, as these bounding boxes are axis-aligned, such cases can still be detected by the matching process. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

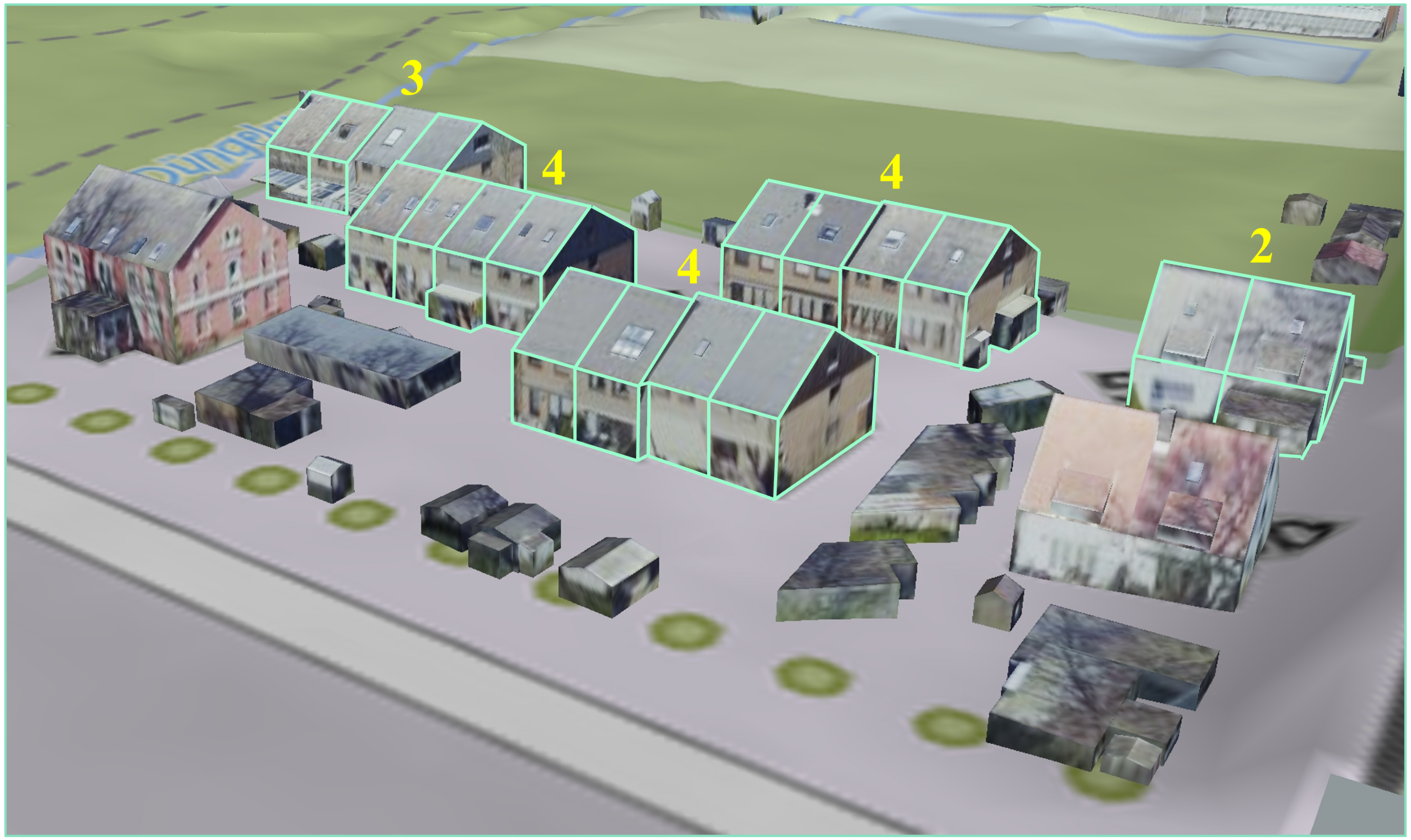


Figure 7.16.: A 3D view of building split changes of an excerpt area (Kieler Straße) from the Hamburg datasets based on Figure 7.15. The buildings shown have been split into between two and four smaller adjacent buildings (yellow texts and cyan wireframes). Source: Landesbetrieb Geoinformation und Vermessung Hamburg, 2024.

7.5.1. Employed Change Pattern Rules

The selection of change patterns and the corresponding rules to describe them may vary based on the specific use cases and analysis requirements. Some of the most prominent rules for detecting change patterns, empirically selected and employed in this thesis, are summarized as follows:

1. **Pattern *id***: If an updated property has been detected for a building or building part and the property name is *id*, it indicates that the identifier of that building or building part has been updated. If this change is observed in all non-deleted buildings and building parts, it is classified as a global change.

   → A change in the identifiers of all top-level features within the city model or a region may suggest a fundamental change in the software or the data operators.

2. **Pattern *creationDate***: If an updated property has been detected for a building or building part and the property name is *creationDate*, it indicates that the modification date of that building or building part has been updated. If the property *creationDate* has been updated for all non-deleted buildings and building parts, it is classified as a global change.

   → A change in the modification date of a building or building part may potentially indicate further changes within its sub-elements. If this type of change is observed in all objects within a specific region, it could imply that the corresponding segment of the city model has been extracted from the city model, modified, and then reintegrated back into the city model, independent of the rest of the datasets.

3. **Pattern Coded Values**: The CityGML data model allows for the storage of thematic information, such as the class, function, and usage of various city objects. For buildings, it can also store additional information about their roof types. These values are coded as numbers or strings, each assigned with a distinct meaning according to predefined code lists (Gröger et al., 2012). If an updated property has been detected in the properties *class*, *function*, *usage*, or *roofType*, it indicates a change in the assigned class, function, or usage of a city object, as well as the roof type of a building, respectively.

   → Changes in these properties represent significant thematic changes, even though they may not necessarily coincide with geometric changes, as they dictate the important information of class, function, usage, and roof types of city objects that cannot be derived elsewhere. For example, a change in a building's function from 1000 (residential) to 1150 (commercial) indicates either a repurposing or a correction of a previously incorrect classification of that building. In addition, a change in a building's roof type from and to 1000 (flat roof) may indicate an increase or decrease in the building's measured height. If such changes occur consistently for all city objects within a region, it may suggest a policy of repurposing objects in that area. Given the hundreds of distinct values available in the predefined code lists, a comprehensive analysis of changes for each possible pair of code values would require a massive matrix with quadratic space. Therefore, only some relevant code values for city objects are considered in this experiment.

4. **Other Thematic Patterns**: When a change in the form of an insertion or deletion is detected for a thematic element of a building, such as its name, address, number of storeys, and measured height, this change can be directly interpreted as an enrichment or a loss of thematic information of that building.

   → In contrast to insertions and deletions, updates of a thematic element are generally more difficult to interpret. This is due to the additional information available about both the older or newer thematic values of the building, compared to only one temporal state in insertions and deletions. On the other hand, the interpretation of any changes to generic attributes requires prior knowledge about the meaning of their names and values.

5. **Pattern Translations**: If a translation has been detected for a surface, it implies a movement in a boundary roof, wall, or ground surface of a building or building part, depending on the surface type. If such a shift with a consistent translation vector is observed across all roof, wall, or ground surfaces of a building or building part, it indicates that all roof, wall, or ground surfaces of that building or building part have been translated by the same vector, respectively. If all three types of boundary surfaces have been marked with the same translation, it suggests that the entire building or building part has been relocated by that vector. If all buildings and building parts within a specific region of a city model have been moved by an identical vector, it may imply a systematic shift of that entire region.

   → A translation is classified as vertical if its corresponding vector solely has a non-zero z-component. Conversely, a translation is classified as horizontal if its corresponding vector does not have a z-component (or it is zero). In addition, translations in surfaces often also involve size changes of other adjacent surfaces. For example, a raised roof is supported by the vertical enlargement of all its corresponding wall surfaces.

6. **Pattern Size Changes**: If a size change has been detected for a surface, it indicates a resizing of a boundary roof, wall, or ground surface of a building or building

part, depending on the surface type. If such a resizing with consistent margins (in 3D) is observed across all roof, wall, or ground boundary surfaces of a building or building part, it indicates that all roof, wall, or ground boundary surfaces of that building or building part have been resized by the same margins, respectively.

$\rightarrow$ A size change is classified as vertical if its corresponding margins solely have a non-zero z-component. Conversely, a size change is classified as horizontal if its corresponding margins do not have a z-component (or it is zero). Geometrically, it is impossible for all boundary surfaces of a building or building part to be resized by identical margins, as these surfaces have different orientations but collectively form a closed three-dimensional shape. Similar to translations, size changes in surfaces often also involve size changes and translations of other adjacent surfaces. For example, a horizontally enlarged roof and ground surface are supported by horizontally translated or horizontally enlarged wall surfaces.

7. **Pattern Raised Roofs**: The roof surfaces of a building or building part are considered elevated vertically (without changing their shapes and sizes) by $\Delta h > 0$, if (1) all roof surfaces have been translated upwards by Δh, (2) all wall surfaces have been enlarged vertically by Δh, and (3) the value *measuredHeight* of that building or building part has also been increased by Δh.

   $\rightarrow$ Similar pattern rules can be defined for lowered roofs with $\Delta h < 0$. In real-world scenarios, as will be shown in the Hamburg datasets, it is possible that the ground surfaces may also have been moved vertically, in addition to the vertical shift of the roofs. In such cases, the new wall heights and the updated value *measuredHeight* should account for both the vertical translations. In most cases, except for actual renovations or events like earthquakes that cause buildings to rise or sink, minor shifts in roof and ground surfaces are most likely due to improved surface measurements or enhanced terrain models.

For each of the aforementioned rules, a scope is calculated to determine whether the associated patterns apply locally, within specific regions, or globally. These rules can be adjusted and extended to accommodate specific use cases.

Listing 7.5 gives an example Cypher query for defining and matching change patterns on the building property *creationDate*. All rule nodes are labelled as *RULE*, while their relationships have the unique type *AGGREGATED_TO*, indicating the aggregative nature of these patterns. Within a Cypher query, rule nodes can be created once and then referenced by their unique names, such as '*updated_property*' and '*updated_building_creation_date*'. This allows them to be reused in subsequent pattern rules, ensuring that only one rule node per type exists in the rule network. The corresponding rule network created from this Cypher query can be found in Figure 7.17.

```
// Pattern in creationDate of buildings
MERGE (updated_property:RULE { // unique node names within the query
  change_type: 'UpdatedProperty' // unique change type within the database
})-[:AGGREGATED_TO {
  next_content_type: 'Building', // the label of the next content node
  search_length: 0, // the updated property is inside the content node
  conditions: 'NAME === "creationDate"', // JavaScript syntax
  propagate: 'RIGHT_VALUE', // propagate only updated value
  weight: 1
}]->(updated_building_creation_date:RULE {
  change_type: 'UpdatedBuildingCreationDate',
  calc_scope: 'RIGHT_VALUE' // calculate scope over this property
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel', // attach scope nodes to CityModel node
  scope: 'global',
  propagate: 'RIGHT_VALUE'
}]->(global_updated_building_creation_dates:RULE {
  change_type: 'GlobalUpdatedBuildingCreationDates'
})
```

Listing 7.5: An example Cypher query for defining pattern rules on the property *creationDate* of buildings. These notations are explained in Section 5.3.5. A graph visualization is shown in Figure 7.17.

Figure 7.18 presents a visualization in Neo4j Browser of the entire rule network employed in this research. The highest-level rule nodes (sink nodes) represent the patterns of elevated roofs or moved buildings and building parts. Conversely, the lowest-level rule nodes (source nodes) symbolize the base changes, such as the surface size changes and translations, as well as updated node properties. Based on these three lowest-level rule nodes, all pattern rules in this use case can be established. The full description of this rule network in Cypher is available in Listing B.1.

7.5.2. Unchanged Buildings

To better evaluate the results of the interpretation process for the patterns provided above, one of the first steps involves investigating the buildings that have remained unchanged between 2016 and 2022 in the Hamburg datasets. A building is considered unchanged if no change node is attached to any of its sub-elements. Listing 7.6 provides a Cypher query used to search for such buildings within the old CityGML datasets.

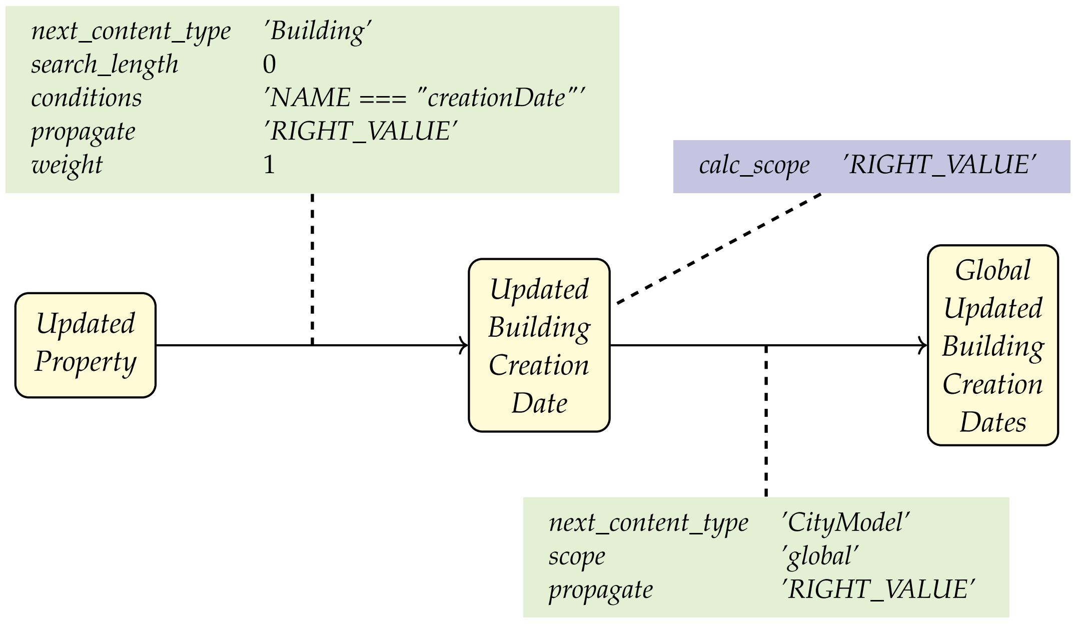

next_content_type 'Building'
search_length 0
conditions 'NAME === "creationDate"'
propagate 'RIGHT_VALUE'
weight 1
calc_scope 'RIGHT_VALUE'
Updated Property
Updated Building Creation Date
Global Updated Building Creation Dates
next_content_type 'CityModel'
scope 'global'
propagate 'RIGHT_VALUE'

Figure 7.17.: A visualization of the pattern rules given in Listing 7.5 used to detect changes on the property *creationDate* of buildings. The relationship properties are shown in green, while the node properties are shown in blue. For clarity, the node property values *changeType* are omitted.

```
// Search for unchanged buildings
MATCH (b:Building:LeftDataset) // from old datasets
WHERE NOT exists((b)<-[*]-(:Change)) // non-deleted, non-split
  AND NOT exists((b)-[*]->()<-[]-(:Change)) // without any changes
RETURN b // '*' represents paths of arbitrary length from/to b
```

Listing 7.6: A Cypher query for detecting unchanged buildings.

By applying this query, only 1,402 or approximately 0.4 % of all 348,976 non-deleted, non-split buildings from the old datasets were detected to be unchanged. Figure 7.19 illustrates the spatial distribution of these unchanged buildings in Hamburg.

Among the three stakeholders introduced in Section 2.4, unchanged buildings could reveal information useful to the city mayor, as they might represent older structures (based on the available construction year) that could soon require repairs or renovations, or they could represent historical buildings that must be preserved. Additionally, unchanged buildings may also be of interest to data brokers, as they provide insights into how city objects are stored and represented within the city model.

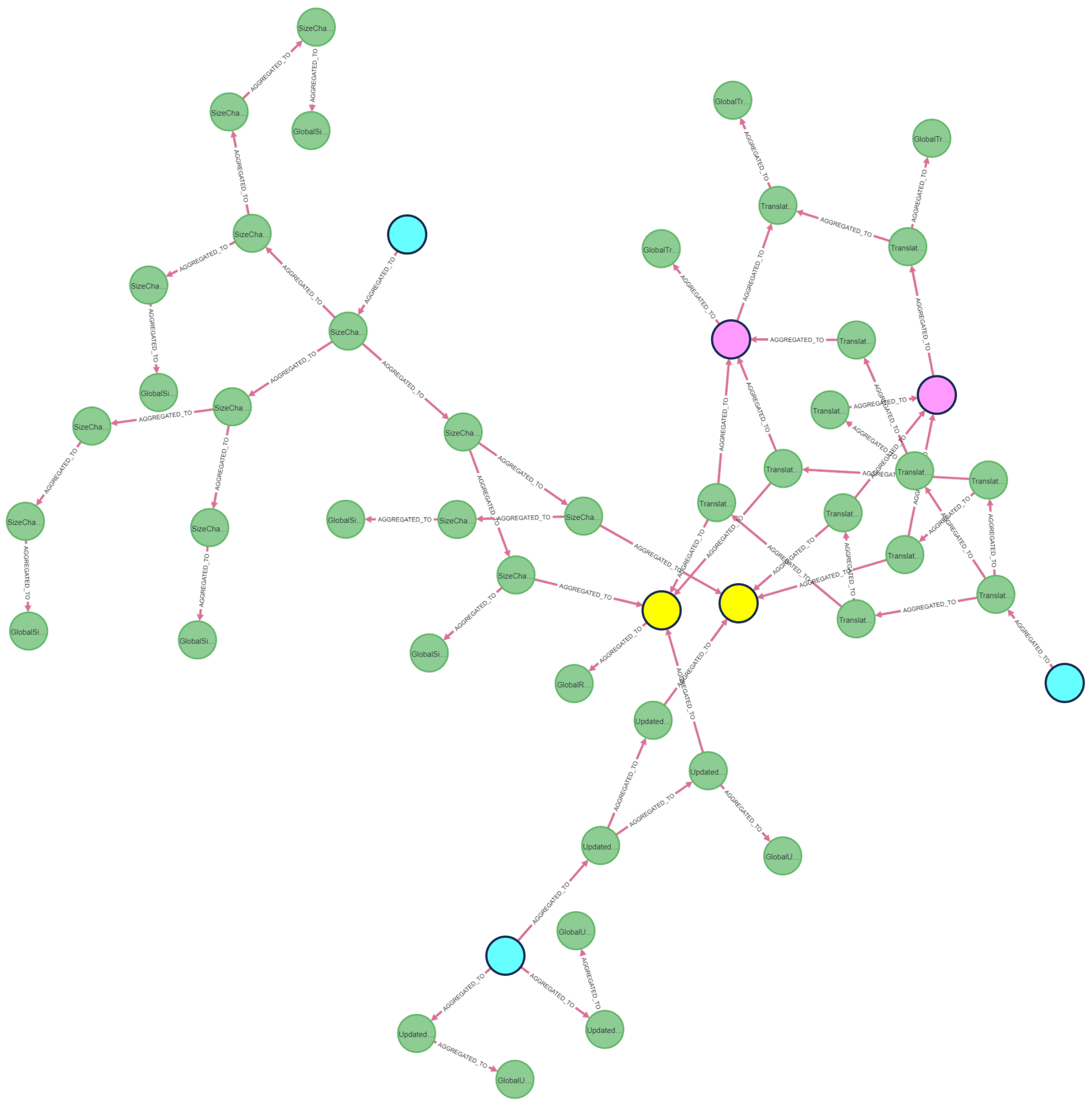


Figure 7.18.: A visualization of the entire rule network employed to identify change patterns in the Hamburg datasets. The yellow rule nodes represent the patterns associated with raised roofs in a building (left) or a building part (right). The purple rule nodes represent the patterns for a translated building (left) or building part (right). The cyan rule nodes denote the source and lowest-level rule nodes for size changes (left), updated properties (middle), and translations (right), upon which all rules are established. Visualized with Neo4j Browser. The full implementation of this rule network in Cypher is available in Listing B.1.

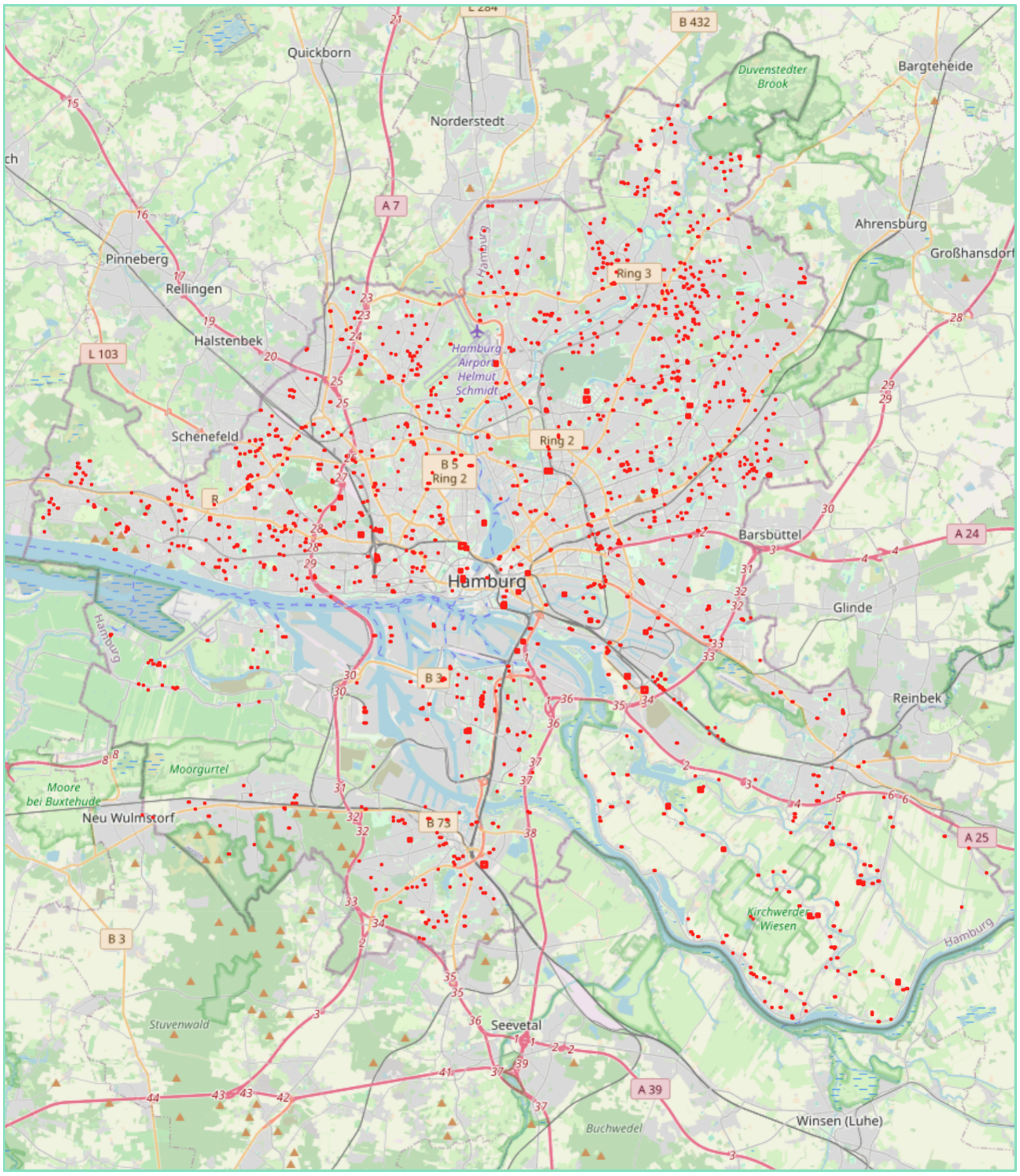


Figure 7.19.: The spatial distribution of unchanged buildings (red) in Hamburg. The majority of these buildings, both in terms of quantity and area, are located in the upper half of Hamburg, including the boroughs Altona, Wandsbek, and Hamburg-Mitte. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

For instance, between 2016 and 2022, no thematic or geometric changes were detected for the Congress Center Hamburg, as shown in Figure 7.20. Notably, the building underwent extensive renovations from 2017 until April of 2022 (CCH, 2022). However, the new Hamburg datasets were created in February 2022, prior to the reopening of the new Congress Center. As a result, despite the ongoing renovations, the building remained unchanged in the city model, since the changes could not be recorded during the time period. Therefore, in this context, an unchanged status may also indicate that the building is already undergoing renovations, which may not yet be complete or registered in the 3D city model.

The unchanged buildings identified in this case exhibit no changes, not even in their identifiers. Thus, the existence of such 1,402 unchanged buildings implies that no top-level change patterns can be detected on a global scale. Despite this, the analysis of the spatial and thematic clustering of these change patterns may still provide valuable insights.

7.5.3. Updated Identifiers of City Objects

Table 7.5 provides an overview of all buildings and building parts in the Hamburg datasets that have been detected with and without updated identifiers.

Table 7.5.: An overview of buildings and building parts detected with and without changed identifiers in the Hamburg datasets.

	Buildings		**Building Parts**	
Total number (non-deleted, non-split)	348,976	100.0 %	487,340	100.0 %
Detected with changed identifiers	347,574	99.6 %	459,218	94.2 %
Without changed identifiers	1,402	0.4 %	28,122	5.8 %

The vast majority of buildings (99.6 %) and building parts (94.2 %) have changed identifiers, with both old and new values starting with the prefix *DEHH*. Conversely, only 1,402 buildings and 28,122 building parts have retained their original identifiers.

Notably, this number 1,402 of buildings without updated identifiers aligns with the number of unchanged buildings (those without any changes attached to themselves or their sub-elements). This implies that any building in the employed Hamburg datasets that has retained its original identifiers have also remained completely unchanged between 2016 and 2022.

Similar to the unchanged buildings shown in Figure 7.19, all 28,122 building parts without updated identifiers are dispersed throughout the entire area of Hamburg.

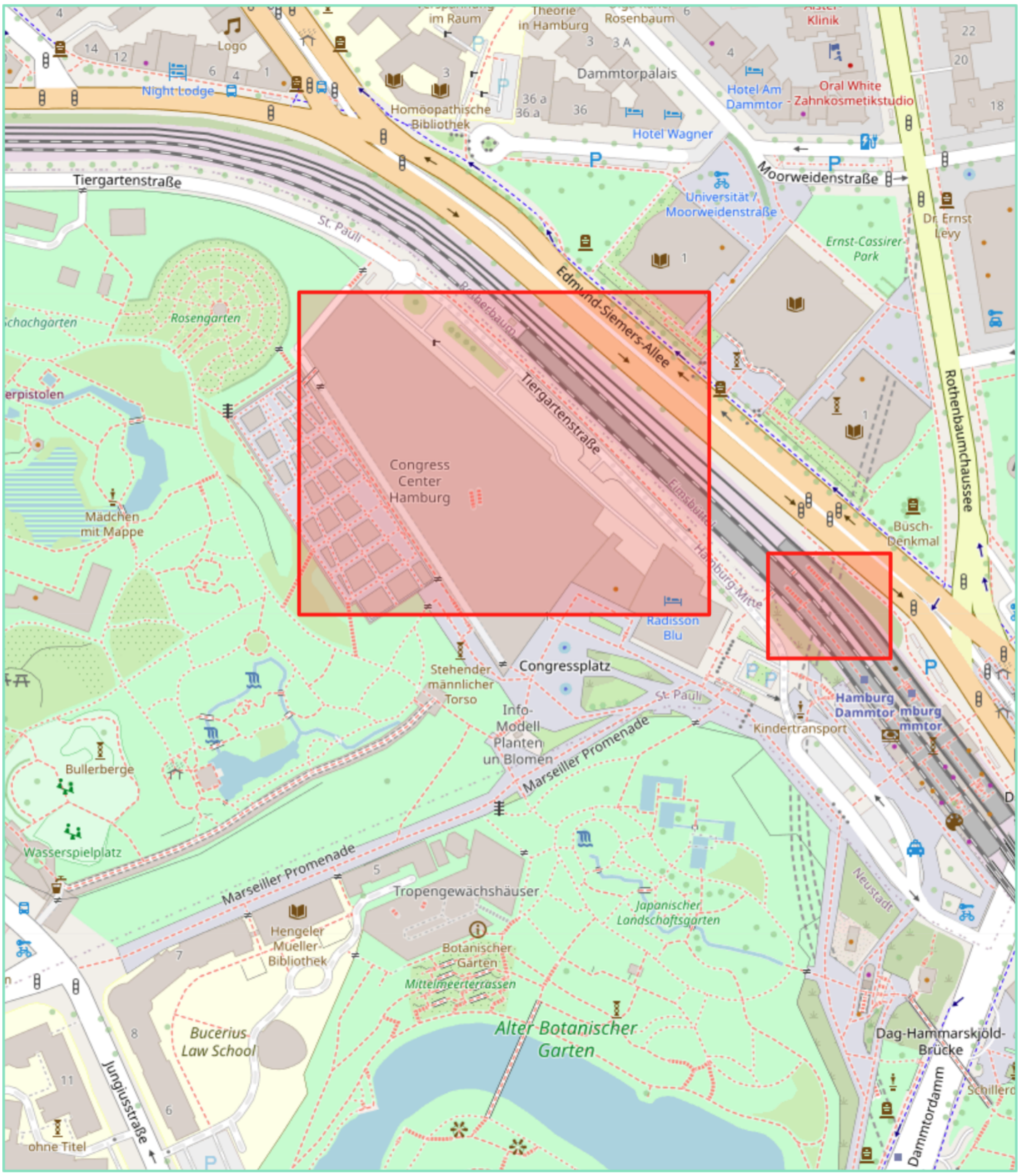


Figure 7.20.: A visualization of the Congress Center Hamburg (left), which was classified as unchanged between 2016 and 2022. However, the building underwent extensive renovations between 2017 and 2022 (CCH, 2022), shortly after the old and new datasets were created. Consequently, the renovations of the facility were not recorded in the datasets, resulting in the unchanged status. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

7.5.4. Updated Property Values *creationDate*

In CityGML, the property *creationDate* of a feature, such as a building or a building part, is often updated to indicate the time of the newest modifications made to that object. This may occur to individual objects, objects within a specific region, or all objects of the same type in the entire city model. Since no global change patterns could be observed for the Hamburg datasets due to the existence of unchanged buildings, as explained previously in Section 7.5.2, the objective is to further investigate the patterns hidden behind updated *creationDate* values within clusters of buildings and building parts.

Given that the pattern rules for the property values *creationDate*, as visualized previously in Figure 7.17, also include a directive *calc_scope* to calculate the thematic and spatial scope of buildings and building parts over *creationDate*, the interpretation process groups buildings and building parts by their common updated *creationDate* values.

As a result, 14 different clusters of buildings and building parts have been identified in total, with each represented by a distinct value *creationDate*. These groups can be derived by retrieving the scope nodes attached to the city model node after the interpretation process is complete, as visualized in Figure 7.21. Each scope node contains information about the updated *creationDate* value, the number of buildings and building parts that share that new value, the bounding box containing all these buildings and building parts, as well as additional metrics such as the thematic and spatial coverage of the cluster over the entire city model. Furthermore, Table 7.6 presents an overview of these 14 groups of buildings and building parts grouped by their updated *creationDate* values.

On average, each group consists of approximately 30 thousand buildings and 25 thousand building parts. The total number of buildings and building parts from these *creationDate* groups, as shown in Table 7.6, coincide with the total number of buildings and building parts with updated identifiers, as presented in Table 7.5. This suggests that the property *creationDate* of all buildings and building parts is updated whenever any modification, even as small as an individual updated identifier, is made to the feature objects and their sub-elements.

As a result, these 14 *creationDate* groups can be thought of as 'update batches' of the entire Hamburg datasets, performed gradually over 14 days by a data manager or a GIS specialist. In each batch, the content and structure of all associated buildings and building parts are modified. Thus, the patterns observed in these groups are of significant importance to stakeholders such as data brokers and city model managers, as they allow for backtracking and capturing the history of modifications made to the city models over time.

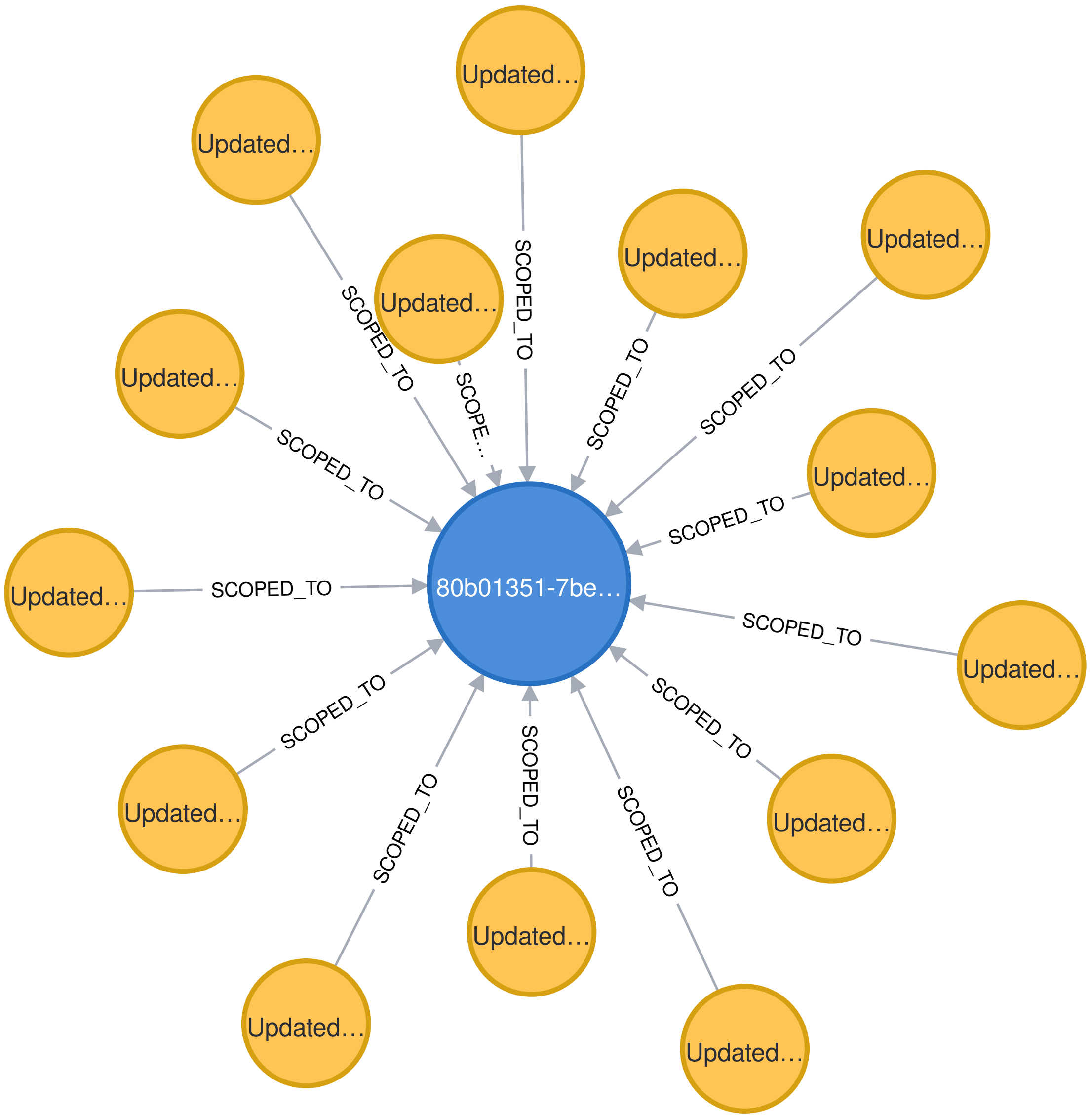


Figure 7.21.: Visualization of scope nodes (orange) representing all 14 clusters of buildings. In each cluster, all buildings share the same updated *creationDate* value. The information stored in each scope node includes the number of buildings that share the same updated property value, the cumulative bounding box containing all these buildings, as well as additional spatial indicators such as the spatial and type coverage of the current cluster over the entire city model. These scope nodes are attached to the city model node (blue). Visualized with Neo4j Browser.

Table 7.6.: An overview of buildings and building parts detected with 14 different updated values *creationDate* in the Hamburg datasets.

New *creationDate*	**Buildings**		**Building Parts**	
2022-01-25	11,155	2.43 %	8,445	2.43 %
2022-01-26	44,085	9.6 %	36,141	10.4 %
2022-01-27	48,379	10.53 %	34,148	9.82 %
2022-01-28	47,230	10.28 %	34,069	9.80 %
2022-01-29	54,423	11.85 %	38,209	10.99 %
2022-01-30	51,375	11.19 %	35,054	10.09 %
2022-01-31	3,723	0.81 %	4,074	1.17 %
2022-02-05	35,378	7.70 %	26,580	7.65 %
2022-02-06	39,482	8.60 %	29,120	8.38 %
2022-02-07	32,744	7.13 %	24,160	6.95 %
2022-02-08	39,079	8.51 %	30,713	8.84 %
2022-02-09	34,842	7.59 %	31,817	9.15 %
2022-02-10	17,248	3.76 %	14,953	4.30 %
2022-02-21	75	0.02 %	91	0.03 %
Total	459,218	100.00 %	347,574	100.00 %

Figure 7.22 provides a visualization of the modified buildings grouped by the first seven *creationDate* values, spanning from *2022-01-25* to *2022-01-31*. Similarly, Figure 7.23 presents a visualization of the changed buildings associated with the remaining seven dates, from *2022-02-05* to *2022-02-10* and *2022-02-21*. In these figures, each *creationDate* group is displayed as a vertical column on the map of Hamburg, arranged in ascending temporal order from left to right. To distinguish between two adjacent groups, their respective buildings are shown in two different alternating colours: red and orange. Figure 7.24 offers a close-up visualization of a boundary area between two of such adjacent groups.

Figures 7.22 to 7.24 show that the entire city model of Hamburg was modified during the period between January 25, 2022 until February 21, 2022. During this timeframe, the city model was updated in sequential segments, starting from the west boundary of the upper half of Hamburg and progressing in vertical zones from left to right towards the east boundary of the lower half of Hamburg. The update process then ended with the modifications of buildings on the island of Neuwerk, located northwest of the city of Hamburg.

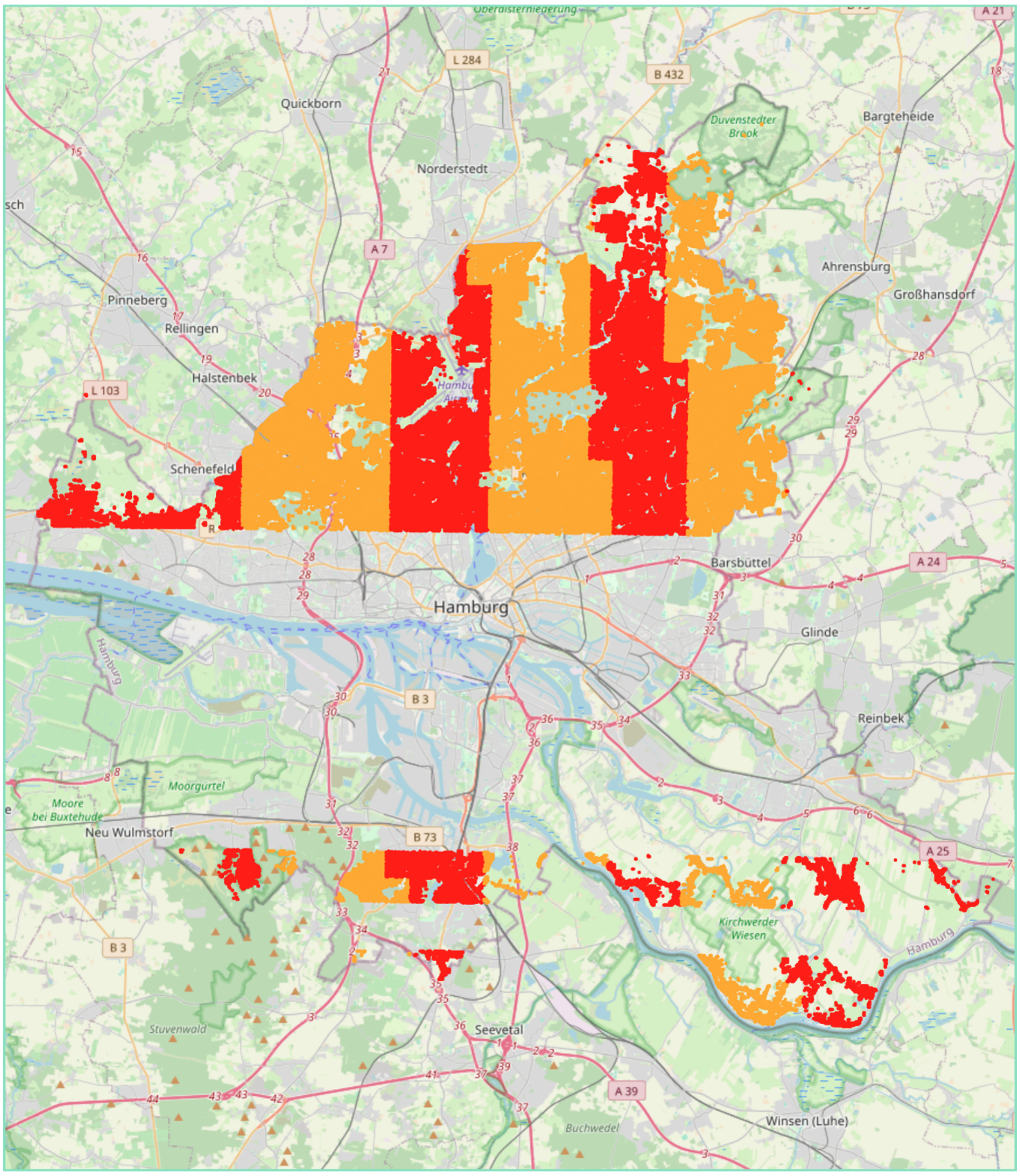


Figure 7.22.: An illustration of the modified buildings of Hamburg grouped by their respective *creationDate* values (first seven dates). Each group is represented as a vertical column, arranged in ascending temporal order from left to right. The groups are distinguished by colours. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

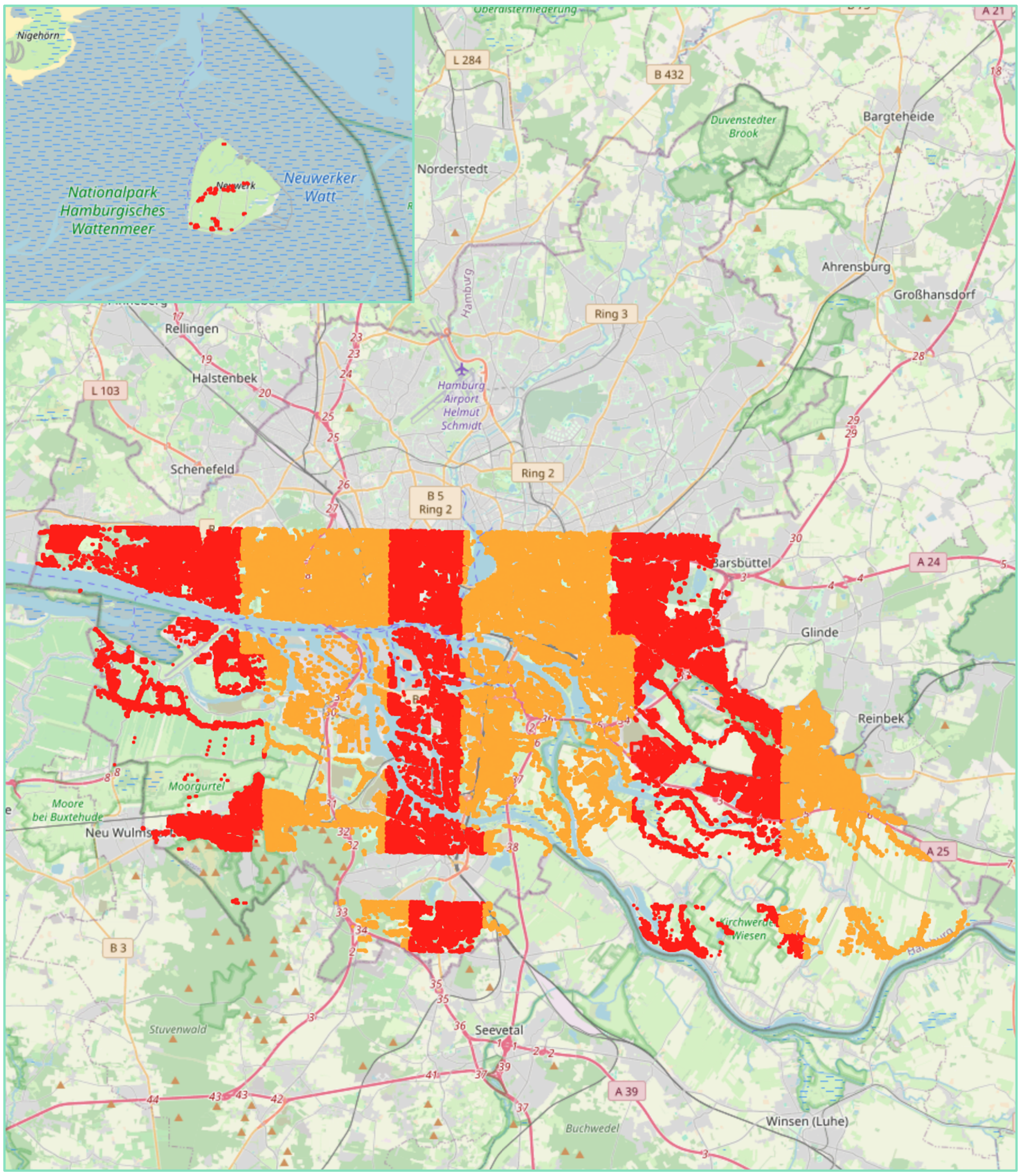


Figure 7.23.: An illustration of the modified buildings of Hamburg grouped by their respective *creationDate* values (last seven dates). Each group is represented as a vertical column, arranged in ascending temporal order from left to right. The groups are distinguished by colours. Changes on the last date were made on the island of Neuwerk (top-left). Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

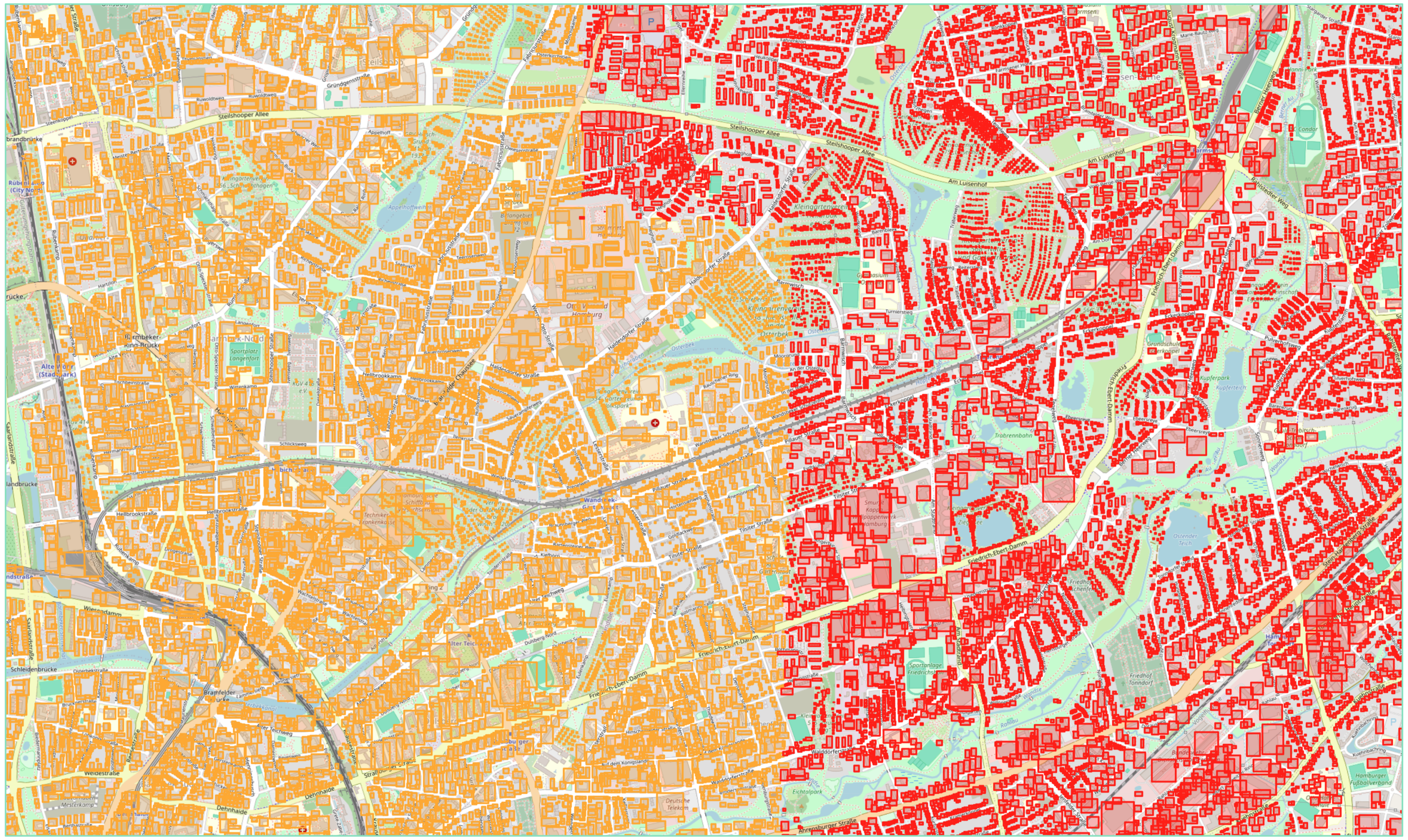


Figure 7.24.: A detailed visualization of the boundary area between two adjacent groups of modified buildings (red and orange) distinguished by their associated *creationDate* values. Although each group generally occupies a vertical region on the map, their boundaries do not strictly adhere to a straight line. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

7.5.5. Updated Coded Property Values

The Hamburg datasets incorporated coded values for feature properties such as *function* and *roofType*. These values originate from comprehensive code lists provided by the Working Committee of the Surveying Authorities of the Laender of the Federal Republic of Germany (AdV) (AdV, 2022). These lists cover a wide range of object types and their thematic properties. For demonstration purposes, a subset containing some values relevant to this use case is utilized, including those associated with residential, commercial, and industrial use, as outlined in Table 7.7.

Based on these values, the function and usage of a building can be determined, thereby allowing for the analysis of the dynamics between the city's available space for residential, commercial, and industrial purposes. For instance, Table 7.8 presents a matrix for interpreting the changes in these function values of buildings, providing insights into the repurposing of buildings and its impact on the city.

Thus, rules can be defined for identifying change patterns in residential, commercial, or industrial space, as outlined in Listing 7.7.

```
// Pattern for conversion of residential to commercial space
MERGE (updated_property:RULE {
  change_type: 'UpdatedProperty'
})-[:AGGREGATED_TO {
  next_content_type: 'Code',
  search_length: 0,
  conditions: 'NAME === "value" ' +
    '&& LEFT_VALUE >= 1000 && LEFT_VALUE <= 1025' + // residential
    '&& RIGHT_VALUE >= 2000 && RIGHT_VALUE < 2100', // commercial
  weight: 1
}]->(residential_to_commercial_code:RULE {
  change_type: 'ResidentialToCommercialCode'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart', // only in buildings
  weight: 1
}]->(residential_to_commercial_building:RULE {
  change_type: 'ResidentialToCommercialBuilding'
})
```

Listing 7.7: Pattern rules in Cypher to detect conversion of available residential to commercial space. This can be extended for other values.

Table 7.7.: Examples of relevant coded values for the property *function* of buildings and building parts. These values are divided into three categories according to their residential, commercial, and industrial use. Excerpt from (AdV, 2022).

(a) Code 10xx: Residential Use

Code	Description
1000	Residential building
1010	Residential house
1020	Residential home
1021	Children's home
1022	Senior citizens' home
1023	Nurses' home
1024	Student dormitory
1025	School hostel

(b) Code 21xx: Industrial Use

Code	Description
2100	Trade-industry facility
2110	Production building
2111	Factory
2112	Operational building
2113	Brewery
2114	Distillery
2120	Workshop
2121	Sawmill
2130	Gas station
2131	Car wash
2140	Storage facility
2141	Cold storage
2142	Storage building
2143	Warehouse
2150	Logistics building
2160	Research facility
2170	Extraction facility
2171	Mine
2172	Salt mine
2180	Company social hub

(c) Code 20xx: Commercial Use

Code	Description
2000	Commercial building
2010	Trade and service building
2020	Office building
2030	Credit institution
2040	Insurance company
2050	Business building
2051	Department store
2052	Shopping centre
2053	Market hall
2054	Shop
2055	Kiosk
2056	Pharmacy
2060	Exhibition hall
2070	Lodging building
2071	Hotel, motel, guest-house
2072	Youth hostel
2073	Cabin (with lodging)
2074	Campsite building
2080	Buildings for catering
2081	Restaurant, diner
2082	Cabin (without lodging)
2083	Canteen
2090	Leisure venue
2091	Banquet hall
2092	Cinema
2093	Bowling alley
2094	Casino
2095	Arcade

Table 7.8.: An example matrix for interpreting changes in buildings' *function* values and their impact on the city's available space for living, commerce, and industry. The rows represent old values, while the columns represent new ones.

	10xx	**20xx**	**21xx**
10xx		↑ Business space ↓ Living space	↑ Industry space ↓ Living space
20xx	↑ Living space ↓ Business space		↑ Industry space ↓ Business space
21xx	↑ Living space ↓ Industry space	↑ Business space ↓ Industry space	

Out of the 140,676 detected changes in the function values of buildings in the Hamburg datasets, 26,320 (18.7 %) are updates exclusively within the selected code values 10xx, 20xx, and 21xx. When grouping all 10xx codes as residential, 20xx codes as commercial, and 21xx codes as industrial, the majority (96.4 %) of these changes are updates within the same group. For example, there are 13,885 changes from code 1000 (residential building) to 1010 (residential house), and 5,193 changes from 2020 (office building) to 2010 (trade and service building). These changes do not affect the overall number of buildings per residential, commercial, and industrial use. Therefore, the remaining 945 changes that actually alternate between the three given building function groups are further analysed.

Figure 7.25 provides an overview of the dynamics of the repurposing of buildings between the three selected groups. The net increase or decrease in the number of buildings for each group is determined by subtracting the number of buildings converted from this group to others from the number of buildings this group gained from conversions of buildings from other groups. For instance, 349 and 165 of residential buildings were repurposed as industrial and commercial buildings, respectively, while 36 and 98 buildings were gained from the industrial and commercial group, respectively. As a result, the residential group has a net loss of 380 buildings. On the other hand, both the industrial and commercial group have a net gain of 254 and 126 buildings, respectively.

Such interpretations are valuable to various stakeholders, such as data brokers, but most importantly the city mayor, as it provides crucial data for informed decision making and policy strategies, thereby allowing for more effective planning in the city of Hamburg.

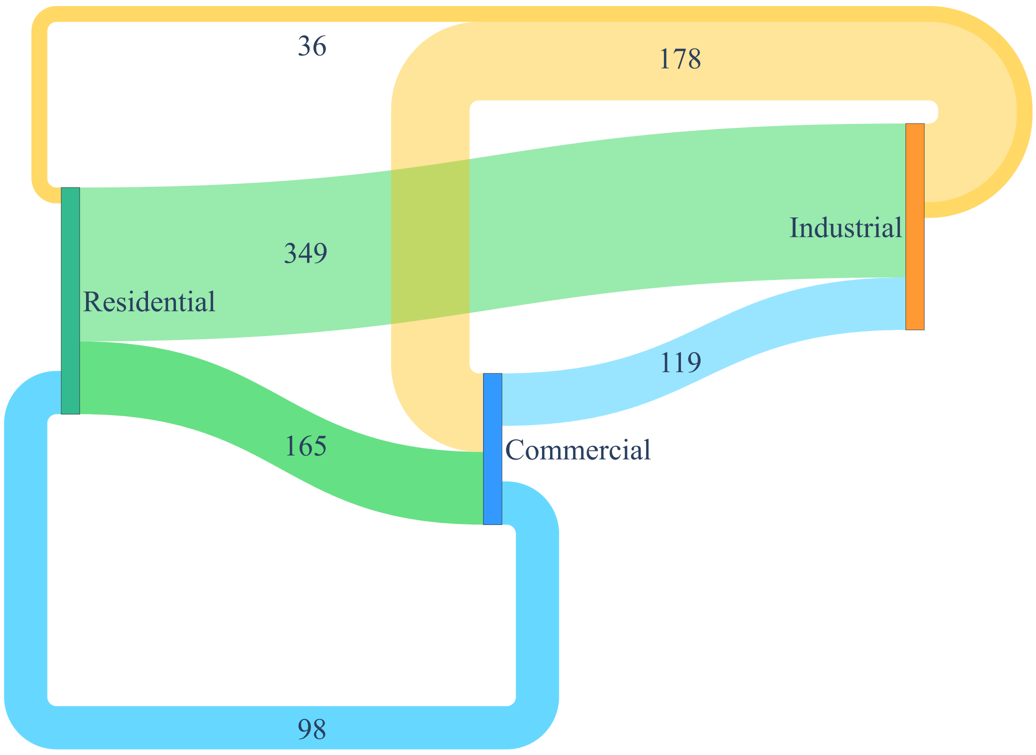


Figure 7.25.: A visualization of the gains and losses of buildings per category for residential (green), commercial (blue), and industrial use (orange). Each connection starts from the right side of one group and ends on the left side of another, representing a conversion of buildings from the initial group to the target group. The number of buildings repurposed is indicated by the width of each connection. This illustration only considers changes in building function values where both the original and updated values fall within the selected three categories.

The calculations presented above are based solely on the number of repurposed buildings, without further considering their actual space allocated for residential, commercial, and industrial use. Demonstrations of how exact fluctuations in such residential, commercial, or industrial space can be computed are provided in Section 7.6. Moreover, the interpretations illustrated in this experiment are derived from the modified *function* values of buildings, but the same can also be applied to the updated *roofType* values and some generic attributes of buildings and building parts that are also encoded using predefined code lists (AdV, 2022).

7.5.6. Raised Roofs of Buildings and Building Parts

By applying the rules given in Section 5.3 and illustrated in Figures 5.5 and 5.6, the interpretation process searches for patterns of elevated roofs among all geometric translations and size changes in buildings and building parts of the entire Hamburg datasets. A building or a building part is considered to have raised roofs if the following conditions are met:

1. All its boundary roof surfaces have been vertically elevated without any change in their size or shape,
2. All its boundary ground surfaces have been moved upwards, downwards, or remained stationary,
3. The heights of all boundary wall surfaces have been adjusted to accommodate the translations of the roof and ground surfaces, and
4. The measured height of that building or building part has been updated accordingly to reflect the new height.

The Cypher query employed to describe this pattern rule among buildings (without building parts) can be found in Listing 7.8. Due to limited space, the definitions for the preceding rule nodes corresponding to the aforementioned conditions are omitted. They include:

1. Rule node *translated_building_roofs_no_bparts* for moved roof surfaces of a building that contains no building parts,
2. Rule node *translated_building_grounds_no_bparts* for moved ground surfaces of a building that contains no building parts,
3. Rule node *resized_building_walls_no_bparts* for resized wall surfaces of a building that contains no building parts, and
4. Rule node *updated_building_measured_height* for updated measured height of a building.

Furthermore, the function *approxEquals(a, b)*, which appears throughout the query, is used to compare two numeric values *a* and *b* with a predetermined error tolerance taken into account. Unlike other internal functions employed during the mapping, matching, and interpretation process, this function can be defined by users and stored separately in a JavaScript file, which can then be parsed by the script engine within the interpreter in runtime. This allows for more flexibility, modularity, and extendability when defining complex rule conditions. The location of this script file in the current implementation can be found in Figure 7.1.

```
// Pattern for raised roofs of buildings (without building parts)
MERGE (updated_building_measured_height)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_height',
  conditions: 'RIGHT_VALUE - LEFT_VALUE > 0',
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: '1'
}]->(raised_building_roofs:RULE { // create once, reuse later
  change_type: 'RaisedBuildingRoofs',
  join: 'approxEquals(rule_resized_walls.z,
            rule_height.RIGHT_VALUE - rule_height.LEFT_VALUE)
      && approxEquals(rule_translated_roofs.z,
            rule_resized_walls.z + rule_translated_grounds.z)'
})
MERGE (translated_building_roofs_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_roofs',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs) // reference existing rule node
MERGE (translated_building_grounds_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_grounds',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs) // reference existing rule node
MERGE (resized_building_walls_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_resized_walls',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs) // reference existing rule node
```

Listing 7.8: Pattern rules in Cypher to detect raised building roofs. The same can be applied to detect raised roofs in building parts.

The *join* conditions in the rule node *raised_building_roofs* specify the necessary criteria that all preceding rule nodes must satisfy to allow for the creation of the new interpreted change of raised building roofs. These conditions are evaluated based on a temporary context memory, which collects all information about the contents of the preceding rule nodes, as dictated by the property *propagate* utilized in each rule relationship. To differentiate attributes from different rule nodes that have the same name within this shared context, each rule node and all their attributes can be assigned with a unique name, as determined by the property *name* in each rule relationship shown in Listing 7.8. For instance, the name *rule_height* is assigned to the preceding rule node *updated_building_measured_height* and can be utilized as *rule_height.LEFT_VALUE* to retrieve the old measured height value of the building before its update.

Once the interpretation process is complete and all pattern rules have been applied, simple Cypher queries can be performed to retrieve the interpretation nodes created for raised building roofs, thereby allowing for more complex analyses based on available data linked to these nodes. For instance, Listing 7.9 presents an example query that provides an overview of all detected raised building roofs, including the minimum and maximum raise value, as well as the number of detected interpretation nodes.

```
// Retrieve an overview of identified raised building roofs
MATCH (c:Change {change_type: "RaisedBuildingRoofs"})
WITH n.RIGHT_VALUE - n.LEFT_VALUE AS dh // updated measured height
RETURN min(dh), max(dh), count(dh) // minimum, maximum, and quantity
```

Listing 7.9: An example Cypher query to provide an overview of all detected raised building roofs. While the information itself is challenging to derive without any pre-processing, the query remains simple and manageable due to the available interpretation nodes created for raised building roofs.

Figure 7.26 shows a visualization in Neo4j Browser of a building with raised roofs, as defined by the pattern rules given in Figure 5.6. A total number of 50,390 buildings (without building parts), which account for approximately 14 % of all non-deleted non-split buildings, have been detected with elevated or lowered roofs. For building parts, a total number of 148,505 instances, or approximately 31 % of all non-deleted building parts, have been identified with raised roofs. As mentioned in Section 4.5.4, a maximum threshold of 3 m for translations is employed to reduce the number of false matches among parallel surfaces. If this threshold is exceedingly large, parallel wall surfaces of a building would be matched to each other, even though they are different geometries. Thus, the height increases and decreases of all roof surfaces range from -3 m to 3 m, as illustrated in Figure 7.27. Roofs shifted by more than 3 m can be detected in buildings with footprints matched within the translation threshold.

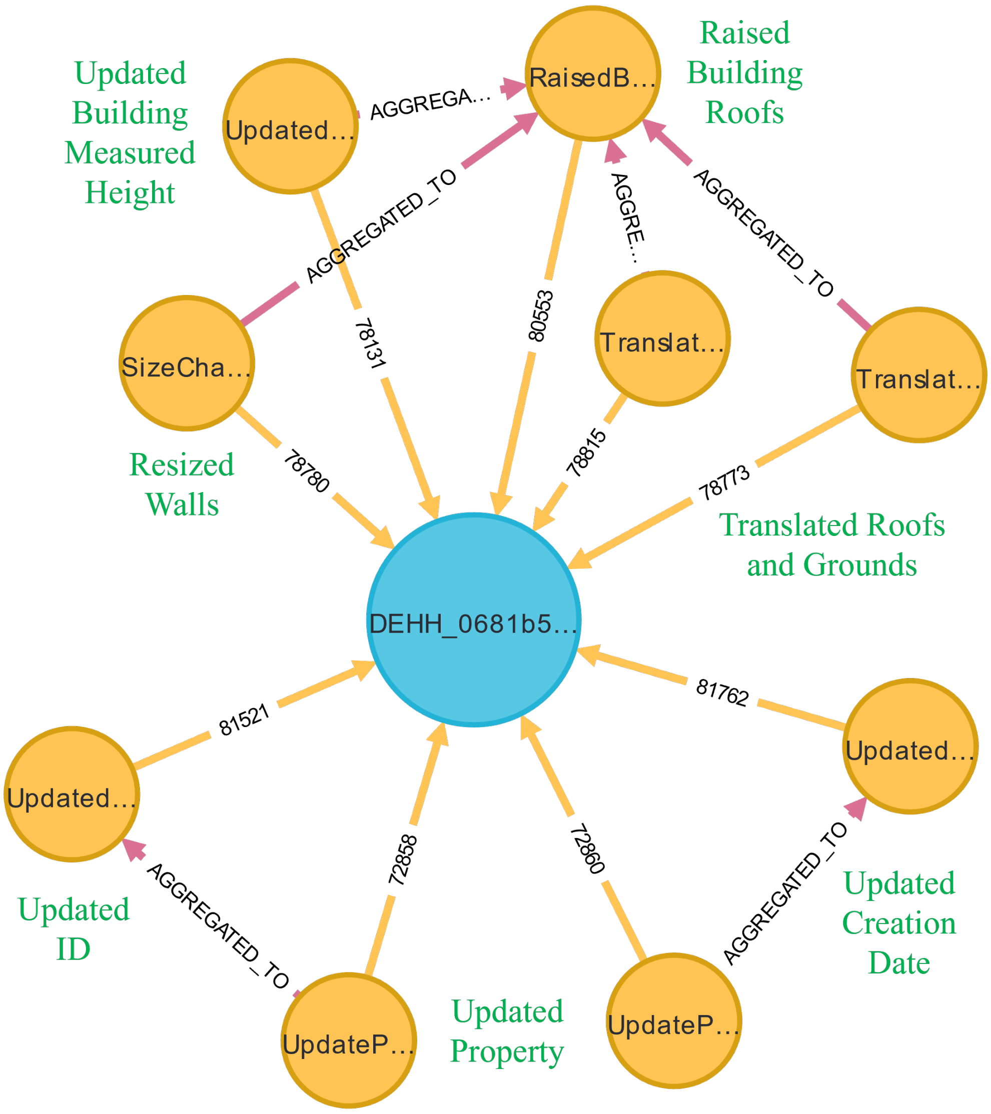
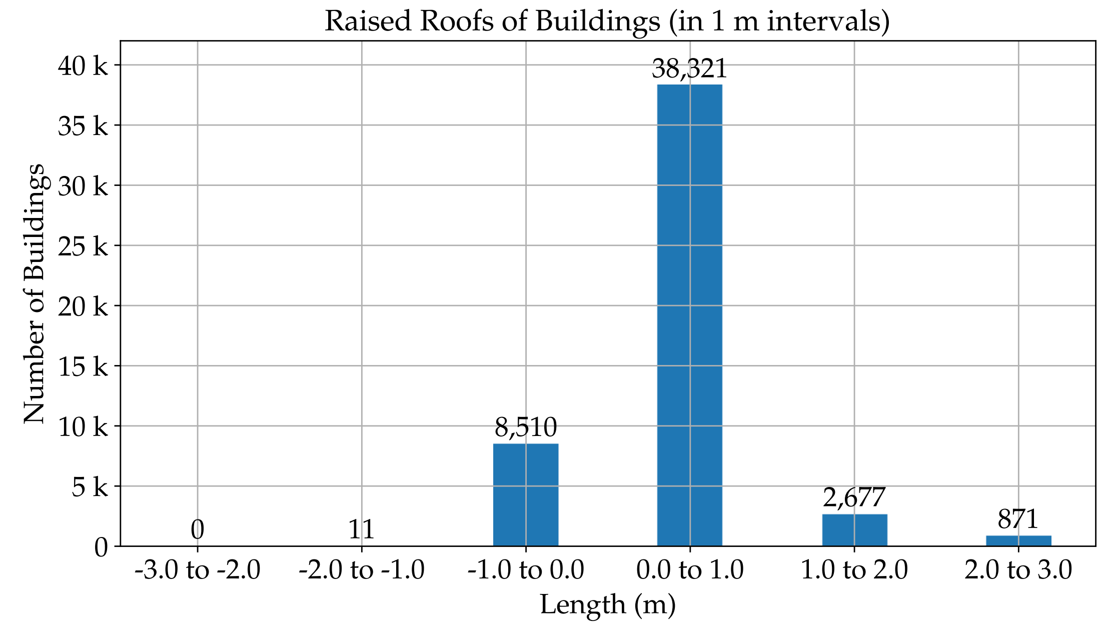


Figure 7.26.: A visualization in Neo4j Browser of a building (blue) with raised roofs that satisfy the pattern rules from Figure 5.6. The four lower interpretation nodes indicate changes in the identifier and *creationDate*. The upper five interpretation nodes represent geometric change patterns that involve translated roof and ground surfaces, resized wall surfaces, as well as the increase in the building's measured height. The topmost interpretation node signifies a detected pattern of raised roofs. These interpretation nodes are attached to their corresponding building node (orange connections), allowing for efficient retrieval of interpreted data.



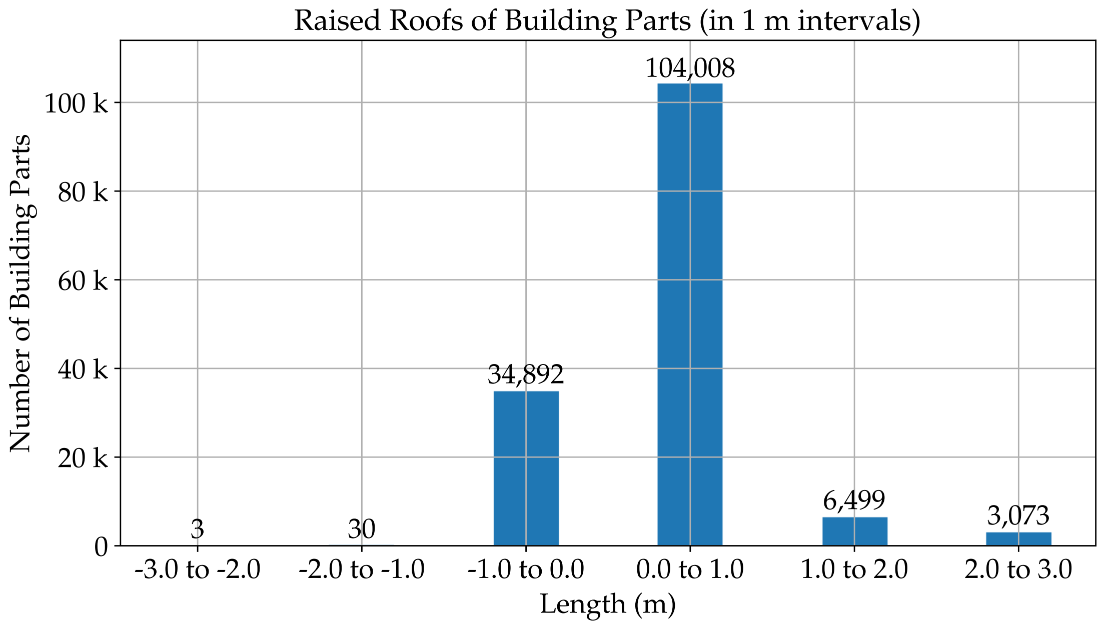


Figure 7.27.: The distribution of detected raised roofs in buildings and building parts in 1 m intervals based on the pattern rules given in Listing 7.8. The maximum translation length employed by the implementation is 3 m.

The movement of the roofs of a building may be influenced by the translation of its ground surfaces. For instance, a roof raise of 2 m may be the result of an elevated ground surface by that same amount, leading to no changes in the living space of that building. Thus, to investigate the actual changes in height of a building, the vertical size changes in wall surfaces are also considered, which correspond to the differences between the (signed) roof and ground translations. In the majority of cases, these resize values of walls also align with the changes in the buildings' measured heights, represented by the property *measuredHeight*.

Figure 7.28 visualizes the distribution of such changes in wall heights. Their values range from 0.001 m to 4.782 m for buildings (without building parts) and from 0.001 m to 4.679 m for building parts. The current implementation allows an error tolerance of 0.001 m for lengths and translations. Any length values smaller than this threshold are considered equal to length zero.

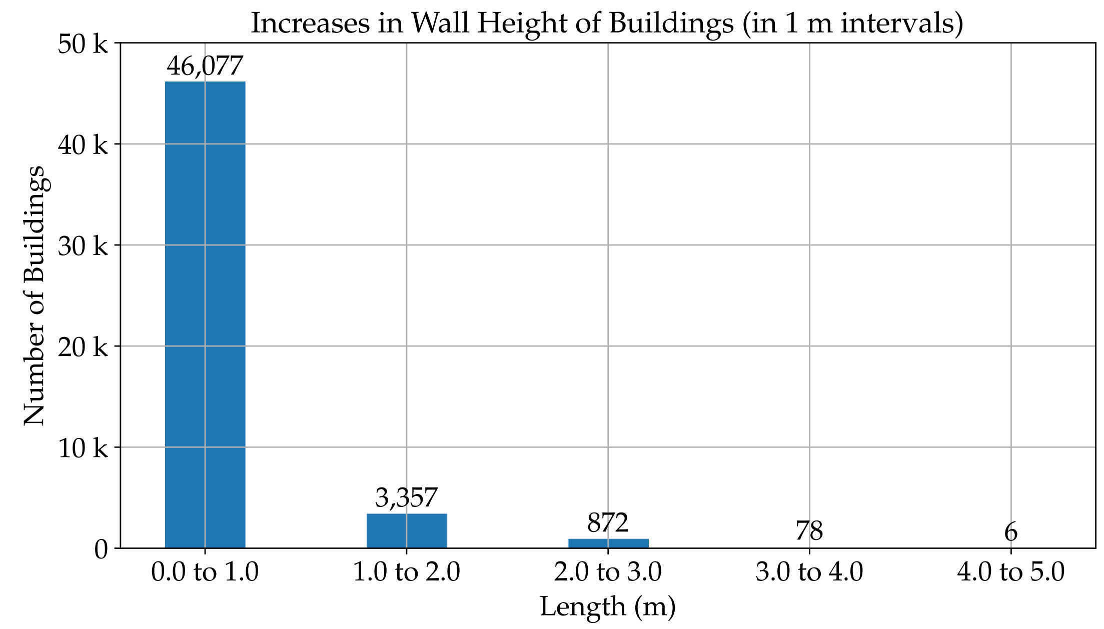
The majority of these height offsets (91 % for buildings and 88 % for building parts) fall below 1 m, with the number of these changes decreases as the wall height offset increases. At the maximum height difference, 6 buildings and 37 building parts were detected with wall heights increased by over 4 m.

No buildings or building parts were found with decreased wall heights, even though more than 8,500 buildings have been observed with lowered roofs. This means that the ground surfaces of these buildings must also have been shifted downwards with a greater offset, resulting in an overall increase in building heights. Such changes can be interpreted as a correction in the elevation of the roof and ground surfaces of building.

These elevated roofs and increased wall heights may indicate newly added storeys, resulting in increased living space. This is illustrated in the example use case of the building **Grüner Bunker** (translated as the Green Bunker) in Hamburg. The building was constructed as a flak bunker in 1942. After the Second World War, it became a historical building and is under monumental protection. Construction work for five new pyramid-shaped storeys on top of the building began in 2019, and by the end of 2023, trees had been planted on the added levels as one of the first steps to convert the building into a city garden above the rooftops of Hamburg (EHP, 2023).

A 3D visualization of this building from the Hamburg datasets 2016 and 2022 is shown in Figure 7.29. Satellite images of the building before and during the construction work in 2019 and 2021, respectively, are displayed in Figure 7.30.

In the Hamburg datasets, the building Grüner Bunker is represented as a collection of ten building parts, as visualized in Figure 7.31. Each yellow rectangle represents the 2D bounding box of a building part. The measurements associated with each bounding box depict the consistent height increase in metres of all walls within the corresponding building part. These increases range from 3.8 m to 12.5 m. In addition, ground surfaces of all building parts have been moved downwards by the same offset of 1.5 m.



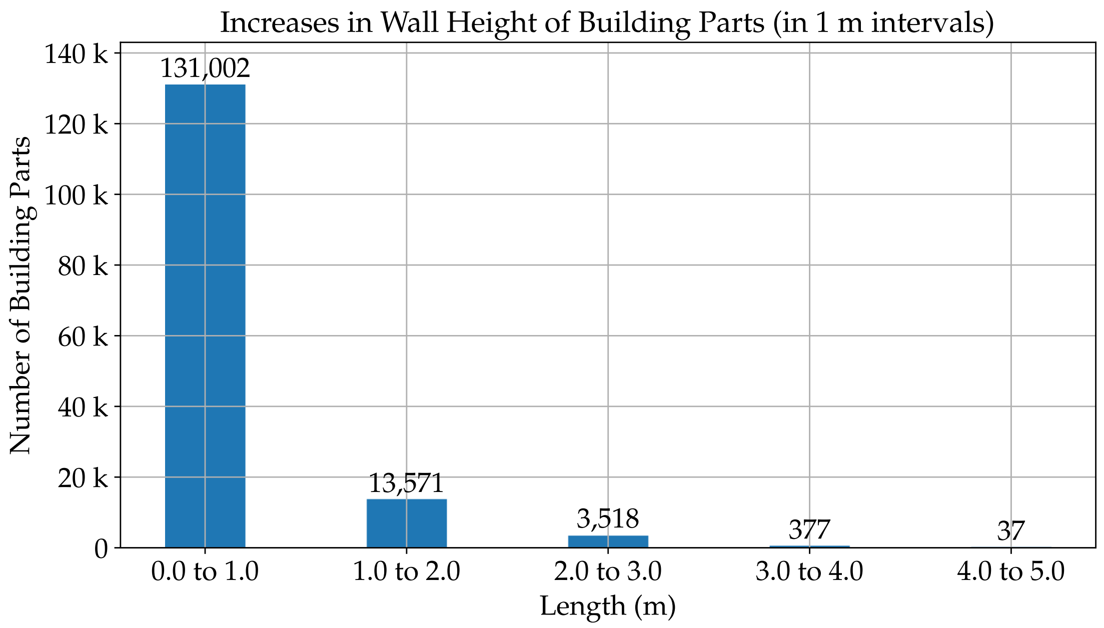
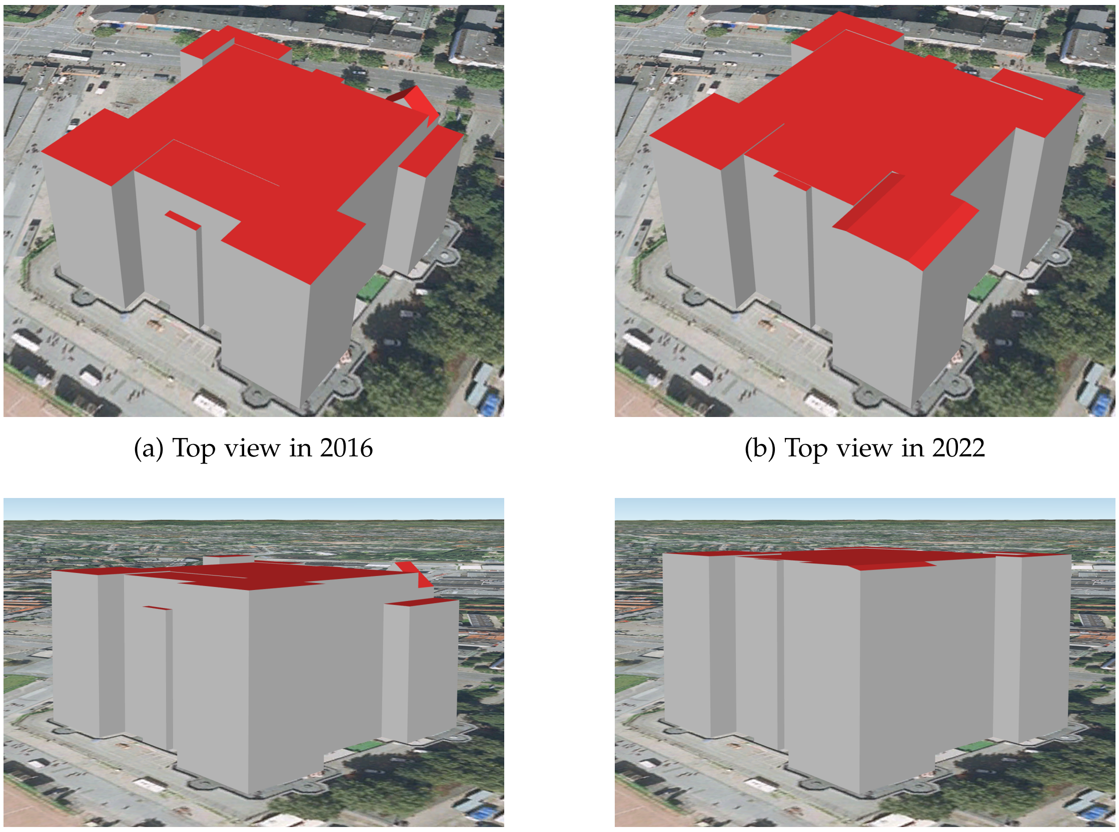


Figure 7.28.: The distribution of the increases in wall heights of buildings and building parts in 1 m intervals, computed from (signed) translations of roof and ground surfaces.



(a) Top view in 2016

(b) Top view in 2022

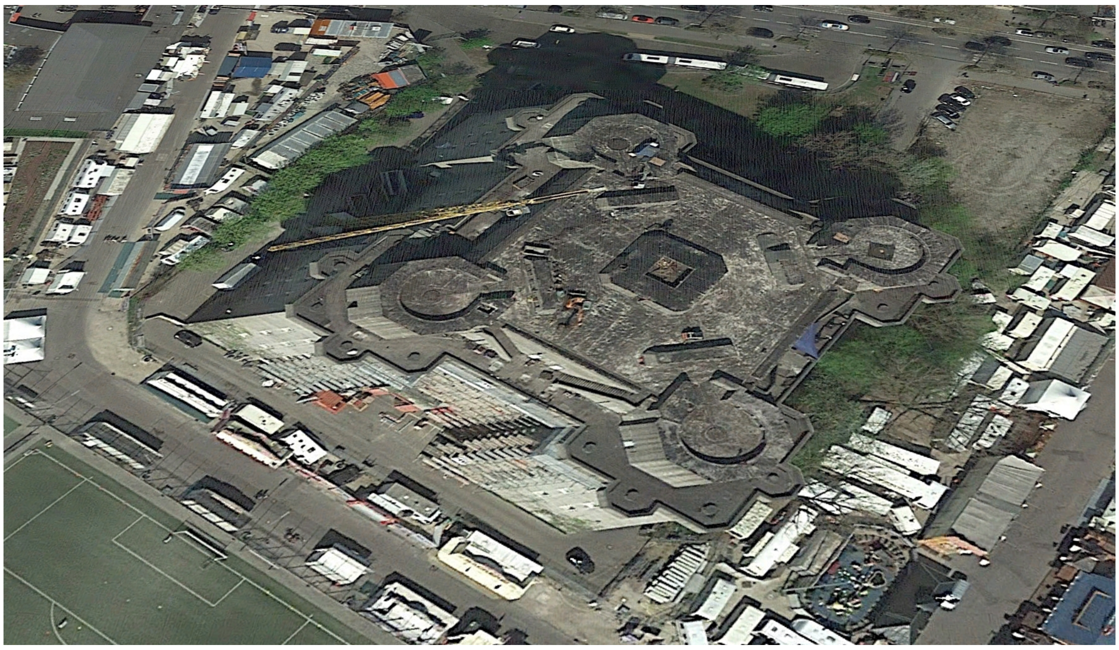
(c) Side view in 2016

(d) Side view in 2022

Figure 7.29.: The 3D models of the building Grüner Bunker in 2016 (left) and 2022 (right), displayed in top view (top) and side view (bottom). The visualization shows an overall increase in the building's height. Visualized in Google Earth Pro using the 3D building models exported from the 3DCityDB.

The increases in roof elevations and wall heights discussed above can be extracted and analysed using the detected thematic and geometric changes and their interpretations attached directly to the graph representations of each building part. Figure 7.32 provides an illustration of such nodes in the graph database Neo4j.

In the real world, a sufficiently large elevation in the roofs of a building or a building part often coincides with a change in the number of its storeys. This correlation in the datasets can be evaluated by searching for buildings and building parts that are attached with both a change in the roof elevation and a change in the number of storeys. The results are illustrated in Table 7.9. A total number of 80 such buildings and 511


(a) Before construction work in 2019

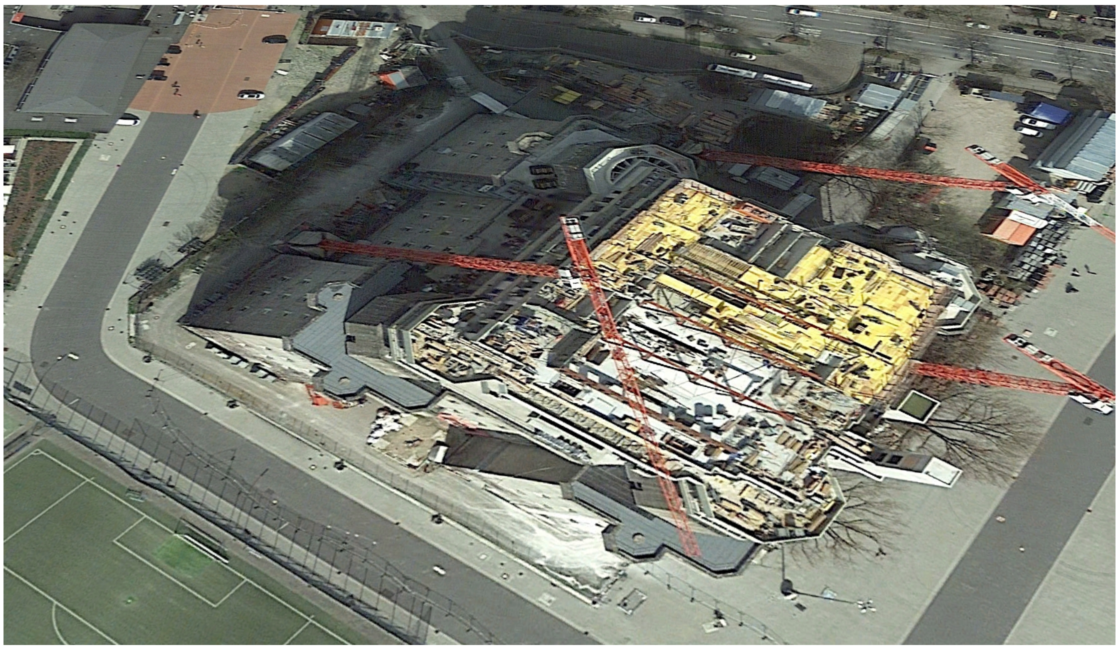

(b) During construction work in 2021

Figure 7.30.: Satellite images of the building Grüner Bunker before and during construction work for new storeys in 2019 and 2021, respectively. Visualized with historical data from 2019 and 2021 in Google Earth Pro.

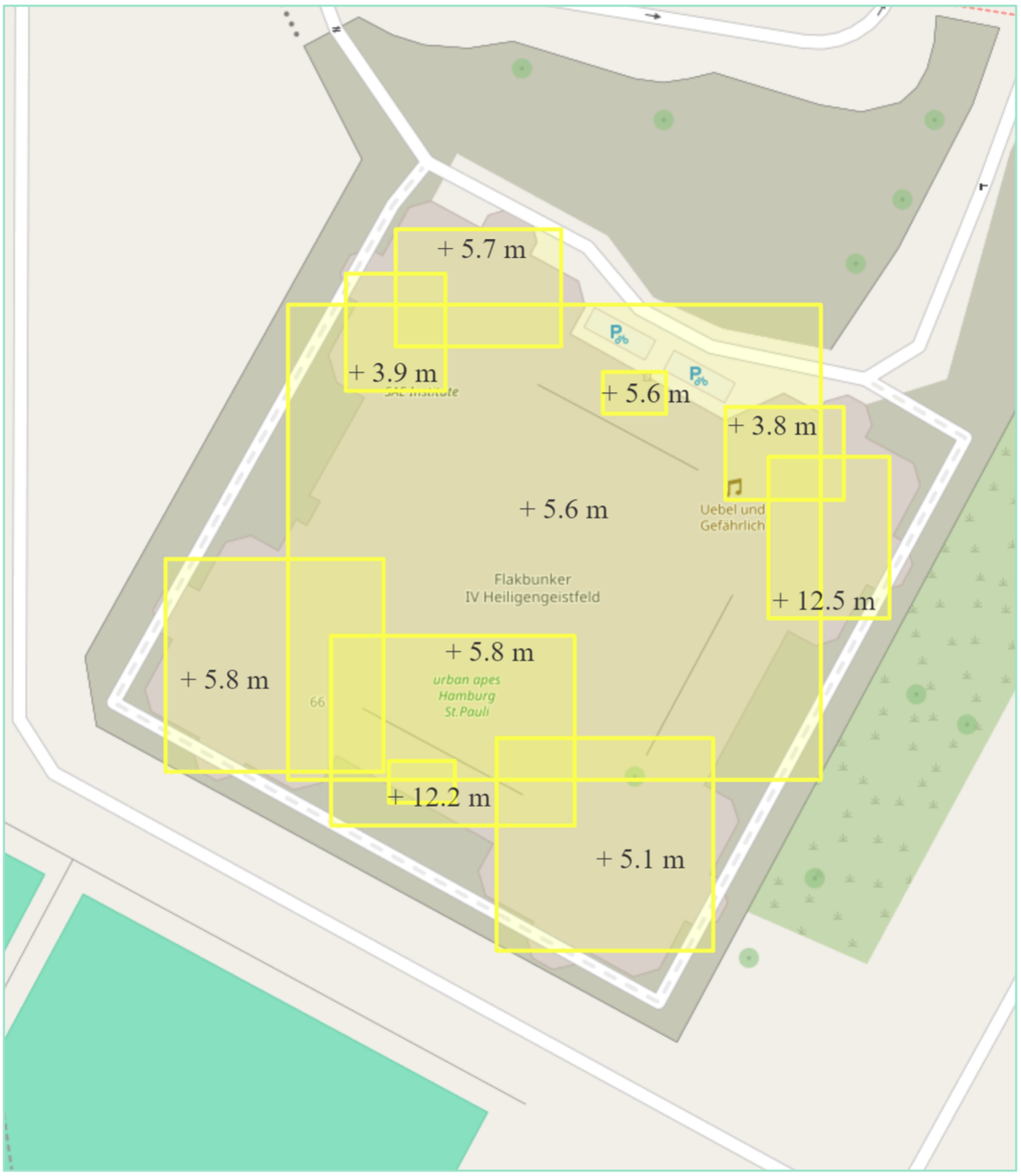


Figure 7.31.: A visualization of the bounding boxes of all ten building parts contained in the building Grüner Bunker. The depicted values above refer to the changes in the heights of all walls of each building part. In addition, all ground surfaces have been shifted downwards by 1.5 m. Visualized with Folium, Leaflet, and OpenStreetMap (OSM).

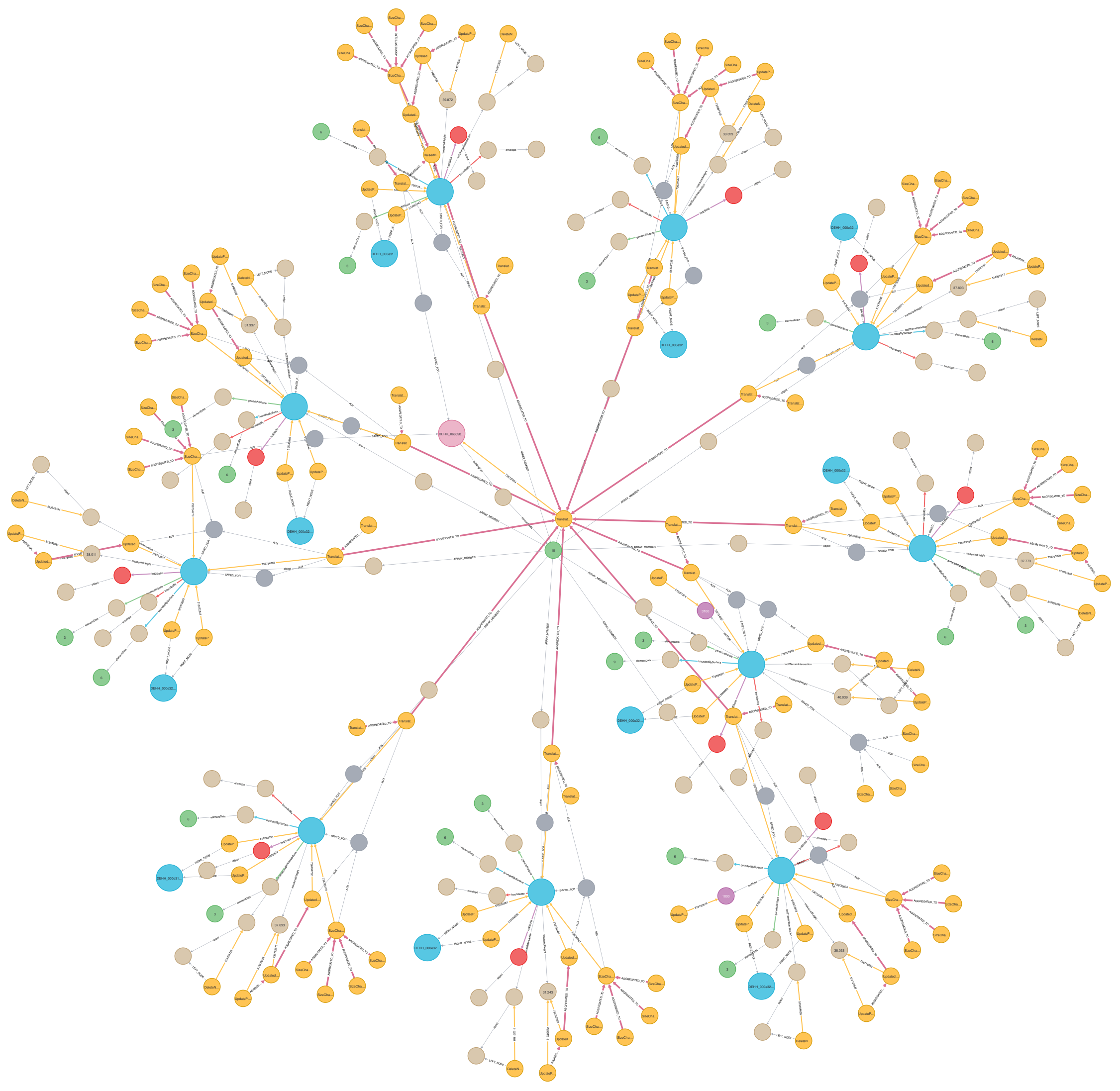


Figure 7.32.: A visualization in Neo4j Browser of the changes and their interpretation nodes (orange) in the graph representations of the building Grüner Bunker between 2016 and 2022. There are 20 building parts shown (blue), with 10 from each dataset. Changes are connected within a network and attached to corresponding building part nodes, allowing for efficient querying and analysis. The base changes include updated measured heights, increased wall heights, translated ground surfaces, deleted roof surfaces, inserted roof surfaces, and raised roof surfaces. The orange node in the middle with many incoming red relationships represents a consistent downward shift of 1.5 m of the ground surfaces in all building parts.

such building parts have been detected. While the changes associated with a roof raise are geometrically consistent, such as the height offsets of roof surfaces and the size changes of wall surfaces, the changes in the number of storeys of a building or building part often do not align with these geometric changes.

Table 7.9.: Example buildings (top) and building parts (bottom) detected with both raised roofs and changed number of storeys. As shown in the top table, while the height of a building is increased by 2.435 m, its number of storeys decreases by 1. Similarly, as shown in the bottom table, an increase of 5 storeys does not align with a small 0.164 m increase in height.

(a) Example buildings with both raised roofs and changed storeys

Before Update		*After Update*		*Difference in*	
Height (m)	**Storeys**	**Height (m)**	**Storeys**	**Height (m)**	**Storeys**
30.826	6	34.358	7	3.532	1
5.903	2	8.649	3	2.746	1
15.596	4	18.031	3	2.435	−1
7.924	1	9.876	2	1.952	1
21.906	3	23.827	5	1.921	2
		. . .			

(b) Example building parts with both raised roofs and changed storeys

Before Update		*After Update*		*Difference in*	
Height (m)	**Storeys**	**Height (m)**	**Storeys**	**Height (m)**	**Storeys**
21.728	1	23.563	6	1.835	5
21.714	1	21.878	6	0.164	5
22.125	1	22.345	6	0.220	5
24.175	5	24.946	9	0.771	4
26.851	5	28.155	9	1.304	4
		. . .			

For example, as shown in Table 7.9a, although the height of a building is increased by over 2 m, its number of storeys decreases by 1. This could suggest a change in the internal structure of the building or simply a correction of a previous inconsistency in the data. Similarly, as shown in Table 7.9b, an increase of 5 storeys does not align with a negligible 0.164 m increase in height.

Furthermore, in the use case of the building Grüner Bunker above, no information about the number of storeys is stored in any building parts. Thus, no changes in the number of storeys could be found for this building, despite the added new storeys in the real city.

Similar to the number of storeys, changes in the roof elevation may also be correlated with other properties, such as the functions, roof types, or any other relevant generic attributes available in buildings and building parts. However, while the functions and roof types are defined in the CityGML encoding standard, the thematic meaning of generic attributes often depends strongly on the use case and employed datasets.

Lastly, the interpretations of a roof raise carry significant importance, as one such interpretation can represent all translations and size changes within a building or a building part. As depicted in Table 7.10, each single interpretation of a roof raise in a building can represent an average of 10 and up to 447 base changes detected during the matching process. In total, these interpretations in buildings and building parts alone cover more than 1.5 million changes, which account for nearly one fifth of all detected changes and one third of all detected geometric changes.

Table 7.10.: An overview of changes represented by raised roofs in buildings and building parts. These base changes were detected during the matching process and are required for the creation of the interpretation nodes for raised roofs.

	Buildings	**Building Parts**
Changes interpreted per roof raise		
Minimum	4	4
Maximum	447	33
Average	10	7
Total number of		
Raised roofs	50,390	148,505
Represented changes	486,065	1,068,370
Among represented changes		
Updated properties	50,390	148,505
Translated surfaces	100,780	297,010
Resized surfaces	334,895	622,855

The pattern rules and examples presented in this use case are employed for identifying an increase in the roof elevation of a building or building part. The same can be applied to detect lowered roofs, as well as raised and lowered ground surfaces.

7.6. Leveraging Interpretation Results for Further Analyses

When a pattern among detected changes is identified, interpretation nodes are created and attached to the source content nodes. This not only enables efficient and direct access to the interpretation results stored in the graph database, as demonstrated previously, but also allows these results to be further utilized for more complex calculations and analyses, such as calculating the changes in the existing residential, commercial, and industrial space of Hamburg. Such a task would have been significantly difficult and time-consuming using 'only' the raw data provided by the original datasets. However, by leveraging the interpretation nodes produced, such analyses can be performed quickly and efficiently. In addition, the interpretation results can also be employed to assess the coverage of interpreted changes over all detected changes.

7.6.1. Calculating Changes in the Living Space of the City

Firstly, the total net change of all available 2D space or area of Hamburg between 2016 and 2022 can be computed as follows:

$$A = I + D + S \tag{7.1}$$

Where:

1. A is the net change of all available space in Hamburg.
2. D is the negative total ground area of all deleted buildings and building parts.
3. I is the positive total ground area of all inserted buildings and building parts.
4. S is the net sum of all detected size changes in the ground surfaces.

All I, D, S, and consequently A can be either positive or negative, indicating a net increase or decrease in the areas, respectively. The calculation of I, D, and S involves the ground surface area of each building and building part, which can be determined either by using their 2D bounding boxes or the measured surface areas given in the generic attributes of the ground surfaces.

The first option can always be performed, as the bounding boxes of buildings and building parts are automatically calculated and mapped onto graphs during the mapping process described in Chapter 3. However, this method generally can only provide an approximation of the actual results, as bounding boxes are axis-aligned while ground surfaces can have any orientations in the horizontal plane. Using the actual geometries like polygons available in the datasets would lead to more accurate results (Harter, 2021; Kaden, 2014), at the cost of increased computational complexity.

The second option can produce more reliable results, as it employs ground surface areas that were measured using additional methods and tools. However, such information, often stored as generic attributes in the ground surface, may not always be available, as in the case of Hamburg, without the use of additional tools such as the 3DCityDB. Therefore, the first option involving bounding boxes is employed for the calculation of ground surface areas of buildings and building parts.

In contrast to I and D, the computation of S additionally depends on the signed margin offsets in each dimension, which are provided by each base size change detected during the matching process. These margins specify the extent to which the bounding box of the surface expands in each direction. For example, a margin vector $(1, -1, 0)$ indicates that the new bounding box is 1 m wider in the x-dimension and 1 m narrower in the y-dimension. For a 2D bounding box of sizes (X, Y) from the old datasets and its resize margins $(\Delta x, \Delta y)$, the net change s in the ground surface area of a building or building part is computed as follows:

$$s = (X + \Delta x)\,(Y + \Delta y) - XY \tag{7.2}$$

Similarly, Equations (7.1) and (7.2) can be extended to include the third dimension to allow for the calculation of the total net change $V = I' + D' + S'$ in all available volumes of buildings and building parts in Hamburg. This can be performed based on either their 3D bounding boxes or their recorded number of storeys. However, not every building or building part is available with this number of storeys, as in the case of Hamburg, where only 56 % of all buildings have this attribute. Therefore, in this experiment, the 3D bounding boxes of buildings and building parts are employed instead.

The computed changes in the total ground surface areas and volumes of buildings and building parts within the Hamburg datasets are summarized in Table 7.11. As shown in this table, the city underwent a decrease of approximately 9 km^2 but at the same time an increase of 12 km^2 in total ground surface areas, resulting in a net gain of 3 km^2. Similarly, the city saw both a decrease of 81 million cubic metres and an increase of 133 million cubic metres in total volumes of buildings and building parts, resulting in a net increase of 52 million cubic metres. The major driving factor for these increases can be attributed to the newly inserted buildings and building parts, whose total ground surface areas and volumes outweigh those of deleted and resized buildings and building parts combined.

Notably, existing buildings (without building parts) that underwent a size change saw an increase in ground areas (634 m^2), but a decrease in volumes ($-40{,}399$ m^3). This might indicate that, on average, these buildings tend to become larger horizontally but shorter vertically. In contrast, existing building parts that underwent a size change saw a decrease in ground areas ($-13{,}542$ m^2), but a significant increase in volumes

Table 7.11.: Changes in the ground surface areas (top) and volumes (bottom) of all buildings and building parts of the entire Hamburg datasets.

(a) Changes in ground areas

Ground Area (m^2)			**Decrease**	**Increase**	**Net Change**
D	All deleted	Buildings only	6,515,813	0	−6,515,813
		Building Parts	2,246,003	0	−2,246,003
I	All inserted	Buildings only	0	11,322,486	11,322,486
		Building Parts	0	610,432	610,432
S	All resized	Buildings only	737	1,371	634
		Building Parts	42,578	29,036	−13,542
A	Total change in areas		8,805,131	11,963,325	3,158,194

(b) Changes in volumes

Volume (m^3)			**Decrease**	**Increase**	**Net Change**
D'	All deleted	Buildings only	53,704,133	0	−53,704,133
		Building Parts	27,422,302	0	−27,422,302
I'	All inserted	Buildings only	0	123,114,498	123,114,498
		Building Parts	0	9,737,172	9,737,172
S'	All resized	Buildings only	115,954	75,555	−40,399
		Building Parts	234,712	734,401	499,689
V	Total change in volumes		81,477,101	133,661,626	52,184,525

(499,689 m^3). This could suggest that these building parts tend to become narrower horizontally but taller vertically, indicating a trend towards **verticalization** of building parts in the city. Such information is of great interest to many stakeholders, particularly the urban planners and city mayor.

Lastly, the calculations presented above can be extended to further differentiate between residential, commercial, and industrial space or volume based on the property values *function*. While this attribute is available in every building in the Hamburg datasets, it is present in none of the building parts. Therefore, the functions of building parts are derived from those of the buildings they belong to. Table 7.12 summarizes the computed changes in the ground areas and volumes of buildings and building parts, divided into each category of the three categories: residential, commercial, and industrial.

Table 7.12.: Changes in the residential (*Res.*), commercial (*Com.*), and industrial (*Ind.*) ground surface areas (top) and volumes (bottom) of all buildings and building parts in the Hamburg datasets.

(a) Changes in ground areas

Ground Area (m^2)		**Deleted**	**Inserted**	**Resized**	**Net Change**
Res.	Buildings only	−1,401,275	4,027,089	−157	2,625,657
	Building Parts	−455,107	123,378	−6,371	−338,100
Com.	Buildings only	−501,083	1,014,970	−420	513,467
	Building Parts	−341,101	101,694	−4,724	−244,131
Ind.	Buildings only	−1,300,185	1,618,744	119	318,678
	Building Parts	−430,627	125,298	−795	−306,124
Total change in areas		−4,429,378	7,011,173	−12,348	2,569,447

(b) Changes in volumes

Volume (m^3)		**Deleted**	**Inserted**	**Resized**	**Net Change**
Res.	Buildings only	−13,140,040	47,926,052	−3,887	34,782,125
	Building Parts	−5,018,904	1,483,282	294,233	−3,241,389
Com.	Buildings only	−5,670,587	16,790,807	−8,756	11,111,464
	Building Parts	−5,151,485	1,663,301	104,811	−3,383,373
Ind.	Buildings only	−12,566,693	20,044,279	964	7,478,550
	Building Parts	−5,654,800	1,511,743	24,979	−4,118,078
Total change in volumes		−47,202,509	89,419,464	412,344	42,629,299

Table 7.12 shows that among these three categories, residential space saw the largest net expansion in Hamburg between 2016 and 2022, both in terms of area and volume. This is followed by the net increase in both commercial area and volume. On the other hand, the industrial area remains nearly unchanged, while the volume increases notably. This trend might suggest that during this timeframe, the focus of the city was primarily on creating more living space, followed by increasing sufficient amount of commercial space, while the industrial space barely expanded horizontally but saw a growth in vertical size.

7.6.2. Assessing Interpretation Coverage among Detected Changes

When interpreting a large number of detected changes based on a large number of predefined pattern rules, one common query arises as how to determine the coverage of matched patterns over all changes, as well as how to identify changes that still remain uninterpreted. A change is considered interpreted if it has been aggregated or acquired for the creation of a next higher-level change. Thus, a change is considered uninterpreted if it has not been aggregated to construct any higher-level changes, indicating that it has not yet been found in any change patterns by the interpreter. As described in Chapter 5, each time a new interpreted change is created, a connection is also established between it and all its preceding components. Therefore, such relationships in the graphs can be utilized to identify both interpreted and uninterpreted changes. Listing 7.10 presents a Cypher query to detect such interpreted and uninterpreted changes based on the results of the interpretation process.

```
// Search for interpreted changes (both base and intermediate changes)
MATCH (c:Change)
WHERE exists ((c)-[:AGGREGATED_TO]->()) // has been aggregated
RETURN c

// Search for uninterpreted changes
MATCH (c:Change)
WHERE NOT exists((c)-[:AGGREGATED_TO]->()) // not yet aggregated
  AND NOT exists (()-[:AGGREGATED_TO]->(c)) // not an aggregation
RETURN c
```

Listing 7.10: A Cypher query used to search for interpreted and uninterpreted changes among all detected changes.

Table 7.13 provides an overview of the coverage of interpreted changes over all detected changes in the use case of Hamburg. The table considers a change as interpreted if it belongs to at least one predefined pattern. Changes that are part of multiple patterns are counted only once in this table (the interpretation process, however, considers all potential patterns for each change regardless of whether it has been previously found in any patterns). The patterns employed in this experiment are outlined in Section 7.5.1.

As shown in the table, even with a small set of patterns used, more than 5 out of 9 million detected changes could be interpreted. Furthermore, alone the geometric pattern (and its intermediate components) for detecting raised roofs in buildings and building parts already accounts for 90.2 % of all surface translations, 74.2 % of all surface size changes, and 30.5 % of all updated measured heights of buildings and building parts.

Table 7.13.: An overview of the coverage of interpreted edit nodes and base changes over all detected changes in the use case of Hamburg. A change is considered interpreted if it belongs to at least one identified pattern. The change patterns employed in this experiment are outlined in Section 7.5.1.

Base Change		**Detected**	**Interpreted**	**Coverage**
Inserted Node	Address	930	930	100.0 %
	Boundary surface	2,659	1,515	57.0 %
	Building	30,532	30,532	100.0 %
	Building name	1,553	1,553	100.0 %
	Building part list	696	696	100.0 %
	Measured height	704	704	100.0 %
	Roof type	971	971	100.0 %
	Other geometries	1,462	0	0.0 %
Deleted Node	Address	1,792	1,792	100.0 %
	Building	23,322	23,322	100.0 %
	Building part	13,904	13,904	100.0 %
	Boundary surface	1,079,091	2,476	0.2 %
	Generic attribute	2,088	2,088	100.0 %
	Measured height	696	696	100.0 %
	Roof type	694	694	100.0 %
	Other geometries	659,366	0	0.0 %
Inserted Property	Number of storeys	711	711	100.0 %
Deleted Property	Number of storeys	768	768	100.0 %
Updated Property	Address	6,100	0	0.0 %
	Building name	663	0	0.0 %
	Building function	140,676	26,319	18.7 %
	Generic attribute	876,228	0	0.0 %
	Identifier	806,792	806,792	100.0 %
	Measured height	653,154	198,895	30.5 %
	Modification date	806,792	806,792	100.0 %
	Number of storeys	2,246	0	0.0 %
	Roof type	65,028	0	0.0 %
Geometric Change	Translation	1,143,966	1,032,293	90.2 %
	Size change	2,911,548	2,161,053	74.2 %
	Top-level split	2,692	2,692	100.0 %
Total number and coverage		9,237,824	5,118,188	55.4 %

Notably, inconsistencies were detected while parsing and matching the geometric contents of 2D surfaces in the original datasets, where all points of a surface are collinear, preventing the matching process from constructing a plane and normal vector for the given surface. In such cases, these geometries are considered invalid, thus not eligible for matching. This issue was observed in 2,476 surfaces from the 2016 datasets and 1,515 surfaces from the 2022 datasets. This leads to 2,476 deleted surfaces and 1,515 inserted surfaces, which are marked as interpreted in Table 7.13. Other deleted boundary surfaces include those that exceed the allowed thresholds for translation and rotation, as well as those differing in sizes and shapes from the newer ones.

The patterns used for interpreting the repurposing of buildings in Hamburg within the residential, commercial, and industrial space involves the code values in the ranges of $[1000, 1025]$, $[2000, 2100)$, and $[2100, 2200)$, respectively, as detailed in Section 7.5.5. These patterns account for approximately one fifth of all updated building function values, as shown in Table 7.13.

7.7. Multi-perspective Change Interpretation

Based on the stakeholders introduced earlier in Section 2.4 and their varying interests in different types of changes discussed throughout this chapter, a semantic change-stakeholder network representing the relevance relations between these stakeholders and interpreted changes can be established, as detailed in Section 5.5 and visualized in Figure 7.33.

The figure demonstrates how the complex and hidden interrelations between stakeholders and changes can be captured in one single graph. Starting from a node representing a stakeholder or an actor role on the left, paths can be traced across adjacent layers until the nodes representing changes on the right are reached. The existence of fully traced paths between the left and right layers indicates that relevance relations have been found between the corresponding stakeholders and changes.

For instance, the city planning department, represented by the city mayor, is interested in processes that may have an impact on the city's available living space, such as building repurpose, building renovation, floor addition, or historic preservation of buildings. These actions are reflected in the datasets through changes such as updated building functions, elevated roofs of buildings and building parts, as well as unchanged buildings (which might also be caused by an already ongoing renovation such as in the case of the Congress Center Hamburg detailed in Section 7.5.2).

Although the traced paths in this example are shown in the direction from left to right, the network is bidirectional, allowing for both forward and backward path-tracing techniques.

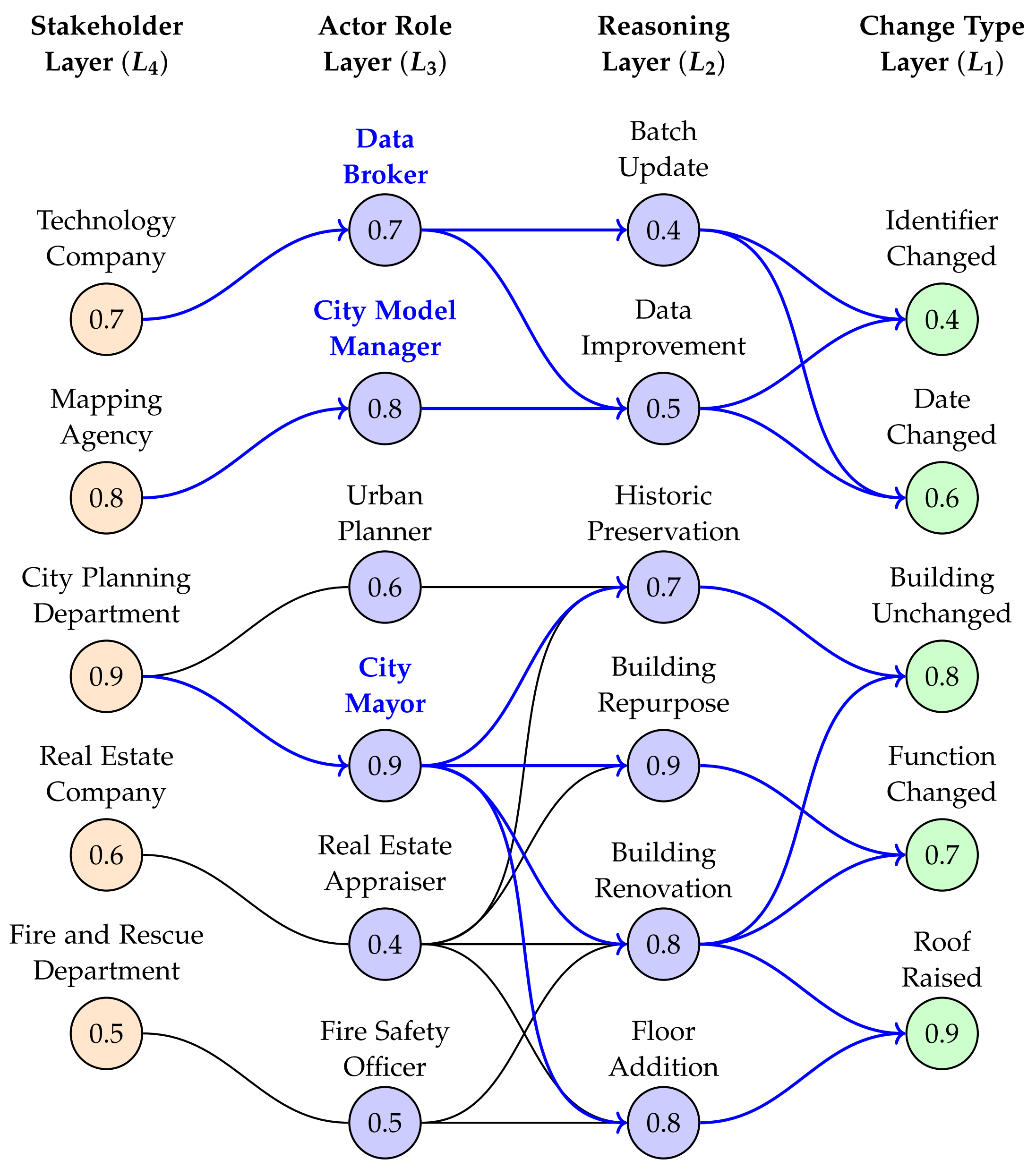


Figure 7.33.: A network visualization of the relevance relations between given stakeholders (two left-most layers) and interpreted changes (right-most layer) in the example of Hamburg. For a specific stakeholder, relevant changes can be found by tracing paths within the network from left to right (blue). Conversely, for a specific change, interested stakeholders can be discovered by tracing paths from right to left (not highlighted in this figure). For visual clarity, only normalized node weights are shown.

Moreover, even if two stakeholders share an interest in the same change, their levels of interest may differ. These interest or relevance levels can be represented by the weights of both the relationships and nodes in the network, as detailed in Section 5.5.2. For visual clarity, only normalized node weights are shown in Figure 7.33.

The normalized weights of the relationships within this network can be found in Figure 7.34, which provides a visualization of the network in Neo4j Browser. For visual clarity, only relationships in the backward direction from the Stakeholder Layer L_4 to the Change Type Layer L_1 are shown.

Listing 7.11 gives an example of how such a network, along with the weights of its nodes and relationships, can be constructed using Cypher. The weights assigned to nodes and relationships indicate the relevance levels of the concepts they represent, with higher values meaning higher relevance. All weight values provided in this example can be adjusted based on individual use cases. Furthermore, while only one path-tracing direction is selected, the same techniques can be applied to the other direction.

```
// Create nodes in the Change-Stakeholder Network (CSN)
MERGE (city_planning_department:STAKEHOLDER:CSN {
  name: "City Planning Department", weight: 0.9})
MERGE (city_mayor:ACTOR_ROLE:CSN {
  name: "City Mayor", weight: 0.9})
MERGE (building_repurpose:REASONING:CSN {
  name: "Building Repurpose", weight: 0.9})
MERGE (function_changed:CHANGE_TYPE:CSN {
  name: "Function Changed", weight: 0.7})

// Create relationships for backward path-tracing
MERGE (city_planning_department)-[:BACKWARD {
  weight: 0.9}]->(city_mayor)
MERGE (city_mayor)-[:BACKWARD {
  weight: 0.9}]->(building_repurpose)
MERGE (building_repurpose)-[:BACKWARD {
  weight: 1.0}]->(function_changed)
```

Listing 7.11: An example Cypher query for constructing a network representing stakeholders, changes, and their semantic relations.

A complete compilation of all Cypher queries employed in this thesis to construct the entire network can be found in Listing C.1. As the network is a type graph, this step only needs to be performed once.

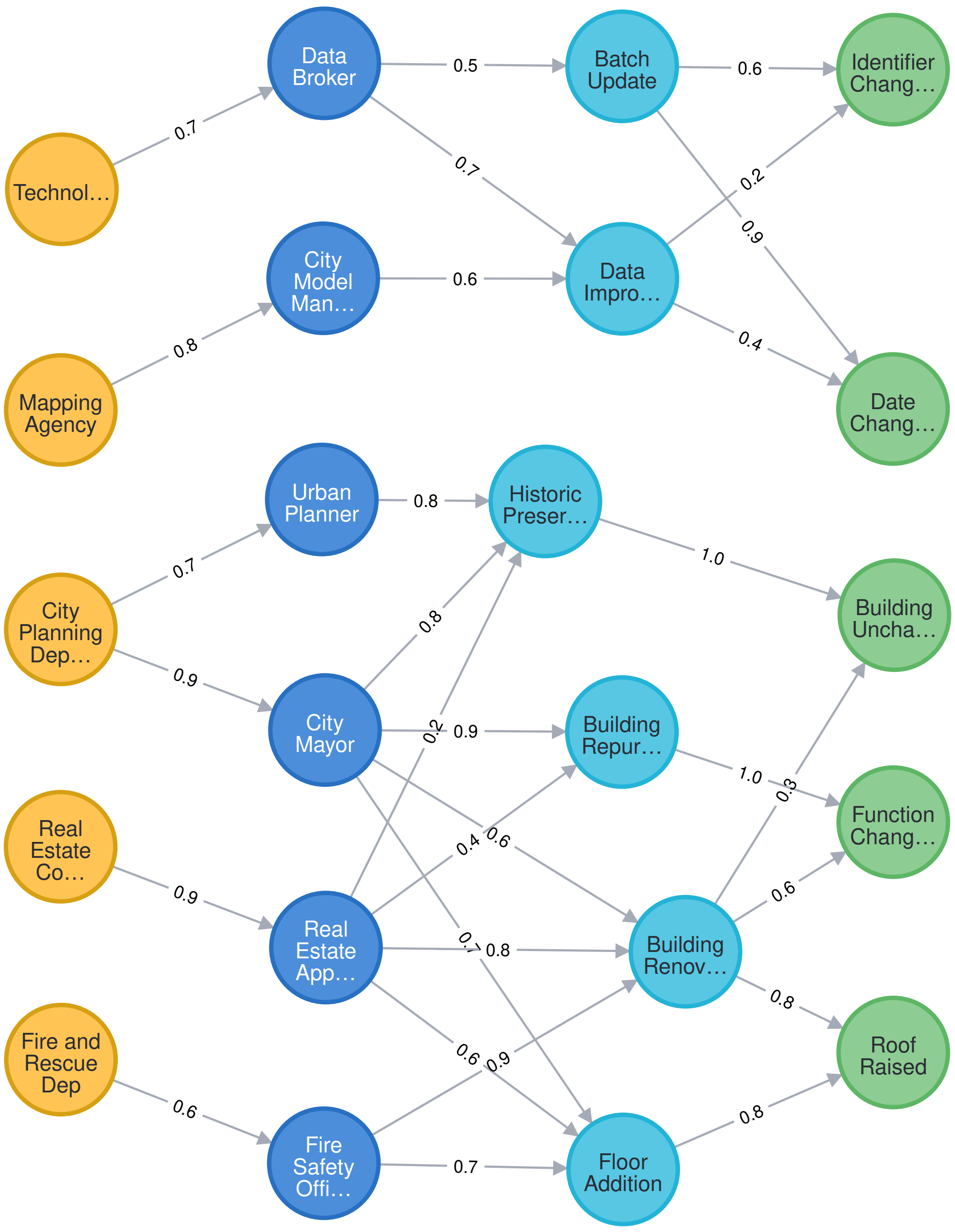


Figure 7.34.: A visualization in Neo4j Browser of the change-stakeholder network introduced in Figure 7.33. For clarity, only normalized relationship weights from left to right are shown, and connections from the change type nodes (green) to its instances detected between the datasets are omitted.

Based on the constructed change-stakeholder network, path-tracing techniques can be employed to determine, for example, which changes may be relevant to the city mayor representing the city planning department. These techniques were introduced in Section 5.5.3 and are illustrated in the following Cypher query shown in Listing 7.12.

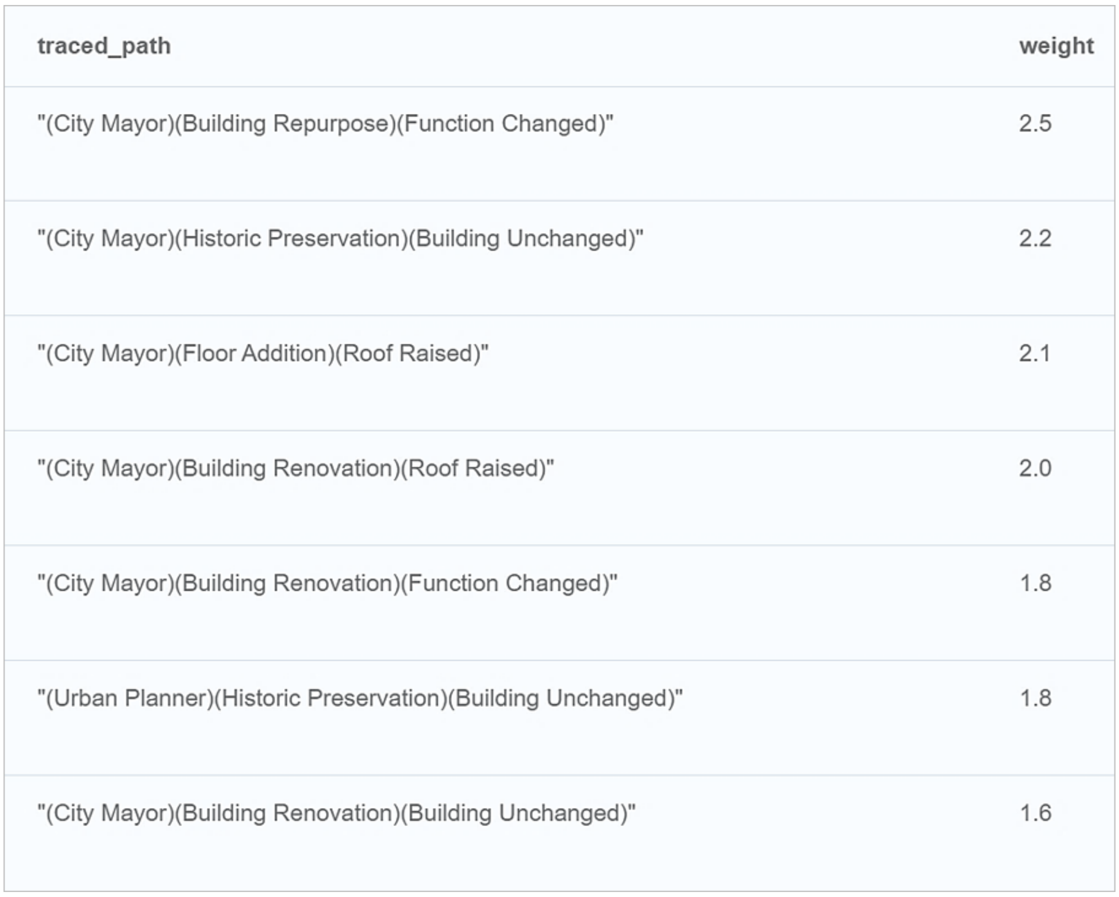
```
// Stakeholder-Change analysis
// Find all change types reachable from a given stakeholder
MATCH p=(city_planning_department:STAKEHOLDER {
  name: "City Planning Department"
})-[:BACKWARD*]->(c:CHANGE_TYPE)
// Display paths and calculate their accumulated weights
RETURN reduce(name="", r in relationships(p)
        | name + "(" + endNode(r).name + ")") AS traced_path,
      round(reduce(weight=0, r in relationships(p)
        | weight + startNode(r).weight * r.weight), 1) AS weight
// Sort traced paths by their weights in descending order
ORDER BY weight DESC
```

Listing 7.12: Path-tracing analysis in Cypher to evaluate relevance levels of changes for a specific stakeholder.

The query begins with a stakeholder node representing the city planning department on the left side of the network. It then traces all paths leading to change types on the right. The length of the paths, and consequently the number of layers in the network, can be arbitrary, as allowed by the notation '$*$' in the query. The relationship type *BACKWARD* implies the use of backward path-tracing techniques in a consistent direction, from the Stakeholder Layer L_4 to the Change Type Layer L_1. Finally, the fully traced paths are evaluated based on their accumulated weights using Equation (5.1). The results are presented in Figure 7.35.

The results show that, based on the node and relationship weights in this specific example, the city mayor is primarily interested in buildings that may have been repurposed, such as those transitioned from industrial to residential use, as indicated by the updated function values of the buildings. On the other hand, an urban planner, another representative of the city planning department, is more interested in unchanged buildings that are part of a historic preservation program.

Each change type node at the end of the fully traced paths represents all instances of that change type. Listing 7.13 provides an example Cypher query for connecting such a change type node with its corresponding instances stored in the graphs. For instance, the node *Function Changed* represents all 140,676 buildings with updated function values, as showcased previously in Section 7.5.5.



traced_path	weight
"(City Mayor)(Building Repurpose)(Function Changed)"	2.5
"(City Mayor)(Historic Preservation)(Building Unchanged)"	2.2
"(City Mayor)(Floor Addition)(Roof Raised)"	2.1
"(City Mayor)(Building Renovation)(Roof Raised)"	2.0
"(City Mayor)(Building Renovation)(Function Changed)"	1.8
"(Urban Planner)(Historic Preservation)(Building Unchanged)"	1.8
"(City Mayor)(Building Renovation)(Building Unchanged)"	1.6

Figure 7.35.: The Cypher results of the path-tracing analysis for the stakeholder *City Planning Department*. All traced paths are evaluated and sorted by their accumulated weights. The traced path through the nodes *City Mayor*, *Building Repurpose*, and *Function Change* has the highest accumulated weight of 2.5, indicating that repurposed buildings are of the greatest interest to the city mayor. On the other hand, unchanged buildings, with an accumulated weight of 1.6, are of the least interest to the city mayor among the listed paths.

```
// Connect a change type node with all instances
MERGE (:CHANGE_TYPE:CSN {name: "Function Changed"})<-[:INSTANCE]
-(:CHANGE {change_type: "UpdatedBuildingFunction"})
```

Listing 7.13: An example Cypher query for connecting a change node type with all its instances previously created during the matching and interpretation process.

These changes, aggregated by the interpretation process based on predefined change pattern rules, are attached directly to building nodes. As a result, buildings affected by a given change can be efficiently retrieved using their existing connection, as illustrated in Listing 7.14.

```
// Retrieve all repurposed buildings
MATCH (b:BUILDING)<-[:ATTACHED]-(:CHANGE)
  -[INSTANCE]->(:CHANGE_TYPE:CSN {name: "Function Changed"})
RETURN b
```

Listing 7.14: An example Cypher query for retrieving all buildings corresponding to a change type node. This requires existing connections between the change type node and all its instances.

Since all individual changes and their interpreted changes are connected within a change network, the path-tracing techniques can trace from a starting stakeholder node all the way to individual building nodes and even further to the smallest, lowest-level individual changes stored deep within the database. This automatic approach requires no expert knowledge of the database structure when performing path-tracing analyses.

However, this may require a significant number of new relationships. In the example above, with the change type node *Function Changed*, 140,676 additional relationships are created. This also means that any newly interpreted change must also be linked to its type node, leading to high maintenance costs if the interpretation process is performed multiple times.

Alternatively, the explicit relationships between change type nodes and their instances can be omitted. Building nodes can be queried at runtime based on the change type by leveraging semantic indexing on node labels and properties. This approach avoids a high number of additional relationships, thereby eliminating the need to repeatedly link newly created change instances with their corresponding type nodes. However, querying using semantic indexes is generally slower than using explicit connections. Moreover, this approach cannot be automatically applied when integrated into the path-tracing analyses between stakeholders and changes.

7.8. Runtime Complexity and Scalability

To evaluate the runtime complexity and scalability of all methods proposed in this research, several subset datasets, in addition to the full Hamburg datasets, are also employed. These subsets are created by selecting the first 25 %, 50 %, and 75 % of tiles from the complete datasets, arranged in ascending alphabetical order.[10] Each tiled dataset is named using an ordered unique identifier, ensuring that a lexicographical sort also corresponds to a spatial sort of the tiles.

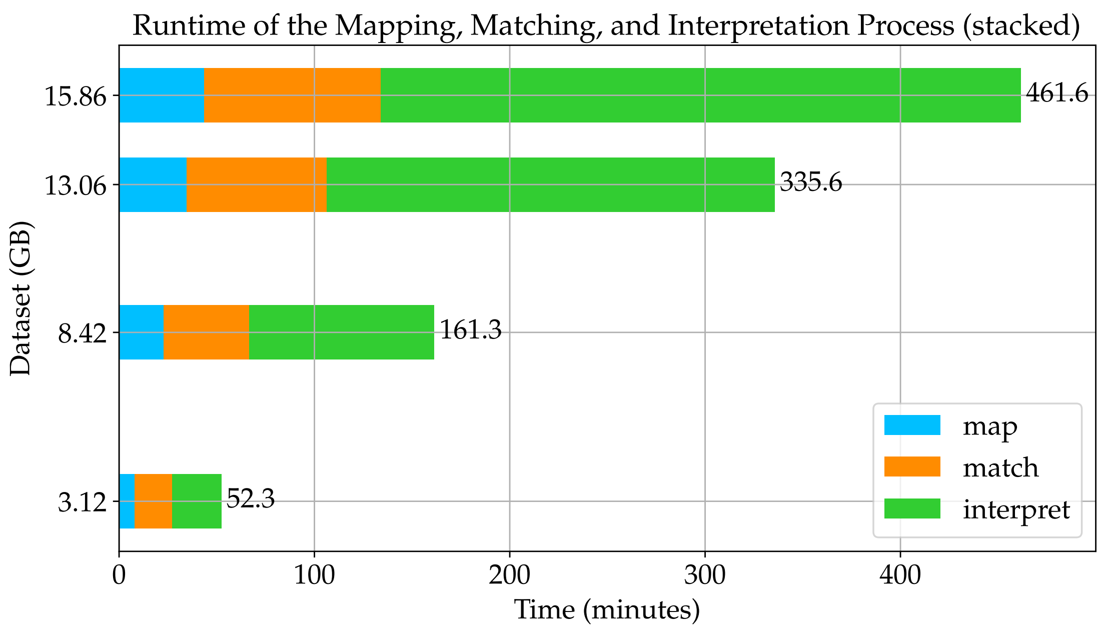
However, despite the number of selected tiled documents being proportional, the number of buildings and the total disk size of each subset dataset may not be proportional to 25 %, 50 %, and 75 % of those of the full datasets. This is due to the fact that many tiled documents may be empty, such as those located between the island of Neuwerk and the city of Hamburg, as shown in Figure 7.23. A summary of all datasets employed in this study is shown in Table 7.14.

Table 7.14.: An overview of all CityGML datasets used for assessing the processes proposed in this research.

Employed Dataset		Nr. of Tiles	Nr. of Buildings	Total Size
Hamburg 2016				
- subset	25 %	197	81,923	1.61 GB
- subset	50 %	394	196,595	4.06 GB
- subset	75 %	591	304,341	6.25 GB
- subset	100 %	788	374,990	7.61 GB
Hamburg 2022				
- subset	25 %	222	73,224	1.51 GB
- subset	50 %	443	197,890	4.36 GB
- subset	75 %	665	312,527	6.81 GB
- subset	100 %	887	383,439	8.25 GB

In this experiment, the complete workflow, containing the mapping, matching, and interpretation process, was performed three times for each dataset. An average runtime was then calculated to provide a more reliable value. These average values of the total runtime, along with the runtime of each process, are visualized in Figure 7.36.

[10]An alternative approach involves importing all tiled datasets from both 2016 and 2022 into two separate database instances of the 3DCityDB, followed by re-exporting them as CityGML datasets representing the same spatial region. This ensures that the exported subsets are spatially comparable.



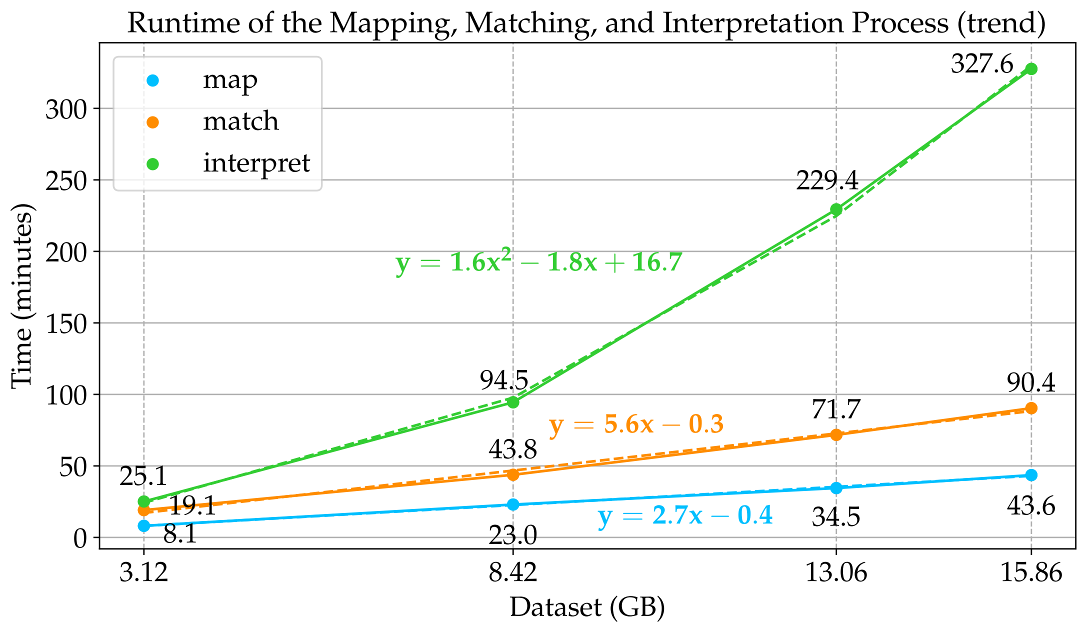


Figure 7.36.: A visualization of the average runtime of the mapping, matching, and interpretation process employed for the Hamburg datasets from 2016 and 2022. While the runtime for the mapping and matching process exhibits a linear growth (blue and orange), the runtime of the interpretation process follows a quadratic polynomial trend (green).

The mapping of the entire Hamburg datasets of 2016 and 2022 took approximately 44 minutes. This corresponds to a mapping speed of 290 buildings per second. More than 30 % of the total runtime for mapping is allocated for resolving XLinks, a process that connects all isolated graph representations of CityGML objects to form a cohesive connected graph, as described in Section 3.4.

The process of matching these fully mapped datasets took an additional 1.5 hours. This corresponds to a comparison speed of 140 buildings per second or a detection speed of more than 1,700 changes per second.

Additional 5.5 hours were needed to match, identify, and interpret the predefined patterns among detected changes, which corresponds to an interpretation speed of nearly 40 buildings per second or an aggregation speed of 470 changes per second. A substantial part of this time was allocated for the evaluation of complex conditions required for the creation of the next higher-level changes using a JavaScript engine. The script engine allows for comprehensive use and parsing of complex boolean expressions, along with the application of user-defined functions, at the cost of additional runtime.

The mapping, matching, and interpretation process of the entire Hamburg datasets account for 10 %, 20 %, and 70 % of the total runtime, respectively. While the runtime of the mapping and matching process exhibits a linear trend, the runtime of the interpretation process follows a quadratic polynomial trajectory, as approximated at the bottom of Figure 7.36.

The polynomial fitting functions of these process are detailed as follows:

$$\text{Mapping Process} : y = 2.7x - 0.4 \tag{7.3}$$

$$\text{Matching Process} : y = 5.6x - 0.3 \tag{7.4}$$

$$\text{Interpretation Process} : y = 1.6x^2 - 1.8x + 16.7 \tag{7.5}$$

In the fitting functions depicted above, the x values denote the disk sizes of the employed datasets (in gigabytes), and the y values represent their corresponding runtime (in minutes). As observed at both the top and bottom of Figure 7.36, despite being selected proportionally to the number of tiled documents, the disk sizes of the employed datasets are disproportional to the number of their respective tiles, as indicated by the unequal spacing between the datasets in the figures.

As previously discussed in Section 4.1.3, the graph and subgraph isomorphism problem generally pose significant challenges due to its high complexity, particularly in large graphs. However, by leveraging optimization and heuristic strategies based on the semantic and geometric information available in CityGML documents, the graph matching methods proposed in this thesis demonstrate an efficient linear runtime complexity, as shown in Figure 7.36 and Equation (7.4).

8. Conclusion and Outlook

This chapter summarizes the methods proposed in this thesis, discusses their strengths and limitations, and highlights the contributions of this research to other studies within similar fields. Lastly, the chapter outlines how the work can be adjusted, extended, and utilized in future applications and studies.

8.1. Summary of this Work

Within the context of smart cities and semantic 3D city modelling, this thesis introduces the concepts of Urban Digital Twins (UDTs) in general and emphasizes those that utilize a semantic 3D city model, predominantly given in CityGML, as one of their core components for the virtual representation of the physical city. As one of the defining features of an urban digital twin, the bidirectional data flow between the physical city and its virtual counterpart requires automatic and efficient change detection between different temporal versions of both the real world and its virtual city model. With the focus on the virtual side, this research shows the challenges and complexities of not only detecting changes in CityGML documents but also deciphering them to gain valuable insights into their interrelationships. Thus, this research further explores how the detected changes can be interpreted by revealing and studying their hidden patterns, and how these interpretations can be leveraged to benefit various stakeholders.

The thesis addresses this problem by first highlighting the structural similarities between CityGML documents and graphs. It proposes an efficient method for mapping text-based CityGML documents onto graphs stored in a graph database. The use of graphs allows for not only efficient querying but also interactive visualization of CityGML objects and their complex interrelationships. The mapping process causes no information loss in all tested datasets and can be employed to map all objects across all fourteen thematic modules and all five different Level of Details (LODs) of CityGML. To manage large CityGML datasets, as in the case of Hamburg, the input documents are first divided into smaller pieces, each of which can then be mapped concurrently onto graphs. However, the resulting graphs become disconnected due to these pieces being separated from the original CityGML document. To address this, XLinks can be resolved to reconnect generated subgraphs. This resolution process can also be employed to replace existing XLink connections with explicit graph relationships, such as between a solid and its boundary surfaces each referenced through an XLink.

Given both the complexities and limitations of directly comparing CityGML documents in their text form, this thesis proposes employing their graph representations as a basis for the comparison of the original CityGML documents. However, matching graphs, a problem often referred to as the graph and subgraph isomorphism problem in graph theory, presents numerous challenges, especially in large graphs. To address this, optimization and heuristic strategies are introduced, such as those that utilize the semantic labelling of nodes and relationships, as well as spatial indexing for efficient querying of city objects with geometric content. Despite these optimizations, multiple matching candidates may still exist for a reference object, especially in one-to-many and many-to-many relationships. The optimal match among these matches is determined based on its semantic and geometric similarity levels to the reference object. For instance, among all points located within the vicinity of a reference point, the point with the minimum distance is considered the best match. A surface is considered an optimal match if it has minimum difference in orientation and overlapping volume with the reference surface in 3D space. During these geometric calculations, error tolerances, such as for lengths, angles, areas, and volumes are taken into account. Additionally, the matching process also considers the possibility of translations and size changes among geometric objects in 3D space, as well as split changes of top-level features. Whenever a change is detected, a node representing this change is created and attached to the source nodes where the change occurred, enabling efficient retrieval of these changes and their context.

Identifying changes is only half of the solution; the other half is to understand them. The matching process of large CityGML datasets often results in millions of changes, rendering them impossible for humans to comprehend. While an individual, isolated change may not provide any crucial information, collectively, they can reveal hidden patterns with valuable insights. However, a pattern may consist of numerous changes, while a change may be part of multiple patterns at the same time. Therefore, a rule network is proposed to allow for describing all rules for matching such complex change patterns in one centralized type graph, thereby eliminating any redundancies. Rule nodes and relationships can be assigned with various properties and directives to guide the interpretation process, such as specifying the next content type to search for while navigating within the graphs. These rules are aggregative, reducing a large number of low-level changes into a significantly smaller number of higher-level semantic changes. As a result, a single interpreted change may represent hundreds or thousands of lower-level changes, as observed with the patterns for raised building roofs. Like changes, when a pattern has been matched, a new interpretation node is created and attached to the corresponding source content nodes. Given the reduced number of interpreted changes and their high semantic levels, the interpretation results become more manageable for both humans and machines. Based on these generated interpretation nodes, meaningful interpretations and efficient analyses can be performed.

However, the perception of these changes varies significantly among stakeholders. While a data broker may be interested in the changes made to the geometric representations of buildings, a city mayor has great interest in changes that could impact the city on a broader scale, such as fluctuations in the city's residential, commercial, and industrial space. Even when two stakeholders share an interest in the same change, their levels of interest may differ. Such complex interrelations between stakeholders and changes, as well as within stakeholders and changes themselves, can be captured in one semantic layered network. Each layer can represent all changes, the actions that may have caused them, or the stakeholders. These layers are serially connected with bidirectional relationships, allowing for analyses between stakeholders and changes in both directions: (1) given a change, find interested stakeholders, and (2) given a stakeholder, identify relevant changes. This can be achieved by employing path tracing among layers, a technique that allows for the efficient identification of reachable nodes in the network from a starting node.

Optimization strategies for dealing with massive CityGML datasets are proposed. These strategies leverage both the built-in thematic indexing provided by the graph database Neo4j and the implemented spatial indexing using an R-tree. Moreover, to improve runtime efficiency, parallel execution of the mapping, matching, and interpretation process is performed. To avoid concurrency issues such as deadlocks, the processes are executed in separate database transactions with isolated input and output resources.

Finally, the methods proposed in this thesis are evaluated using the massive CityGML datasets of Hamburg from 2016 and 2022. Detected changes and their corresponding patterns are shown and analysed. Some interesting findings and observations about the Hamburg datasets are also discussed and visualized. The research demonstrates how these interpreted changes can be utilized for further analyses, such as calculating the changes in residential, commercial, and industrial space in the city.

8.2. Discussion and Contributions

This section discusses the strengths and limitations of the mapping, matching, and interpretation process. It then highlights the scientific contributions of this thesis to relevant research fields.

8.2.1. Strengths and Limitations

All methods in this research are graph-based and operate on the same centralized graph consisting of two subgraphs for the old and new CityGML documents. The mapping process generates these graphs that account for circular references, ensuring accurate representations of the original CityGML documents. However, for the subsequent processes to terminate, these graphs must be acyclic. As a result, the matching and interpretation process are applicable only to CityGML version 2.0 datasets, which is the focus of this thesis, as version 3.0 datasets often include circular links. Moreover, to guarantee that all information is processed, all content nodes of a CityGML document must be reachable from the source node representing its city model. As a result, CityGML graphs are weakly connected. These prerequisites, as outlined in Research Question RQA3 (Graph Data Model for CityGML), were addressed in Section 3.2.

The Mapping Process

A major advantage of the proposed mapping methods is their generic nature, allowing for graphs to be generated from across all fourteen thematic modules in five different LODs of CityGML. This was proposed in Research Question RQA4 (Mapping Methods and Evaluation). Although the primary focus of this research is on CityGML version 2.0, the mapping process has been successfully employed to produce lossless graph representations of CityGML documents in version 3.0. These graphs enable further analyses and applications, such as multi-modal navigation within the city using the graph representations of the new street space models in CityGML version 3.0 (Olbrich, 2023; Olbrich et al., 2024). In addition to CityGML, these methods can also be applied to map other document types and exchange formats, including CityJSON, onto graphs.

While spatial RDBMSs offer a wide range of geometric and topological capabilities that Neo4j lacks, Neo4j, like most graph databases, is schema-less, providing great flexibility when performing write operations. For instance, neighbourhood relationships between building nodes can be created to represent physical proximity, a task easily handled in graph databases but difficult using a relational structure.

In response to Research Question RQA7 (Reconstruction of Graphs to CityGML Objects), methods for reconstructing generated graphs back into their original in-memory CityGML objects are also proposed. This is particularly useful when handling geometric objects that can be defined in various syntactic ways. These reverse methods can be further extended to allow for both the import of CityGML documents into the graph database and the export of the generated graphs back into their original CityGML documents. This enhanced set of functionalities opens up new possibilities for a comprehensive graph database management system for urban objects.

However, while the mapping methods are independent of programming languages, only a select few languages can effectively realize these. These languages must fully support object-oriented modelling, allow for the chunk-wise reading and parsing of CityGML documents, and grant complete access to the internal structure and content of in-memory objects. As a result, Java was chosen due to its extensive support for object-oriented programming, the library *citygml4j* for reading and parsing CityGML documents, and the capability of the Java Reflection API for accessing and manipulating the internal structure and content of objects. Moreover, the employed graph database Neo4j is also written in Java (Neo4j, 2023).

The Matching Process

One of the biggest strengths of the matching process is its ability to quickly identify the best match for a given subgraph, as required by Research Question RQB8 (Finding Best Match). This is achieved by utilizing the rich semantic content available in the graphs as well as thematic and spatial indexing among graph entities. Moreover, in response to Research Question RQB5 (Syntactic Ambiguities), the matching process is able to match geometric objects regardless of how they were originally defined in the CityGML documents. In addition, the comparison of these geometries considers both potential translations and size changes of surfaces in three dimensions, with error tolerances taken into account. This corresponds to Research Question RQB6 (Geometric Uncertainties) and Research Question RQB7 (Geometric Transformations).

Despite the high complexity of the graph and subgraph isomorphism problem, the matching process presented in this study exhibits linear runtime complexity. This efficiency is achieved through a combination of various optimization and heuristic strategies introduced throughout this thesis, which leverage the semantic and geometric information available in the graph representations of CityGML documents. This refers to Research Question RQB2 (Graph and Subgraph Isomorphism) and Research Question RQB3 (Advantages and Challenges of CityGML Graphs).

The implementation of the matching process relies on external geometry libraries for handling complex geometric objects in 3D space, such as merging adjacent coplanar 2D surfaces or the calculation of their normal vectors. Despite their effectiveness, these libraries do not provide additional handling for invalid geometries, such as when all points of a surface are collinear or not all points are coplanar. Such cases can occur, as observed in the Hamburg datasets. In these cases, the matching process simply marks the invalid geometries as unmatched. Moreover, while the use of various thematic and spatial indexing allows for efficient retrieval of indexed nodes and relationships, it requires additional disk space storage. For example, the R-tree for indexing building footprints in Hamburg alone consists of approximately 80 million nodes.

The Interpretation Process

During the interpretation process, the rule network serves as both a descriptor for the complex interrelations among changes and a repository of predefined rules for matching patterns among these changes. Since the rule network is a type graph, each of its nodes represents all instances of a specific change type. This enables the capture of the dependencies among all patterns without redundancy, addressing Research Question RQC3 (Rule Definition for Change Patterns).

One of its major advantages is that, despite its compact size, a rule network can be applied to entire graphs as the interpreter searches for all instances represented by each rule node. Another advantage is its adaptability, allowing users to construct their own rules for matching change patterns using the predefined node and relationship properties. Complex conditions and user-defined functions required for the creation of higher-level interpreted changes are supported, which are then evaluated using an internal script engine.

However, in its current state, the rule network can only describe aggregative patterns among changes. While more complex user-defined functions given in an external script file are supported, they only have access to the data directly linked to the changes they process, such as properties stored in change nodes. Thus, complex computations that involve data stored deeper within the graphs, such as the calculation of the bounding box of each building, must be performed first during the matching process before their results can be utilized by the patterns. In addition, while the script engine plays a crucial role in processing complex boolean expressions with shared local parameters among adjacent rule nodes, it requires time to initialize and evaluate every condition, leading to increased runtime.

While the interpretation process can provide significant insights into the change patterns, it is limited to identifying only those patterns that are provided in the rule network. Moreover, despite the interpretation process being fully automatic, additional post-processing steps may be performed to leverage the interpretation results for further analyses, such as those proposed in Research Question RQC6 (Change-Stakeholder Model) and Research Question RQC7 (Graph-based Change-Stakeholder Analysis).

While the proposed framework offers robust tools for defining rules, its results would benefit from an empirical investigation into changes in semantic 3D city models. Key aspects include when changes occur, their types, who causes them, the real-world changes they reflect, and typical scopes for certain types of objects and changes.

As observed in the use case of Hamburg in Section 7.5.5, not all patterns need to be overly complex to reveal valuable insights. For instance, an updated building function from residential to commercial indicates its repurposing, resulting in a decrease in living space and an increase in commercial space. Similarly, while the modification date of a single building may seem minor, a shared update among all buildings within a region may indicate a mass update.

On the other hand, changes can sometimes reveal the opposite of what they appear to represent. For instance, a building that remained unchanged in the datasets could still be undergoing renovations in the real world, as these real-world changes may not yet be reflected in the datasets, as observed in the case of the Congress Center Hamburg explained in Section 7.5.4.

8.2.2. Scientific Contributions

This thesis stands at the intersection of multiple disciplines, offering scientific contributions to each of these fields through its introduction of novel concepts and methods. The disciplines benefiting from this work include Geographic Information System (GIS), Computer Science, and Graph Theory.

Contributions to the Field of Geographic Information System (GIS)

This research aligns with numerous other studies in demonstrating the potential and usability of graphs for storing, representing, managing, analysing, and processing urban data. These graphs may be referred to by different terms, such as ontology or knowledge graphs (Ding et al., 2024), or semantic networks (Nguyen & Kolbe, 2022), but they essentially originate from the same concept: a centralized, attributed graph capable of storing various types of information of the entire city. To enable the creation of such graphs, this study provides a novel method to map any city models or objects onto graphs. These graphs can then be enriched with additional information over time, allowing for condensing all types of information in one place. Based on these graphs, complex urban analyses can be performed.

This study is one of the first to provide a full implementation for mapping both CityGML versions 2.0 and 3.0 completely onto graphs in Neo4j. The methods are designed for parallelization in multicore systems, enabling high-performance mapping of massive city models, such as the mapping of the entire Hamburg datasets containing over 750 thousand buildings in under 45 minutes. Additionally, the research is also among the first to address the challenges of both comparing semantic 3D city models and interpreting their changes with respect to different perspectives of stakeholders.

Contributions to the Field of Computer Science

When handling massive datasets, this study employs a number of efficient algorithms and data structures for its methods, especially during the matching process. In many-to-many relationships, a brute-force approach could lead to the comparison of all possible pairs (a Cartesian product between the sets of matching candidates). To avoid this, the matching process leverages an R-tree for determining matching candidates from millions of others in logarithmic time. However, this often results in multiple candidates found. To further reduce this number to a single optimal match, this study introduces the concept of similarity levels, a metric used to categorize and sort matching pairs of objects based on their geometric and semantic resemblance. The matching pairs with the minimum or maximum similarity level (depending on the implementation) are considered the best match.

During the interpretation process, as a pattern is being processed and its components are being collected, a memory node is created and attached to a content node that corresponds to this pattern. This memory node stores temporary information crucial to the interpretation process, such as the type of the current pattern, the number of each change components required to activate this pattern, and the number of change components acquired thus far. To some extent, these memory nodes are functionally similar to the global working memory utilized in Rete networks. However, these memory nodes are decentralized and only keep track of the counts of collected elements, a mechanism similar to those in Petri nets. Unlike the standard Petri nets, where tokens are indistinguishable, each change component can be identified based on it type, attributes, and its location in the graphs. This combination of the strengths of both Rete networks and Petri nets allows the interpretation methods to efficiently process all detected changes without repetition, thereby enhancing the runtime complexity and reducing the memory consumption.

Contributions to the Field of Graph Theory

The methods for matching massive graph representations of CityGML documents presented in this thesis demonstrate how semantic and spatial information of the graph contents can be combined as heuristic optimizations to boost the performance of graph matching, a problem that is generally challenging in the field of graph theory.

The syntactic ambiguities allowed in CityGML often cause a typical problem when matching subgraph representations of CityGML objects, where an object can be defined using different syntactic methods, resulting in different subgraphs of the same content. In such cases, a subgraph isomorphism would fail to deliver a match. To address this issue, the matching process compares not only the structure of subgraphs but also their semantic contents, such as when comparing the graph representations of two geometrically equivalent polygons that were defined in different syntactic ways. Once two subgraphs are identified as a match using this approach, there is no longer a need to compare their internal structure and sub-elements. This strategy contributes to improved runtime complexity of the matching process.

8.3. Extendability and Future Work

In the context of Urban Digital Twins (UDTs) with a semantic 3D city model as one of their core components, the concepts proposed in this study can be utilized to enable automatic detection and interpretation of changes across different temporal versions of the virtual city model. The resulting interpretations can not only reveal insights into changes in the past, but also predict planned changes to be made to the real city in the future, such as those resulting from a simulation. However, the applicability of the concepts and methods proposed in this research is not limited to urban digital twins. They can be adapted, modified, and applied to other application domains, where the ability to detect and interpret changes in the systems is required.

The graph-based nature of CityGML largely originates from its inheritance from GML, which is based on the International Organization for Standardization (ISO) 19100 series. Therefore, all concepts and methods introduced in this thesis can also be extended and applied to domains with datasets based on GML or those conforming to the ISO 19100 series (Cox et al., 2004), including ALKIS (AdV, 2008) and IndoorGML (Jang et al., 2023; Lee et al., 2020). This also extends to datasets with comparable semantic and geometric contents, such as those found in the built environment sector, including Building Information Modelling (BIM) and Industry Foundation Classes (IFC) standards (Esser, 2024; Kolbe & Donaubauer, 2021).

The edit operations produced during the matching process can be utilized to update the old CityGML document to the state of the new one. These edit operations can be either employed directly in the graphs or converted into SQL-based operations to update relational representations of CityGML objects, such as those in the 3D City Database (3DCityDB). Additionally, the interpretation results can also be utilized to filter these edit operations prior to updating, such as to allow only changes that reflect actual changes in the physical world. Together with the mapping process and its methods for reconstructing graphs back into CityGML objects, the processes introduced in this thesis can be employed to implement a comprehensive graph database management system for urban objects.

Since changes are attached directly to source nodes in the graphs during the matching process, it is possible to pinpoint the exact location and extract the content related to these detected changes. By using a combination of the function *toObject(node)* proposed in Section 3.6 to convert any subgraph representation back into its original in-memory CityGML object, and the library *citygml4j* to convert it to its corresponding XML-encoded element, both the states before and after the change of a city object can be retrieved and exported in CityGML format. This is particularly useful when, for example, enriching an existing CityGML document, such as the newer one, with the *Version* and *VersionTransition* features that explicitly capture all changes between the older and newer document, as conceptualized in the new versioning module of CityGML 3.0 (Kolbe et al., 2021). However, this requires that both the matching and interpretation process are compatible with the newer version of CityGML.

While the methods for the matching and interpretation process proposed in this thesis were designed and tested for CityGML 2.0, they can be extended and upgraded for the newer version 3.0 in future development without extensive effort. This is due to three main reasons: (1) the matching process is generic and can already detect all changes in node properties as well as graph structure at its most basic level, (2) both versions 2.0 and 3.0 of CityGML utilize the same GML geometries conforming to the ISO 19100 series, and (3) the rule networks employed during the interpretation process have their own structure independent of CityGML. However, since the implementation of this thesis employs the CityGML classes from the library *citygml4j*, which assigns different naming schemes to object classes between version 2.0 and 3.0, the same city objects that exist in both versions may still belong to different classes. Thus, the necessary step to upgrade the matching and interpreting process for CityGML version 3.0 is to use the correct class names and their respective functions for each version.

Furthermore, while the rule network was initially designed to describe and define rules for identifying patterns among changes, it can also be employed to define rules for examining the consistency and validity of the data provided in CityGML documents, as well as their changes. For instance, a rule can be constructed to determine whether a vertical increase in size of the walls of a building coincides with its new measured height. If these do not coincide, a geometric inconsistency has been detected. This approach can also be leveraged to enable quality control in CityGML documents.

One of the primary objectives in the conceptualization of the rule network is to ensure its adaptability, thereby enabling users to define their own rules for pattern matching. This rule network can be extended to meet the specific requirements of various use cases, either manually on an ad-hoc basis or automatically using grammar-based machine learning (Dehbi & Plümer, 2011). Chapter 7 not only demonstrates how these rules can be defined and applied, but also showcases the techniques for evaluating and interpreting their results. Even in different use cases with different pattern rules, these same techniques can still be applied. Similarly, the stakeholders presented in the semantic layered networks throughout this thesis serve as examples. They illustrate how the relevance relations between any given change and stakeholder can be evaluated. Based on the same concepts, these stakeholders and their corresponding networks can be adapted and extended to accommodate specific use cases.

Due to their structural similarities, the graph representations of CityGML documents employed in this thesis can be further utilized as training datasets for deep learning and other research in artificial intelligence for cities in the future.

Publications

The list of publications produced over the course of this thesis, all of which were peer-reviewed with the author serving as the lead contributor, is presented as follows. All but the last underwent a double-blind process:

1. Nguyen, S. H., & Kolbe, T. H. (2024, September). Identification and Interpretation of Change Patterns in Semantic 3D City Models [18th 3D GeoInfo Conference 2023, Technical University of Munich (TUM), Munich, Germany]. In T. H. Kolbe, A. Donaubauer & C. Beil (Eds.), Recent Advances in 3D Geoinformation Science (pp. 479–496). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43699-4_30.

2. Nguyen, S. H., & Kolbe, T. H. (2022, October). Path-tracing Semantic Networks to Interpret Changes in Semantic 3D City Models [17th International 3D GeoInfo Conference 2022, University of New South Wales (UNSW), Sydney, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 217–224, Vol. X-4/W2-2022). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-x-4-w2-2022-217-2022. ***Best Young Researcher Paper Award***.

3. Nguyen, S. H., & Kolbe, T. H. (2021, October). Modelling Changes, Stakeholders and their Relations in Semantic 3D City Models [16th International 3D GeoInfo Conference 2021, New York University (NYU), NY, USA]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 137–144, Vol. VIII-4/W2-2021). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-viii-4-w2-2021-137-2021. ***Outstanding Paper Award***.

4. Nguyen, S. H., & Kolbe, T. H. (2020, September). A Multi-Perspective Approach to Interpreting Spatio-Semantic Changes of Large 3D City Models in CityGML using a Graph Database [15th International 3D GeoInfo Conference 2020, University College London (UCL), London, UK]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 143–150, Vol. VI-4/W1-2020). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-vi-4-w1-2020-143-2020.

5. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2017, October). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database [12th

International 3D GeoInfo Conference 2017, University of Melbourne, Melbourne, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 99–106, Vol. IV-4/W5). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-iv-4-w5-99-2017.

6. Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2018). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database. In gis.Science (pp. 85–100, Vol. 3). Wichmann Verlag. https://gispoint.de/artikelarchiv/gis/2018/gisscience-ausgabe-32018.html.

The following list contains publications with the author serving as a contributor. While not directly related to the topics discussed in this thesis, they maintain a thematic relevance within the field of city modelling and urban data management:

1. Olbrich, F., Beil, C., Nguyen, S. H., & Kolbe, T. H. (2024, March). Multimodale Navigationsanwendungen für CityGML 3.0-konforme 3D-Straßenraummodelle mittels Graphdatenbanken. In T. P. Kersten & N. Tilly (Eds.), DGPF-Jahrestagung 2024 - Stadt, Land, Fluss - Daten vernetzen, 44. Wissenschaftlich-Technische Jahrestagung der DGPF (pp. 357–369, Vol. 32). Deutsche Gesellschaft für Photogrammetrie, Fernerkundung und Geoinformation (DGPF) e.V.. https://doi.org/10.24407/KXP:1885708890.

2. Chaturvedi, K., Matheus, A., Nguyen, S. H., & Kolbe, T. H. (2019, December). Securing Spatial Data Infrastructures for Distributed Smart City Applications and Services. In Future Generation Computer Systems (pp. 723–736, Vol. 101). Elsevier BV. https://doi.org/10.1016/j.future.2019.07.002.

3. Chaturvedi, K., Matheus, A., Nguyen, S. H., & Kolbe, T. H. (2018, October). Securing Spatial Data Infrastructures in the Context of Smart Cities. In 2018 International Conference on Cyberworlds (CW) (pp. 403–408). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/cw.2018.00078.

The implementation of this thesis is available in the following open-source GitHub repositories:

1. **3DCityKG**: An automatic, high-performance tool to generate knowledge graphs from semantic 3D city models

   URL: https://github.com/tum-gis/3dcitykg

2. **citymodel-compare**: A graph-based tool to detect and interpret changes in large semantic 3D city models

   URL: https://github.com/tum-gis/citymodel-compare

Appendices

A. Assessing the Mapping of *FZK-Haus* and *Railway-Scene* Datasets

Table A.1 presents the complete evaluation of the preservation of thematic and structural content in the generated graph representations of the CityGML datasets *FZK-Hause* (KIT IAI, 2017) and *Railway-Scene* (Häfele & Nagel, 2015). This table describes the mapping patterns for each CityGML element. These descriptions use the following symbols:

1. Node : Create either a single node or a source node of a subgraph
2. Rel : Create an outgoing relationship from the current node
3. Prop : Insert a property to the current node
4. *L* : Use the name from the left column in 'CamelCase' without namespace prefix
5. *l* : Use the name from the left column in 'camelCase' without namespace prefix

To distinguish between elements and attributes that share the same name but belong to different namespaces, the (shortened) package names of their corresponding Java classes in *citygml4j* can be used. For instance, the elements *bldg:GroundSurface* and *tun:GroundSurface* have identical names without the namespace prefixes. Thus, their corresponding nodes in the graph can be labelled as *building.GroundSurface* and *tunnel.GroundSurface*, where *building* and *tunnel* are the (shortened) names of their packages. However, these package names are omitted in the following descriptions for simplicity, if the element and attribute names are unique and not shared by any other elements or attributes. For example:

1. ***core:CityModel*** Node *L*
   Create a node with the label *CityModel*.
2. ***core:cityObjectMember*** Rel *l*
   Create a relationship with the type *cityObjectMember*.
3. ***core:creationDate*** Prop *l*
   Insert a property with the name *creationDate* to the current node.

4. ***bldg:measuredHeight*** **Prop** ***l*** + **Node** ***Length***
   Create a relationship with the type *measuredHeight* and a node with the label *Length*. The new relationship connects the current node with the new node.

5. ***xlink:href*** **Rel** ***object*** $\longrightarrow$ **Node** **found by** ***href***
   Create a relationship with the type *object* and point it to an existing node that has the same identifier as the value of *href* (without '#').

All non-zero cells of Table A.1 are blue. This indicates that the generated graph representations of all CityGML datasets have achieved the value of 100 % for the type, instance, and relationship coverage, as well as the XLink replacement, leading to the total preservation of thematic and structural data of the original CityGML documents.

An excerpt version of this table was shown in Table 3.2.

Table A.1.: Assessing the preservation of thematic and structural content in the generated graph representations of the CityGML datasets *FZK-Haus* (KIT IAI, 2017) and *Railway-Scene* (Häfele & Nagel, 2015). The total number of occurrences of each CityGML element and attribute per dataset is shown in the cells located in the respective rows and columns. Cells with full coverage are shown in blue.

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
CityGML elements in ascending XML levels:								
00	*core:CityModel*	Node L	1	1	1	1	1	1
01	*app:appearanceMember**	Rel l + Node L	0	0	0	0	2	151
01	*core:cityObjectMember**	Rel l	1	1	1	1	1	52
01	*gml:boundedBy*	Rel l	1	1	1	1	1	0
01	*gml:name*[t]	Rel l	1	1	1	1	1	0
02	*app:Appearance*	Node L	0	0	0	0	2	151
02	*bldg:Building*	Node L	1	1	1	1	1	3
02	*brid:Bridge*	Node L	0	0	0	0	0	4
02	*dem:ReliefFeature*	Node L	0	0	0	0	0	1
02	*frn:CityFurniture*	Node L	0	0	0	0	0	11
02	*gen:GenericCityObject*	Node L	0	0	0	0	0	2
02	*gml:Envelope*	Node L	1	1	1	1	1	0
02	*grp:CityObjectGroup*	Node L	0	0	0	0	0	1
02	*tran:Railway*	Node L	0	0	0	0	0	10
02	*tun:Tunnel*	Node L	0	0	0	0	0	4
02	*veg:SolitaryVegetationObject*	Node L	0	0	0	0	0	15
02	*wtr:WaterBody*	Node L	0	0	0	0	0	1
03	*app:surfaceDataMember**	Rel l	0	0	0	0	2	151
03	*app:theme*[t]	Prop l	0	0	0	0	0	151

[t] Contains texts * Multi-instances Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD					Rail-way
			0	1	2	3	4	
03	*bldg:address*	Rel *l*	1	1	1	1	1	0
03	*bldg:boundedBy*	Rel *l*	0	0	7	7	7	40
03	*bldg:class*^{t}	Rel *l* + Node *Code*	1	1	1	1	1	0
03	*bldg:function*^{t,*}	Rel *l* + Node *Code*	1	1	1	1	1	1
03	*bldg:interiorRoom**	Rel *l*	0	0	0	0	7	0
03	*bldg:lod0FootPrint*	Rel *l*	1	0	0	0	0	0
03	*bldg:lod0RoofEdge*	Rel *l*	1	0	0	0	0	0
03	*bldg:lod1Solid*	Rel *l*	0	1	0	0	0	0
03	*bldg:lod2Solid*	Rel *l*	0	0	1	0	0	0
03	*bldg:lod3Solid*	Rel *l*	0	0	0	1	0	0
03	*bldg:lod4Solid*	Rel *l*	0	0	0	0	1	0
03	*bldg:measuredHeight*^{t}	Rel *l* + Node *Length*	1	1	1	1	1	0
03	*bldg:outerBuildingInstallation**	Rel *l*	0	0	0	0	0	56
03	*bldg:roofType*^{t}	Rel *l* + Node *Code*	1	1	1	1	1	0
03	*bldg:storeysAboveGround*^{t}	Prop *l*	1	1	1	1	1	0
03	*bldg:storeysBelowGround*^{t}	Prop *l*	1	1	1	1	1	0
03	*bldg:usage*^{t,*}	Rel *l* + Node *Code*	1	1	1	1	1	0
03	*bldg:yearOfConstruction*^{t}	Rel *l* + Node *LocalDate*	1	1	1	1	1	0
03	*brid:class*^{t}	Rel *l* + Node *Code*	0	0	0	0	0	2
03	*brid:function*^{t,*}	Rel *l* + Node *Code*	0	0	0	0	0	2
03	*brid:lod3MultiSurface*	Rel *l*	0	0	0	0	0	4
03	*brid:outerBridgeConstruction**	Rel *l*	0	0	0	0	0	3
03	*brid:outerBridgeInstallation**	Rel *l*	0	0	0	0	0	2
03	*core:creationDate*^{t}	Prop *l*	1	1	1	1	1	52

^{t} Contains texts * Multi-instances Left value in *l* 'camelCase' or *L* 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
03	*core:relativeToTerrain*^{t}	Rel l + Node L	1	1	1	1	1	46
03	*core:relativeToWater*^{t}	Rel l + Node *RelativeToWater*	0	0	0	0	0	1
03	*dem:lod*^{t}	Prop l	0	0	0	0	0	1
03	*dem:reliefComponent**	Rel l	0	0	0	0	0	1
03	*frn:function*^{t,*}	Rel l + Node *Code*	0	0	0	0	0	11
03	*frn:lod3Geometry*	Rel l	0	0	0	0	0	11
03	*gen:function*^{t,*}	Rel l + Node *Code*	0	0	0	0	0	1
03	*gen:lod3Geometry*	Rel l	0	0	0	0	0	2
03	*gen:measureAttribute**	Rel l + Node L	1	1	1	1	1	0
03	*gen:stringAttribute**	Rel l + Node L	2	2	2	2	2	0
03	*gml:boundedBy*	Rel l	0	0	0	0	0	51
03	*gml:description*^{t}	Rel l + Node *StringOrRef*	1	1	1	1	1	5
03	*gml:lowerCorner*^{t}	Rel l + Node *DirectPosition*	1	1	1	1	1	0
03	*gml:name*^{t,*}	Rel l + Node *Code*	1	1	1	1	1	40
03	*gml:upperCorner*^{t}	Rel l + Node *DirectPosition*	1	1	1	1	1	0
03	*grp:groupMember*^{t,*}	Rel l	0	0	0	0	0	14
03	*tran:function*^{t,*}	Rel l + Node *Code*	0	0	0	0	0	10
03	*tran:lod3MultiSurface*	Rel l	0	0	0	0	0	10
03	*tun:boundedBy*	Rel l	0	0	0	0	0	24
03	*tun:outerTunnelInstallation**	Rel l	0	0	0	0	0	8
03	*veg:class*^{t}	Rel l + Node *Code*	0	0	0	0	0	14
03	*veg:function*^{t,*}	Rel l + Node *Code*	0	0	0	0	0	1
03	*veg:lod3ImplicitRepresentation*	Rel l	0	0	0	0	0	15
03	*veg:species*^{t}	Rel l + Node *Code*	0	0	0	0	0	15

^{t} Contains texts * Multi-instances Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
03	*wtr:boundedBy*	Rel *l*	0	0	0	0	0	2
03	*wtr:class*^{t}	Rel *l* + Node *Code*	0	0	0	0	0	1
04	*app:ParameterizedTexture*	Node *L*	0	0	0	0	0	66
04	*app:X3DMaterial*	Node *L*	0	0	0	0	2	85
04	*bldg:BuildingInstallation*	Node *L*	0	0	0	0	0	56
04	*bldg:GroundSurface*	Node *L*	0	0	1	1	1	3
04	*bldg:OuterCeilingSurface*	Node *L*	0	0	0	0	0	3
04	*bldg:OuterFloorSurface*	Node *L*	0	0	0	0	0	1
04	*bldg:RoofSurface*	Node *L*	0	0	2	2	2	8
04	*bldg:Room*	Node *L*	0	0	0	0	7	0
04	*bldg:WallSurface*	Node *L*	0	0	4	4	4	25
04	*brid:BridgeConstructionElement*	Node *L*	0	0	0	0	0	3
04	*brid:BridgeInstallation*	Node *L*	0	0	0	0	0	2
04	*core:Address**	Rel *l* + Node *L*	1	1	1	1	1	0
04	*core:ImplicitGeometry*	Node *L*	0	0	0	0	0	15
04	*dem:TINRelief*	Node *L*	0	0	0	0	0	1
04	*gen:value*^{t}	Prop *l*	3	3	3	3	3	0
04	*gml:Envelope*	Node *L*	0	0	0	0	0	51
04	*gml:MultiSurface*	Node *L*	2	0	0	0	0	27
04	*gml:Solid*	Node *L*	0	1	1	1	1	0
04	*tun:ClosureSurface*	Node *L*	0	0	0	0	0	8
04	*tun:GroundSurface*	Node *L*	0	0	0	0	0	4
04	*tun:RoofSurface*	Node *L*	0	0	0	0	0	4
04	*tun:TunnelInstallation*	Node *L*	0	0	0	0	0	8

^{t} Contains texts * Multi-instances Left value in *l* 'camelCase' or *L* 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
04	tun:WallSurface	Node L	0	0	0	0	0	8
04	wtr:WaterGroundSurface	Node L	0	0	0	0	0	1
04	wtr:WaterSurface	Node L	0	0	0	0	0	1
05	app:borderColor^{t}	Rel l + Node ColorPlusOpacity	0	0	0	0	0	66
05	app:diffuseColor^{t}	Rel l + Node Color	0	0	0	0	2	85
05	app:emissiveColor^{t}	Rel l + Node Color	0	0	0	0	2	84
05	app:imageURI^{t}	Prop l	0	0	0	0	0	66
05	app:specularColor^{t}	Rel l + Node Color	0	0	0	0	2	85
05	app:target*	Rel l	0	0	0	0	0	13,933
05	app:target^{t,*}	Rel l	0	0	0	0	5	31,808
05	app:textureType^{t}	Rel l + Node L	0	0	0	0	0	66
05	app:transparency^{t}	Prop l	0	0	0	0	2	84
05	app:wrapMode^{t}	Rel l + Node L	0	0	0	0	0	66
05	bldg:boundedBy	Rel l	0	0	0	0	48	0
05	bldg:function^{t,*}	Rel l + Node Code	0	0	0	0	0	56
05	bldg:interiorFurniture*	Rel l	0	0	0	0	15	0
05	bldg:lod2MultiSurface	Rel l	0	0	7	0	0	0
05	bldg:lod3Geometry	Rel l	0	0	0	0	0	56
05	bldg:lod3MultiSurface	Rel l	0	0	0	7	0	40
05	bldg:lod4MultiSurface	Rel l	0	0	0	0	7	0
05	bldg:lod4Solid	Rel l	0	0	0	0	7	0
05	bldg:opening*	Rel l	0	0	0	13	13	43
05	bldg:roomInstallation*	Rel l	0	0	0	0	10	0
05	brid:function^{t,*}	Rel l + Node Code	0	0	0	0	0	4

^{t} Contains texts * Multi-instances   Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
05	*brid:lod3Geometry*	Rel *l*	0	0	0	0	0	5
05	*core:creationDate*[t]	Prop *l*	0	0	0	0	0	136
05	*core:referencePoint*	Rel *l*	0	0	0	0	0	15
05	*core:relativeGMLGeometry*	Rel *l*	0	0	0	0	0	3
05	*core:relativeGMLGeometry*[t]	Rel *l*	0	0	0	0	0	12
05	*core:relativeToTerrain*[t]	Rel *l* + Node *L*	0	0	0	0	0	48
05	*core:relativeToWater*[t]	Rel *l* + Node *RelativeToWater*	0	0	0	0	0	60
05	*core:transformationMatrix*[t]	Rel *l* + Node *TransMatrix4x4*	0	0	0	0	0	15
05	*core:xalAddress*	Rel *l*	1	1	1	1	1	0
05	*dem:lod*[t]	Prop *l*	0	0	0	0	0	1
05	*dem:tin*	Rel *l*	0	0	0	0	0	1
05	*gml:boundedBy*	Rel *l*	0	0	0	0	0	136
05	*gml:description*[t]	Rel *l* + Node *StringOrRef*	0	0	1	1	8	0
05	*gml:exterior*	Rel *l*	0	1	1	1	1	0
05	*gml:lowerCorner*[t]	Rel *l* + Node *DirectPosition*	0	0	0	0	0	51
05	*gml:name*[t,*]	Rel *l* + Node *Code*	0	0	7	7	14	101
05	*gml:surfaceMember*[*]	Rel *l*	2	0	0	0	0	38,666
05	*gml:upperCorner*[t]	Rel *l* + Node *DirectPosition*	0	0	0	0	0	51
05	*tun:function*[t,*]	Rel *l* + Node *Code*	0	0	0	0	0	8
05	*tun:lod3Geometry*	Rel *l*	0	0	0	0	0	8
05	*tun:lod3MultiSurface*	Rel *l*	0	0	0	0	0	24
05	*wtr:lod3Surface*	Rel *l*	0	0	0	0	0	2
06	*app:TexCoordList*	Node *L*	0	0	0	0	0	13,933
06	*bldg:BuildingFurniture*	Node *L*	0	0	0	0	15	0

[t] Contains texts   [*] Multi-instances   Left value in *l* 'camelCase' or *L* 'CamelCase'   *Continued on next page*

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
06	*bldg:CeilingSurface*	Node *L*	0	0	0	0	8	0
06	*bldg:ClosureSurface*	Node *L*	0	0	0	0	6	0
06	*bldg:Door*	Node *L*	0	0	0	2	2	9
06	*bldg:FloorSurface*	Node *L*	0	0	0	0	7	0
06	*bldg:IntBuildingInstallation*	Node *L*	0	0	0	0	10	0
06	*bldg:InteriorWallSurface*	Node *L*	0	0	0	0	27	0
06	*bldg:Window*	Node *L*	0	0	0	11	11	34
06	*gml:CompositeSurface*	Node *L*	0	1	1	1	1	2
06	*gml:Envelope*	Node *L*	0	0	0	0	0	136
06	*gml:MultiSurface*	Node *L*	0	0	7	7	7	136
06	*gml:Point*	Node *L*	0	0	0	0	0	15
06	*gml:Polygon*	Node *L*	2	0	0	0	0	38,666
06	*gml:Solid*	Node *L*	0	0	0	0	7	0
06	*gml:TriangulatedSurface*	Node *L*	0	0	0	0	0	1
06	*xAL:AddressDetails*	Node *L*	1	1	1	1	1	0
07	*app:textureCoordinates*^{t,*}	Rel *l* + Node *L*	0	0	0	0	0	14,106
07	*bldg:function*^{t,ast}	Rel *l* + Node *Code*	0	0	0	0	25	0
07	*bldg:lod3MultiSurface*	Rel *l*	0	0	0	13	0	43
07	*bldg:lod4Geometry*	Rel *l*	0	0	0	0	25	0
07	*bldg:lod4MultiSurface*	Rel *l*	0	0	0	0	61	0
07	*bldg:opening**	Rel *l*	0	0	0	0	19	0
07	*core:creationDate*^{t}	Prop *l*	0	0	0	0	0	43
07	*core:relativeToTerrain*^{t}	Rel *l* + Node *L*	0	0	0	0	0	37
07	*core:relativeToWater*^{t}	Rel *l* + Node *RelativeToWater*	0	0	0	0	0	5

^{t} Contains texts * Multi-instances Left value in *l* 'camelCase' or *L* 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
07	*gml:boundedBy*	Rel *l*	0	0	0	0	0	43
07	*gml:description*^{t}	Rel *l* + Node *StringOrRef*	0	0	0	0	25	0
07	*gml:exterior*	Rel *l*	2	0	0	0	7	38,666
07	*gml:interior**	Rel *l*	0	0	0	0	0	153
07	*gml:lowerCorner*^{t}	Rel *l* + Node *DirectPosition*	0	0	0	0	0	136
07	*gml:name*^{t,*}	Rel *l* + Node *Code*	0	0	0	13	85	17
07	*gml:pos*^{t,*}	Rel *l* + Node *DirectPosition*	0	0	0	0	0	15
07	*gml:surfaceMember*^{t,*}	Rel *l*	0	6	14	29	29	4,692
07	*gml:trianglePatches*	Rel *l*	0	0	0	0	0	1
07	*gml:upperCorner*^{t}	Rel *l* + Node *DirectPosition*	0	0	0	0	0	136
07	*xAL:Locality*	Rel *l* + Node *L*	1	1	1	1	1	0
08	*bldg:Door*	Node *L*	0	0	0	0	8	0
08	*bldg:Window*	Node *L*	0	0	0	0	11	0
08	*gml:CompositeSurface*	Node *L*	0	0	0	4	11	0
08	*gml:Envelope*	Node *L*	0	0	0	0	0	43
08	*gml:LinearRing*	Node *L*	2	0	0	0	0	38,819
08	*gml:MultiSurface*	Node *L*	0	0	0	13	86	43
08	*gml:Polygon*	Node *L*	0	6	7	5	5	4,692
08	*gml:Triangle*	Node *L*	0	0	0	0	0	22,022
08	*xAL:LocalityName*^{t,*}	Rel *l* + Node *L*	1	1	1	1	1	0
08	*xAL:PostalCode*	Rel *l* + Node *L*	1	1	1	1	1	0
08	*xAL:Thoroughfare*	Rel *l* + Node *L*	1	1	1	1	1	0
09	*bldg:lod4MultiSurface*	Rel *l*	0	0	0	0	19	0
09	*gml:exterior*	Rel *l*	0	6	7	5	5	26,714

^{t} Contains texts * Multi-instances Left value in *l* 'camelCase' or *L* 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Railway
09	*gml:interior**	Rel l	0	0	0	0	0	56
09	*gml:lowerCorner*^{t}	Rel l + Node *DirectPosition*	0	0	0	0	0	43
09	*gml:name*^{t,*}	Rel l + Node *Code*	0	0	0	0	15	0
09	*gml:posList*^{t}	Rel l + Node *DirectPositionList*	2	0	0	0	0	38,819
09	*gml:surfaceMember**	Rel l	0	0	0	205	14,511	167
09	*gml:upperCorner*^{t}	Rel l + Node *DirectPosition*	0	0	0	0	0	43
09	*xAL:PostalCodeNumber*^{t,*}	Rel l + Node L	1	1	1	1	1	0
09	*xAL:ThoroughfareName*^{t,*}	Rel l + Node L	1	1	1	1	1	0
09	*xAL:ThoroughfareNumber*^{t,*}	Rel *numberOrRange* + Node L	1	1	1	1	1	0
10	*gml:CompositeSurface*	Node L	0	0	0	0	16	0
10	*gml:LinearRing*	Node L	0	6	7	5	5	26,770
10	*gml:MultiSurface*	Node L	0	0	0	0	19	0
10	*gml:OrientableSurface*	Node L	0	0	0	0	70	0
10	*gml:Polygon*	Node L	0	0	0	205	14,425	167
11	*gml:baseSurface*^{t}	Rel l	0	0	0	0	70	0
11	*gml:exterior*	Rel l	0	0	0	205	14,425	167
11	*gml:interior**	Rel l	0	0	0	13	36	24
11	*gml:pos*^{t,*}	Rel l + Node *DirectPosition*	0	0	37	33	33	0
11	*gml:posList*^{t}	Rel l + Node *DirectPositionList*	0	6	0	0	0	26,770
11	*gml:surfaceMember**	Rel l	0	0	0	0	232	0
12	*gml:LinearRing*	Node L	0	0	0	218	14,461	191
12	*gml:OrientableSurface*	Node L	0	0	0	0	16	0
12	*gml:Polygon*	Node L	0	0	0	0	216	0
13	*gml:baseSurface*^{t}	Rel l	0	0	0	0	16	0

^{t} Contains texts * Multi-instances Left value in l 'camelCase' or L 'CamelCase'

Continued on next page

Assessing preservation of thematic and structural content in generated graphs (continued)

Lvl.	Name of CityGML Element or Attribute	Corresponding Node Label, Rel Type, or Prop Name	FZK-Haus in LOD 0	1	2	3	4	Rail-way
13	*gml:exterior*	Rel l	0	0	0	0	216	0
13	*gml:interior**	Rel l	0	0	0	0	11	0
13	*gml:pos*^{t,*}	Rel l + Node *DirectPosition*	0	0	0	1,364	70,420	0
13	*gml:posList*^{t}	Rel l + Node *DirectPositionList*	0	0	0	0	0	191
14	*gml:LinearRing*	Node L	0	0	0	0	227	0
15	*gml:pos*^{t,*}	Rel l + Node *DirectPosition*	0	0	0	0	1,302	0
CityGML attributes:								
	codeSpace	Prop l	4	4	4	4	4	0
	gml:id	Prop l	1	1	22	458	29,383	110,047
	name	Prop l	3	3	3	3	3	0
	orientation	Rel l + Node *Sign*	0	0	0	0	86	0
	ring	Prop l	0	0	0	0	0	14,106
	srsDimension	Prop l	5	9	3	3	3	66,025
	srsName	Prop l	1	1	1	1	1	230
	Type	Prop L	2	2	2	2	2	0
	uom	Prop l	2	2	2	2	2	0
	uri	Prop l	0	0	0	0	0	13,933
	xlink:href	Rel *object* → Node found by *href*	0	0	7	20	106	26
Total preservation of thematic and structural content			☑	☑	☑	☑	☑	☑

CityGML module prefixes: *Core* (*core*), *Appearance* (*app*), *Building* (*bldg*), *Bridge* (*brid*), *Relief* (*dem*), *CityFurniture* (*frn*), *Generics* (*gen*), *CityObjectGroup* (*grp*), *LandUse* (*luse*), *TexturedSurface* (*tex*) (deprecated), *Transportation* (*tran*), *Tunnel* (*tun*), *Vegetation* (*veg*), and *WaterBody* (*wtr*) (Gröger et al., 2012). XML specification prefixes: GML (*gml*) (Cox et al., 2004), xAL (*xAL*) (CIQ TC, 2002), and XLink (*xlink*) (W3C, 2006).

Node Represented as a node Rel Represented as a relationship Prop Stored as a node property

l Value of the left column in 'camelCase' (without namespace) L Value of the left column in 'CamelCase' (without namespace)

^{t} Element contains text content * Multiple instances may exist

B. List of Employed Rules in Cypher for Detecting Change Patterns

Listing B.1 provides a full compilation of all change pattern rules in Cypher employed in the implementation of this thesis. The definitions of these rules can be found in Section 5.3. The rule network containing these patterns was visualized in Figure 7.18. General guidelines on the construction of such rule networks were provided in Sections 7.1.3 and 7.5.1. Excerpts of this network were shown throughout Chapter 7.

```
// Pattern in id of buildings
MERGE (updated_property:RULE {
  change_type: 'UpdatedProperty'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  search_length: 0,
  conditions: 'NAME === "id"',
  propagate: 'NAME',
  weight: 1
}]->(updated_building_id:RULE {
  change_type: 'UpdatedBuildingId',
  calc_scope: 'NAME'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_updated_building_ids:RULE {
  change_type: 'GlobalUpdatedBuildingIds',
})

// Pattern in creationDate of buildings
MERGE (updated_property)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  search_length: 0,
  conditions: 'NAME === "creationDate"',
  propagate: 'RIGHT_VALUE',
  weight: 1
}]->(updated_building_creation_date:RULE {
  change_type: 'UpdatedBuildingCreationDate',
  calc_scope: 'RIGHT_VALUE'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global',
  propagate: 'RIGHT_VALUE'
}]->(global_updated_building_creation_dates:RULE {
  change_type: 'GlobalUpdatedBuildingCreationDates'
})

// Pattern in measuredHeight of buildings
MERGE (updated_property)-[:AGGREGATED_TO {
  next_content_type: 'Length',
  search_length: 1,
  conditions: 'NAME === "value"',
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: 1
}]->(updated_measured_height:RULE {
  change_type: 'UpdatedMeasuredHeight'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  search_length: 1,
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: 1
}]->(updated_building_measured_height:RULE {
  change_type: 'UpdatedBuildingMeasuredHeight',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_updated_building_measured_heights:RULE {
  change_type: 'GlobalUpdatedBuildingMeasuredHeights'
})

// Pattern for translated surfaces
MERGE (translated_polygon:RULE {
  change_type: 'TranslatedPolygon'
```

```
})-[:AGGREGATED_TO {
  next_content_type: 'SurfaceProperty',
  search_length: 1,
  propagate: 'x;y;z',
  weight: 1
}]->(translated_surface:RULE {
  change_type: 'TranslatedSurface'
})

// Pattern for translated roofs
MERGE (translated_surface)-[:AGGREGATED_TO {
  next_content_type: 'RoofSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_roof:RULE {
  change_type: 'TranslatedRoof'
})

// Pattern for translated walls
MERGE (translated_surface)-[:AGGREGATED_TO {
  next_content_type: 'WallSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_wall:RULE {
  change_type: 'TranslatedWall'
})

// Pattern for translated grounds
MERGE (translated_surface)-[:AGGREGATED_TO {
  next_content_type: 'GroundSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_ground:RULE {
  change_type: 'TranslatedGround'
})

// Pattern for translated roofs of building parts
MERGE (translated_roof)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_part_roofs:RULE {
  change_type: 'TranslatedBuildingPartRoofs'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_roofs_with_bparts {
  change_type: 'TranslatedBuildingRoofs_WithBParts'
})

// Pattern for translated walls of building parts
MERGE (translated_wall)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_part_walls:RULE {
  change_type: 'TranslatedBuildingPartWalls'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_walls_with_bparts {
  change_type: 'TranslatedBuildingWalls_WithBParts'
})

// Pattern for translated grounds of building parts
MERGE (translated_ground)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_part_grounds:RULE {
  change_type: 'TranslatedBuildingPartGrounds'
```

```
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_grounds_with_bparts {
  change_type: 'TranslatedBuildingGrounds_WithBParts'
})

// Pattern for translated building parts
MERGE (translated_building_part_roofs)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_translated_roofs',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_part:RULE {
  change_type: 'TranslatedBuildingPart',
  join: 'approxEquals(rule_translated_roofs.x, rule_translated_walls.x)
      && approxEquals(rule_translated_roofs.y, rule_translated_walls.y)
      && approxEquals(rule_translated_roofs.z, rule_translated_walls.z)
      && approxEquals(rule_translated_roofs.x, rule_translated_grounds.x)
      && approxEquals(rule_translated_roofs.y, rule_translated_grounds.y)
      && approxEquals(rule_translated_roofs.z, rule_translated_grounds.z)'
})
MERGE (translated_building_part_walls)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_translated_walls',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_part)
MERGE (translated_building_part_grounds)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_translated_grounds',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_part)

// Pattern for translated buildings (with building parts)
MERGE (translated_building_part)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_with_bparts:RULE {
  change_type: 'TranslatedBuilding_WithBParts',
  calc_scope: 'x;y;z'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global',
  propagate: 'x;y;z'
}]->(global_translated_buildings_with_bparts:RULE {
  change_type: 'GlobalTranslatedBuildings_WithBParts'
})

// Pattern for translated surfaces of buildings (without building parts)
MERGE (translated_roof)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_roofs_no_bparts:RULE {
  change_type: 'TranslatedBuildingRoofs_NoBParts'
})
MERGE (translated_wall)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_walls_no_bparts:RULE {
  change_type: 'TranslatedBuildingWalls_NoBParts'
})
MERGE (translated_ground)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  propagate: 'x;y;z',
  weight: '*'
}]->(translated_building_grounds_no_bparts:RULE {
  change_type: 'TranslatedBuildingGrounds_NoBParts'
})

// Pattern for translated buildings (without building parts)
MERGE (translated_building_roofs_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_roofs',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_no_bparts:RULE {
  change_type: 'TranslatedBuilding_NoBParts',
  join: 'approxEquals(rule_translated_roofs.x, rule_translated_walls.x)
      && approxEquals(rule_translated_roofs.y, rule_translated_walls.y)
      && approxEquals(rule_translated_roofs.z, rule_translated_walls.z)
      && approxEquals(rule_translated_roofs.x, rule_translated_grounds.x)
      && approxEquals(rule_translated_roofs.y, rule_translated_grounds.y)
      && approxEquals(rule_translated_roofs.z, rule_translated_grounds.z)',
  calc_scope: 'x;y;z'
})
MERGE (translated_building_walls_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_walls',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_no_bparts)
MERGE (translated_building_grounds_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_grounds',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building_no_bparts)
MERGE (translated_building_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global',
  propagate: 'x;y;z'
}]->(global_translated_buildings_no_bparts:RULE {
  change_type: 'GlobalTranslatedBuildings_NoBParts'
```

```
})

// Pattern for translated buildings (with or without building parts)
MERGE (translated_building_with_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_with_bparts',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building:RULE {
  change_type: 'TranslatedBuildings',
  join: 'rule_with_bparts || rule_no_bparts',
  calc_scope: 'x;y;z'
})
MERGE (translated_building_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_no_bparts',
  propagate: 'x;y;z',
  weight: 1
}]->(translated_building)
MERGE (translated_building)-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global',
  propagate: 'x;y;z'
}]->(global_translated_buildings:RULE {
  change_type: 'GlobalTranslatedBuildings'
})

// Pattern for resized polygons
MERGE (:RULE {
  change_type: 'ResizedPolygon'
})-[:AGGREGATED_TO {
  next_content_type: 'SurfaceProperty',
  propagate: 'x;y;z',
  weight: 1
}]->(resized_surface:RULE {
  change_type: 'ResizedSurface'
})

// Pattern for resized surfaces
```

```
MERGE (resized_surface)-[:AGGREGATED_TO {
  next_content_type: 'RoofSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(resized_roof:RULE {
  change_type: 'ResizedRoof'
})
MERGE (resized_surface)-[:AGGREGATED_TO {
  next_content_type: 'WallSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(resized_wall:RULE {
  change_type: 'ResizedWall'
})
MERGE (resized_surface)-[:AGGREGATED_TO {
  next_content_type: 'GroundSurface',
  propagate: 'x;y;z',
  weight: 1
}]->(resized_ground:RULE {
  change_type: 'ResizedGround'
})

// Pattern for resized roofs of building parts
MERGE (resized_roof)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_part_roofs:RULE {
  change_type: 'ResizedBuildingPartRoofs'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_roofs_with_bparts:RULE {
  change_type: 'ResizedBuildingRoofs_WithBParts',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
```

```
}]->(global_resized_building_roofs_with_bparts:RULE {
  change_type: 'GlobalResizedBuildingRoofs_WithBParts'
})

// Pattern for resized walls of building parts
MERGE (resized_wall)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_part_walls:RULE {
  change_type: 'ResizedBuildingPartWalls'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_walls_with_bparts:RULE {
  change_type: 'ResizedBuildingWalls_WithBParts',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_resized_building_walls_with_bparts:RULE {
  change_type: 'GlobalResizedBuildingWalls_WithBParts'
})

// Pattern for resized grounds of building parts
MERGE (resized_ground)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_part_grounds:RULE {
  change_type: 'ResizedBuildingPartGrounds'
})-[:AGGREGATED_TO {
  next_content_type: 'Building',
  conditions: 'x;y;z',
  weight: '*'
}]->(size_changed_building_grounds_with_bparts:RULE {
  change_type: 'ResizedBuildingGrounds_WithBParts',
  calc_scope: '*'
```

```
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_resized_building_grounds_with_bparts:RULE {
  change_type: 'GlobalResizedBuildingGrounds_WithBParts'
})

// Pattern for resized roofs of buildings (without building parts)
MERGE (resized_roof)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_roofs_no_bparts:RULE {
  change_type: 'ResizedBuildingRoofs_NoBParts',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_size_changed_buildings_roofs_no_bparts:RULE {
  change_type: 'GlobalResizedBuildingRoofs_NoBParts'
})

// Pattern for resized walls of buildings (without building parts)
MERGE (resized_wall)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_walls_no_bparts:RULE {
  change_type: 'ResizedBuildingWalls_NoBParts',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_resized_buildings_walls_no_bparts:RULE {
  change_type: 'GlobalResizedBuildingWalls_NoBParts',
})

```

```
// Pattern for resized grounds of buildings (without building parts)
MERGE (resized_ground)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  not_contains: 'BuildingPart',
  conditions: 'x;y;z',
  weight: '*'
}]->(resized_building_grounds_no_bparts:RULE {
  change_type: 'ResizedBuildingGrounds_NoBParts',
  calc_scope: '*'
})-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_size_changed_building_grounds_no_bparts:RULE {
  change_type: 'GlobalResizedBuildingGrounds_NoBParts',
})

// Pattern for raised roofs of buildings (without building parts)
MERGE (updated_building_measured_height)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_height',
  conditions: 'RIGHT_VALUE - LEFT_VALUE > 0',
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: '1'
}]->(raised_building_roofs:RULE {
  change_type: 'RaisedBuildingRoofs',
  join: 'approxEquals(rule_size_changed_walls.z,
            rule_height.RIGHT_VALUE - rule_height.LEFT_VALUE)
      && approxEquals(rule_translated_roofs.z,
            rule_size_changed_walls.z + rule_translated_grounds.z)',
  calc_scope: '*'
})
MERGE (translated_building_roofs_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_roofs',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs)
MERGE (translated_building_grounds_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_translated_grounds',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs)
MERGE (resized_building_walls_no_bparts)-[:AGGREGATED_TO {
  next_content_type: 'Building',
  name: 'rule_resized_walls',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_roofs)-[:AGGREGATED_TO {
  next_content_type: 'CityModel',
  scope: 'global'
}]->(global_raised_building_roofs:RULE {
  change_type: 'GlobalRaisedBuildingRoofs'
})

// Pattern for raised roofs of building parts
MERGE (updated_measured_height)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  search_length: 1,
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: 1
}]->(updated_building_part_measured_height:RULE {
  change_type: 'UpdatedBuildingPartMeasuredHeight'
})-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_height_bp',
  conditions: 'RIGHT_VALUE - LEFT_VALUE > 0',
  propagate: 'LEFT_VALUE;RIGHT_VALUE',
  weight: '1'
}]->(raised_building_part_roofs:RULE {
  change_type: 'RaisedBuildingPartRoofs',
  join: 'approxEquals(rule_resized_walls_bp.z,
            rule_height_bp.RIGHT_VALUE - rule_height_bp.LEFT_VALUE)
      && approxEquals(rule_translated_roofs_bp.z,
            rule_resized_walls_bp.z + rule_translated_grounds_bp.z)'
```

```
})
MERGE (translated_building_part_roofs)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_translated_roofs_bp',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_part_roofs)
MERGE (translated_building_part_grounds)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_translated_grounds_bp',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_part_roofs)
MERGE (resized_building_part_walls)-[:AGGREGATED_TO {
  next_content_type: 'BuildingPart',
  name: 'rule_resized_walls_bp',
  conditions: 'approxEquals(x, 0) && approxEquals(y, 0)',
  propagate: 'x;y;z',
  weight: '1'
}]->(raised_building_part_roofs)
```

Listing B.1: Rules for detecting change patterns in Cypher employed in the implementation of this thesis.

C. Cypher Queries for Constructing a Change-Stakeholder Network

Listing C.1 provides all the Cypher queries employed to construct a network representation of stakeholders, changes, and their semantic relations, as introduced in Section 5.5 and visualized in Figures 7.33 and 7.34. Each node and relationship is assigned with a normalized weight ranging between 0 and 1. All weights can be adjusted based on individual use cases. Moreover, the path-tracing direction in this example is set from the Stakeholder Layer L_4 to the Change Type Layer L_1. However, the same techniques can be applied for the opposite direction.

```
// Change Type Layer (L1)
MERGE (identifier_changed:CHANGE_TYPE:CSN {
  name: "Identifier Changed", weight: 0.4})
MERGE (date_changed:CHANGE_TYPE:CSN {
  name: "Date Changed", weight: 0.6})
MERGE (building_unchanged:CHANGE_TYPE:CSN {
  name: "Building Unchanged", weight: 0.8})
MERGE (function_changed:CHANGE_TYPE:CSN {
  name: "Function Changed", weight: 0.7})
MERGE (roof_raised:CHANGE_TYPE:CSN {
  name: "Roof Raised", weight: 0.9})

// Reasoning Layer (L2)
MERGE (batch_update:REASONING:CSN {
  name: "Batch Update", weight: 0.4})
MERGE (data_improvement:REASONING:CSN {
  name: "Data Improvement", weight: 0.5})
MERGE (historic_preservation:REASONING:CSN {
  name: "Historic Preservation", weight: 0.7})
MERGE (building_repurpose:REASONING:CSN {
  name: "Building Repurpose", weight: 0.9})
MERGE (building_renovation:REASONING:CSN {
  name: "Building Renovation", weight: 0.8})
```

```
MERGE (floor_addition:REASONING:CSN {
  name: "Floor Addition", weight: 0.8})

// Actor Role Layer (L3)
MERGE (data_broker:ACTOR_ROLE:CSN {
  name: "Data Broker", weight: 0.7})
MERGE (city_model_manager:ACTOR_ROLE:CSN {
  name: "City Model Manager", weight: 0.8})
MERGE (urban_planner:ACTOR_ROLE:CSN {
  name: "Urban Planner", weight: 0.6})
MERGE (city_mayor:ACTOR_ROLE:CSN {
  name: "City Mayor", weight: 0.9})
MERGE (real_estate_appraiser:ACTOR_ROLE:CSN {
  name: "Real Estate Appraiser", weight: 0.4})
MERGE (fire_safety_officer:ACTOR_ROLE:CSN {
  name: "Fire Safety Officer", weight: 0.5})

// Stakeholder Layer (L4)
MERGE (technology_company:STAKEHOLDER:CSN {
  name: "Technology Company", weight: 0.7})
MERGE (mapping_agency:STAKEHOLDER:CSN {
  name: "Mapping Agency", weight: 0.8})
MERGE (city_planning_department:STAKEHOLDER:CSN {
  name: "City Planning Department", weight: 0.9})
MERGE (real_estate_company:STAKEHOLDER:CSN {
  name: "Real Estate Company", weight: 0.6})
MERGE (fire_and_rescue_department:STAKEHOLDER:CSN {
  name: "Fire and Rescue Department", weight: 0.5})

// Relationships from L4 to L3
MERGE (technology_company)-[:BACKWARD {
  weight: 0.7}]->(data_broker)
MERGE (mapping_agency)-[:BACKWARD {
  weight: 0.8}]->(city_model_manager)
MERGE (city_planning_department)-[:BACKWARD {
  weight: 0.7}]->(urban_planner)
MERGE (city_planning_department)-[:BACKWARD {
  weight: 0.9}]->(city_mayor)
MERGE (real_estate_company)-[:BACKWARD {
  weight: 0.9}]->(real_estate_appraiser)
MERGE (fire_and_rescue_department)-[:BACKWARD {
  weight: 0.6}]->(fire_safety_officer)

// Relationships from L3 to L2
MERGE (data_broker)-[:BACKWARD {
  weight: 0.5}]->(batch_update)
MERGE (data_broker)-[:BACKWARD {
  weight: 0.7}]->(data_improvement)
MERGE (city_model_manager)-[:BACKWARD {
  weight: 0.6}]->(data_improvement)
MERGE (urban_planner)-[:BACKWARD {
  weight: 0.8}]->(historic_preservation)
MERGE (city_mayor)-[:BACKWARD {
  weight: 0.8}]->(historic_preservation)
MERGE (city_mayor)-[:BACKWARD {
  weight: 0.9}]->(building_repurpose)
MERGE (city_mayor)-[:BACKWARD {
  weight: 0.6}]->(building_renovation)
MERGE (city_mayor)-[:BACKWARD {
  weight: 0.7}]->(floor_addition)
MERGE (real_estate_appraiser)-[:BACKWARD {
  weight: 0.2}]->(historic_preservation)
MERGE (real_estate_appraiser)-[:BACKWARD {
  weight: 0.4}]->(building_repurpose)
MERGE (real_estate_appraiser)-[:BACKWARD {
  weight: 0.8}]->(building_renovation)
MERGE (real_estate_appraiser)-[:BACKWARD {
  weight: 0.6}]->(floor_addition)
MERGE (fire_safety_officer)-[:BACKWARD {
  weight: 0.9}]->(building_renovation)
MERGE (fire_safety_officer)-[:BACKWARD {
  weight: 0.7}]->(floor_addition)

// Relationships from L2 to L1
MERGE (batch_update)-[:BACKWARD {
  weight: 0.6}]->(identifier_changed)
MERGE (batch_update)-[:BACKWARD {
  weight: 0.9}]->(date_changed)
```

```
MERGE (data_improvement)-[:BACKWARD {
  weight: 0.2}]->(identifier_changed)
MERGE (data_improvement)-[:BACKWARD {
  weight: 0.4}]->(date_changed)
MERGE (historic_preservation)-[:BACKWARD {
  weight: 1.0}]->(building_unchanged)
MERGE (building_repurpose)-[:BACKWARD {
  weight: 1.0}]->(function_changed)
MERGE (building_renovation)-[:BACKWARD {
  weight: 0.3}]->(building_unchanged)
MERGE (building_renovation)-[:BACKWARD {
  weight: 0.6}]->(function_changed)
MERGE (building_renovation)-[:BACKWARD {
  weight: 0.8}]->(roof_raised)
MERGE (floor_addition)-[:BACKWARD {
  weight: 0.8}]->(roof_raised)
```

Listing C.1: Cypher queries for constructing a network representing stakeholders, changes, and their semantic relations.

Glossary

citygml4j An open-source Java API for CityGML. 37–39, 52, 53, 62, 71, 72, 81, 84, 88, 89, 105, 191, 193, 213, 296, 300, 301, 305, 341

CityJSON A JSON-based encoding for a subset of the CityGML data model. 40, 213, 295

Cypher Neo4j's declarative query language. 88, 164, 165, 194–196, 198, 199, 212, 214, 218, 221, 246–249, 259, 263–265, 280, 284, 286–288, 317, 330, 331, 334, 343, 345, 346, 348, 349

Docker A tool to automate and deploy applications in containers. 212, 214, 221

GeoSPARQL A Geographic Query Language for RDF Data. 46

IndoorGML An OGC Standard for Indoor Spatial Information. 46, 89, 300

nauty A program for computing automorphism groups of graphs and digraphs. 97, 100

Neo4j A graph database management system developed by Neo4j, Inc. 43, 44, 46–49, 51, 52, 56, 60, 62, 77, 79, 83, 86–88, 99, 100, 103, 106, 114, 123, 164, 183, 189, 192, 194, 197–201, 203, 205–210, 212, 213, 216, 218, 221, 270, 294–296, 298, 341, 346, 348

Neo4j Browser An interface for executing Cypher queries and visualizing results. 65, 68–70, 221, 247, 249, 254, 265, 266, 273, 284, 285, 341, 343–345

NoSQL Non-relational, or non-SQL, or sometimes Not Only SQL databases. 43

Oracle A proprietary multi-model database management system. 40

PostGIS A free and open-source extension for PostgreSQL that adds support for geospatial data. 40

PostgreSQL A powerful, free, and open-source object-relational database management system. 40

R-tree A tree data structure for efficient spatial indexing. 87, 122, 123, 200–204, 213, 216, 225, 294, 296, 299, 343

Traces A program for computing automorphism groups of graphs. 97, 100

Acronyms

3DCityDB 3D City Database. 2, 40, 43, 46, 67, 90, 94, 213, 270, 277, 289, 300, 341

AABB Axis-aligned Minimum Bounding Box. 114, 117, 120, 201, 203, 216

ACID Atomicity, Consistency, Isolation, Durability. 43, 205, 207, 210

ADBMS Active Database Management System. 140

ADE Application Domain Extension. 37, 40, 46, 144

ADT Abstract Data Type. 42, 92

AdV Working Committee of the Surveying Authorities of the Laender of the Federal Republic of Germany. 259

ALKIS Amtliches Liegenschaftskatasterinformationssystem. 14, 15, 223, 300

ANN Artificial Neural Network. 177

API Application Programming Interface. 37, 191, 212, 213, 296

B-Rep Boundary Representation. 32

BIM Building Information Modelling. 1, 8, 47, 93, 145, 300

BSP Binary Space Partitioning. 215

BVH Bounding Volume Hierarchy. 200

CAD Computer-aided Design. 42

CAM Computer-aided Manufacturing. 42

CityGML City Geography Markup Language. 2, 8–15, 17–19, 21–24, 26–65, 67–81, 83–95, 98–114, 116, 117, 120, 121, 123–126, 128–137, 139, 143–149, 152–154, 156, 161, 162, 187, 190–193, 199, 203, 205, 206, 211–215, 217, 222, 223, 225, 226, 228, 232, 244, 247, 253, 275, 289, 291–296, 298–301, 305–316, 341–343, 346–348

CPU Central Processing Unit. 222

CRS Coordinate Reference System. 2, 80, 148, 200

DAG Directed Acyclic Graph. 41, 44, 86

DOM Document Object Model. 190, 191

DTD Document Type Definition. 38

ECA Event Condition Action. 24, 140

EDA Event-driven Architecture. 140

FIFO First In First Out. 166, 187

GCS Geographic Coordinate System. 113

GIS Geographic Information System. 7, 15, 42, 93, 96, 253, 298

GML Geography Markup Language. 10–12, 14, 15, 22, 23, 27, 30, 32, 38, 40, 43, 46, 47, 60, 62, 63, 67, 71, 76, 80, 91, 92, 99, 100, 109, 110, 114, 116, 133, 147, 300, 301, 316

GUI Graphical User Interface. 221, 343

HTML HyperText Markup Language. 190

IFC Industry Foundation Classes. 8, 47, 93, 300

IoT Internet of Things. 3

ISO International Organization for Standardization. 15, 30, 147, 300, 301

JAXB Jakarta XML Binding. 36, 37, 191, 193

JAXP Java API for XML Processing. 191, 193

JSON JavaScript Object Notation. 40, 44, 46, 49, 213, 219, 220, 348

JTS JTS Topology Suite. 216

LCA Lowest Common Ancestor. 169, 170

LOD Level of Detail. 10, 11, 18, 21, 31, 33–35, 47, 65–71, 73–76, 81, 82, 85, 86, 88, 128, 145, 149, 150, 187, 222, 292, 295, 307–316, 341, 343, 348

MDA Model-driven Architecture. 147

OGC Open Geospatial Consortium. 9, 30, 44, 46

OSM OpenStreetMap. 46, 89, 229, 230, 236–239, 242, 250, 252, 256–258, 272

OWL Web Ontology Language. 46

PCS Projected Coordinate System. 113

RAM Random-access Memory. 222

RDBMS Relational Database Management System. 38, 44, 295

RDF Resource Description Framework. 46

SAX Simple API for XML. 191

SQL Structured Query Language. 38, 43, 46, 300

SRDBMS Spatially-enhanced Relational Database Management System. 77

StAX Streaming API for XML. 191

UDT Urban Digital Twin. 3, 4, 6, 175, 180, 292, 300, 341

UML Unified Modelling Language. 10, 31, 32, 46, 125, 127, 145–147, 341, 342

uom unit of measurement. 104, 151

URI Uniform Resource Identifier. 112

UTC Coordinated Universal Time. 105

WGS World Geodetic System. 200

WKB Well-known Binary. 77

WKT Well-known Text. 77

xAL Extensible Address Language. 67, 76, 316

XLink XML Linking Language. 10–13, 15, 22, 33, 34, 41, 44, 45, 50, 52, 55, 59–63, 65, 67, 70, 72, 76, 80, 83, 86, 88, 90, 92, 130, 149, 192, 193, 197, 211, 291–293, 306, 316, 341, 343

XML Extensible Markup Language. 10, 11, 23, 30, 33, 36–38, 40, 43, 44, 46, 59, 63, 65, 67, 73, 76, 91, 92, 105, 133, 190, 191, 300, 307, 316, 346

XSD XML Schema Definition. 10, 36–38, 46, 144

XSLT Extensible Stylesheet Language Transformations. 38

List of Figures

List of Tables

List of Code Listings

Bibliography

Agoub, A., Kunde, F., & Kada, M. (2016). Potential of Graph Databases in Representing and Enriching Standardized Geodata. In T. P. Kersten (Ed.), Dreiländertagung der SGPF, DGPF und OVG - Lösungen für eine Welt im Wandel, 36. Wissenschaftlich-Technische Jahrestagung der DGPF (pp. 208–216, Vol. 25). Deutsche Gesellschaft für Photogrammetrie, Fernerkundung und Geoinformation (DGPF) e.V.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1976, March). On Finding Lowest Common Ancestors in Trees. In SIAM Journal on Computing (pp. 115–132, Vol. 5). Society for Industrial & Applied Mathematics (SIAM). https://doi.org/10.1137/0205011

Aho, A. V., & Hopcroft, J. E. (1974, January). The Design and Analysis of Computer Algorithms (1st ed.). Addison-Wesley Longman Publishing Co., Inc.

Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland (AdV). (2008, April). Dokumentation zur Modellierung der Geoinformationen des amtlichen Vermessungswesens (GeoInfoDok) - Hauptdokument. Version 6.0.

Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland (AdV). (2022). Dokumentation zur Modellierung der Geoinformationen des amtlichen Vermessungswesens (GeoInfoDok) - Ausleitung des Objektartenkataloges für das AAA-Anwendungsschema. Version 7.1.2.

Arlazarov, V. L., Zuev, I. I., Uskov, A. V., & Faradzhev, I. A. (1974, January). An Algorithm for the Reduction of Finite Non-oriented Graphs to Canonical Form. In USSR Computational Mathematics and Mathematical Physics (pp. 195–201, Vol. 14). Elsevier BV. https://doi.org/10.1016/0041-5553(74)90114-1

Babai, L. (2016, June). Graph Isomorphism in Quasipolynomial Time [Extended Abstract]. In Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing (pp. 684–697). Association for Computing Machinery (ACM). https://doi.org/10.1145/2897518.2897542

Babai, L., Grigoryev, D. Y., & Mount, D. M. (1982). Isomorphism of Graphs with Bounded Eigenvalue Multiplicity. In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (pp. 310–324). Association for Computing Machinery (ACM). https://doi.org/10.1145/800070.802206

Babai, L., & Luks, E. M. (1983, December). Canonical Labeling of Graphs. In Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (pp. 171–183). Association for Computing Machinery (ACM). https://doi.org/10.1145/800061.808746

Bayer, R., & McCreight, E. (1970, November). Organization and Maintenance of Large Ordered Indices. In Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (pp. 107–141). Association for Computing Machinery (ACM). https://doi.org/10.1145/1734663.1734671

Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990, May). The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In ACM SIGMOD Record (pp. 322–331, Vol. 19). Association for Computing Machinery (ACM). https://doi.org/10.1145/93605.98741

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., & Cöltekin, A. (2015, December). Applications of 3D City Models: State of the Art Review. In ISPRS International Journal of Geo-Information (pp. 2842–2889, Vol. 4). MDPI AG. https://doi.org/10.3390/ijgi4042842

Bodlaender, H. L. (1990, December). Polynomial Algorithms for Graph Isomorphism and Chromatic Index on Partial k-trees. In Journal of Algorithms (pp. 631–643, Vol. 11). Elsevier BV. https://doi.org/10.1016/0196-6774(90)90013-5

Bomze, I. M., Budinich, M., Pardalos, P. M., & Pelillo, M. (1999). The Maximum Clique Problem. In Handbook of Combinatorial Optimization: Supplement Volume A (pp. 1–74). Springer US. https://doi.org/10.1007/978-1-4757-3023-4_1

Bourret, R. (2005). XML and Databases. Retrieved March 26, 2024, from http://www.rpbourret.com/xml

Cayley, A. (1857, March). XXVIII. On the Theory of the Analytical Forms Called Trees. In The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science (pp. 172–176, Vol. 13). Informa UK Limited. https://doi.org/10.1080/14786445708642275

Chamberlin, D. D., & Boyce, R. F. (1976, May). SEQUEL: A Structured English Query Language. In Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (pp. 249–264). Association for Computing Machinery (ACM). https://doi.org/10.1145/800296.811515

Chaturvedi, K., Matheus, A., Nguyen, S. H., & Kolbe, T. H. (2018, October). Securing Spatial Data Infrastructures in the Context of Smart Cities. In 2018 International Conference on Cyberworlds (CW) (pp. 403–408). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/cw.2018.00078

Chaturvedi, K., Matheus, A., Nguyen, S. H., & Kolbe, T. H. (2019, December). Securing Spatial Data Infrastructures for Distributed Smart City Applications and Services. In Future Generation Computer Systems (pp. 723–736, Vol. 101). Elsevier BV. https://doi.org/10.1016/j.future.2019.07.002

Chawathe, S. S., & Garcia-Molina, H. (1997, June). Meaningful Change Detection in Structured Data. In Proceedings of the 1997 ACM SIGMOD international conference on Management of data - SIGMOD '97 (pp. 26–37, Vol. 26). Association for Computing Machinery (ACM). https://doi.org/10.1145/253260.253266

Cheng, Y., Ding, P., Wang, T., Lu, W., & Du, X. (2019, November). Which Category is Better: Benchmarking Relational and Graph Database Management Systems. In Data Science and Engineering (pp. 309–322, Vol. 4). Springer Science and Business Media LLC. https://doi.org/10.1007/s41019-019-00110-3

Cobena, G., Abiteboul, S., & Marian, A. (2002, August). Detecting Changes in XML Documents. In Proceedings 18th International Conference on Data Engineering (pp. 41–52). IEEE Comput. Soc. https://doi.org/10.1109/icde.2002.994696

Cobham, A. (1965). The Intrinsic Computational Difficulty of Functions. In Y. Bar-Hillel (Ed.), Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (pp. 24–30). North-Holland Publishing Company.

Codd, E. F. (1970, June). A Relational Model of Data for Large Shared Data Banks. In Communications of the ACM (pp. 377–387, Vol. 13). Association for Computing Machinery (ACM). https://doi.org/10.1145/362384.362685

Coffman, E. G., Elphick, M., & Shoshani, A. (1971, June). System Deadlocks. In ACM Computing Surveys (pp. 67–78, Vol. 3). Association for Computing Machinery (ACM). https://doi.org/10.1145/356586.356588

Colbourn, C. J., & Booth, K. S. (1981, February). Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs. In SIAM Journal on Computing (pp. 203–225, Vol. 10). Society for Industrial & Applied Mathematics (SIAM). https://doi.org/10.1137/0210015

Combined Communications-Electronics Board (CCEB). (2010, October). ACP 121(I), Communication Instructions - General. Retrieved August 7, 2024, from http://www.rpbourret.com/xml

Congress Center Hamburg. (2022). The New CCH - Congress Center Hamburg: Great Location, Leading-edge Architecture, Top Technology. Retrieved February 27, 2024, from https://www.cch.de/en/news-details/article/das-neue-cch-congress-center-hamburg-beste-lage-moderne-architektur-und-hochwertige-technik

Conte, D., Foggia, P., Sansone, C., & Vento, M. (2004, May). Thirty Years Of Graph Matching In Pattern Recognition. In International Journal of Pattern Recognition and Artificial Intelligence (pp. 265–298, Vol. 18). World Scientific Pub Co Pte Lt. https://doi.org/10.1142/s0218001404003228

Cook, S. A. (1971, May). The Complexity of Theorem-Proving Procedures. In Proceedings of the Third Annual ACM Symposium on Theory of Computing (pp. 151–158). Association for Computing Machinery (ACM). https://doi.org/10.1145/800157.805047

Corneil, D. G., & Gotlieb, C. C. (1970, January). An Efficient Algorithm for Graph Isomorphism. In G. Salton (Ed.), Journal of the ACM (pp. 51–64, Vol. 17). Association for Computing Machinery (ACM). https://doi.org/10.1145/321556.321562

Cox, S., Daisey, P., Lake, R., Portele, C., & Whiteside, A. (2004, February). OpenGIS(R) Geography Markup Language (GML) Implementation Specification [OGC 03-105r1, Version 3.1.1, Recommendation Paper]. Open Geospatial Consortium (OGC). https://doi.org/10.13140/2.1.2846.2401

Customer Information Quality Technical Committee (CIQ TC). (2002, July). Extensible Address Language (xAL) Standard Description Document for W3C DTD/Schema (Version 2.0) [Approved Committee Specification]. OASIS. https://www.immagic.com/eLibrary/ARCHIVES/TECH/OASIS/XAL_V2.PDF

Czerwiński, W., & Orlikowski, Ł. (2022, February). Reachability in Vector Addition Systems is Ackermann-complete. In 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS) (pp. 1229–1240). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/focs52979.2021.00120

Davoudian, A., Chen, L., & Liu, M. (2018, April). A Survey on NoSQL Stores. In ACM Computing Surveys (pp. 1–43, Vol. 51). Association for Computing Machinery (ACM). https://doi.org/10.1145/3158661

Dehbi, Y., & Plümer, L. (2011, March). Learning Grammar Rules of Building Parts from Precise Models and Noisy Observations. In ISPRS Journal of Photogrammetry and Remote Sensing (pp. 166–176, Vol. 66). Elsevier BV. https://doi.org/10.1016/j.isprsjprs.2010.10.001

Devopedia. (2023). Duck Typing (8). Retrieved March 14, 2024, from https://devopedia.org/duck-typing

Dijkstra, E. W. (1959, December). A Note on Two Problems in Connexion with Graphs. In Numerische Mathematik (pp. 269–271, Vol. 1). Springer Science; Business Media LLC. https://doi.org/10.1007/bf01386390

Ding, L., Xiao, G., Pano, A., Fumagalli, M., Chen, D., Feng, Y., Calvanese, D., Fan, H., & Meng, L. (2024, April). Integrating 3D City Data through Knowledge Graphs. In Geo-spatial Information Science (pp. 1–20). Informa UK Limited. https://doi.org/10.1080/10095020.2024.2337360

Dittrich, K. R., Gatziu, S., & Geppert, A. (1995, June). The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In T. Sellis (Ed.), Rules in Database Systems (pp. 1–17, Vol. 985). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-60365-4_116

Docker, Inc. (2024). Docker Docs. Retrieved February 6, 2024, from https://docs.docker.com

EHP Erste Hanseatische Projektmanagement GmbH. (2023). Bunker St. Pauli. Retrieved March 16, 2024, from https://www.bunker-stpauli.de

Eppstein, D. (1999, November). Subgraph Isomorphism in Planar Graphs and Related Problems. In Journal of Graph Algorithms and Applications (pp. 1–27, Vol. 3). World Scientific Publishing. https://doi.org/10.7155/jgaa.00014

Esser, S. (2024). Inkrementelle Versionskontrolle verteilt vorliegender Objektmodelle im Bauwesen [Doctoral dissertation, Technical University of Munich].

Esser, S., Vilgertshofer, S., & Borrmann, A. (2022, August). Graph-based Version Control for Asynchronous BIM Collaboration. In Advanced Engineering Informatics (p. 101664, Vol. 53). Elsevier BV. https://doi.org/10.1016/j.aei.2022.101664

Euler, L. (1736). Solutio Problematis Ad Geometriam Situs Pertinentis (The Solution of a Problem Relating to the Geometry of Position). In Commentarii Academiae Scientiarum Petropolitanae (pp. 128–140, Vol. 8). Petropoli, Typis Academiae.

Euler, L. (1995, January). From the Problem of the Seven Bridges of Königsberg. In R. Calinger (Ed.), Classics of Mathematics (pp. 503–506). Prentice Hall.

Falkowski, K., & Ebert, J. (2009). Graph-based Urban Object Model Processing. In U. Stilla, F. Rottensteiner & N. Paparoditis (Eds.), City Models, Roads and Traffic (CMRT '09): Object Extraction for 3D City Models, Road Databases and Traffic Monitoring - Concepts, Algorithms and Evaluation (pp. 115–120, Vol. 38). International Society for Photogrammetry and Remote Sensing (ISPRS). https://www.isprs.org/proceedings/xxxviii/3-w4

Fernandes, D., & Bernardino, J. (2018). Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In Proceedings of the 7th International Conference on Data Science, Technology and Applications (pp. 373–380). SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0006910203730380

Filotti, I. S., & Mayer, J. N. (1980, April). A Polynomial-Time Algorithm for Determining the Isomorphism of Graphs of Fixed Genus. In Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing (pp. 236–243). Association for Computing Machinery (ACM). https://doi.org/10.1145/800141.804671

Forgy, C. L. (1982, September). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In Artificial Intelligence (pp. 17–37, Vol. 19). Elsevier BV. https://doi.org/10.1016/0004-3702(82)90020-0

Fortune Business Insights. (2023). Digital Twin Market Size, Share, Growth and Forecast 2030. Retrieved January 24, 2024, from https://www.fortunebusinessinsights.com/digital-twin-market-106246

Garey, M. R., & Johnson, D. S. (1979, January). Computers and Intractability: A Guide to the Theory of NP-Completeness (1st ed.). W. H. Freeman & Co.

GNU, Free Software Foundation. (2023). GNU General Public License (3.0). Retrieved August 27, 2023, from https://www.gnu.org/licenses/gpl-3.0.en.html

Goodrich, M. T., & Tamassia, R. (2014). Algorithm Design and Applications (1st ed.). Wiley Publishing.

Grieves, M., & Vickers, J. (2016, August). Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In F.-J. Kahlen, S. Flumerfelt & A. Alves (Eds.), Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches (pp. 85–113). Springer International Publishing. https://doi.org/10.1007/978-3-319-38756-7_4

Grochowski, K., Breiter, M., & Nowak, R. (2020, March). Serialization in Object-Oriented Programming Languages. In K. Sud, P. Erdogmus & S. Kadry (Eds.), Introduction to Data Science and Machine Learning. IntechOpen. https://doi.org/10.5772/intechopen.86917

Gröger, G., Kolbe, T. H., Czerwinski, A., & Nagel, C. (2008, August). OpenGIS(R) City Geography Markup Language (CityGML) Encoding Standard [OGC 08-007r1, Version 1.0.0, International Standard]. Open Geospatial Consortium (OGC). https://portal.ogc.org/files/?artifact_id=28802

Gröger, G., Kolbe, T. H., Nagel, C., & Häfele, K.-H. (2012, April). OGC City Geography Markup Language (CityGML) Encoding Standard [OGC 12-019, Version 2.0.0, International Standard]. Open Geospatial Consortium (OGC). https://portal.ogc.org/files/?artifact_id=47842

Grohe, M. (2010, July). Fixed-Point Definability and Polynomial Time on Graphs with Excluded Minors. In 2010 25th Annual IEEE Symposium on Logic in Computer Science (pp. 179–188). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/lics.2010.22

Grohe, M., & Schweitzer, P. (2015, October). Isomorphism Testing for Graphs of Bounded Rank Width. In 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (pp. 1010–1029). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/focs.2015.66

Grosan, C., & Abraham, A. (2011). Rule-Based Expert Systems. In Intelligent Systems: A Modern Approach (pp. 149–185). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21004-4_7

Guttman, A. (1984, June). R-Trees: A Dynamic Index Structure for Spatial Searching [Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data]. In ACM SIGMOD Record (pp. 47–57, Vol. 14). Association for Computing Machinery (ACM). https://doi.org/10.1145/971697.602266

Häfele, K.-H., & Nagel, C. (2015). CityGML Example Railway Scene. Institute for Automation, Applied Computer Science (IAI), Karlsruhe Institute of Technology (KIT) and virtualcitysystems GmbH. Retrieved August 7, 2024, from https://github.com/3dcitydb/importer-exporter

Harel, D., & Tarjan, R. E. (1984, May). Fast Algorithms for Finding Nearest Common Ancestors. In SIAM Journal on Computing (pp. 338–355, Vol. 13). Society for Industrial & Applied Mathematics (SIAM). https://doi.org/10.1137/0213024

Harold, E. R. (2003, September). An Introduction to StAX. Retrieved October 18, 2023, from https://www.xml.com/pub/a/2003/09/17/stax.html

Harold, E. R., Means, W. S., & Petrycki, L. (2000, December). XML in a Nutshell: A Desktop Quick Reference (1st ed.). O'Reilly & Associates, Inc.

Harter, H. M. (2021). Lebenszyklusanalyse der Technischen Gebäudeausrüstung großer Wohngebäudebestände auf der Basis semantischer 3D-Stadtmodelle [Doctoral dissertation, Technical University of Munich].

Hausdorff, F. (1914, March). Grundzüge der Mengenlehre (1st ed.). Verlag von Veit & Comp.

Haverkort, H. J. (2004, May). Results on Geometric Networks and Data Structures [Doctoral dissertation, Utrecht University].

Heckel, R. (2006, February). Graph Transformation in a Nutshell. In Electronic Notes in Theoretical Computer Science (pp. 187–198, Vol. 148). Elsevier BV. https://doi.org/10.1016/j.entcs.2005.12.018

Held, M., & Karp, R. M. (1970, December). The Traveling-Salesman Problem and Minimum Spanning Trees. In Operations Research (pp. 1138–1162, Vol. 18). INFORMS.

Herring, J. R. (2011, May). OpenGIS(R) Implementation Standard for Geographic Information - Simple Feature Access - Part 1: Common Architecture [OGC 06-103r4, Version 1.2.1, Implementation Standard]. Open Geospatial Consortium (OGC). https://portal.ogc.org/files/?artifact_id=25355

Herring, J. R. (2020, August). Features and Geometry - Part 1: Feature Models [17-087r13, Version 1.0, Abstract Specification]. Open Geospatial Consortium (OGC). http://www.opengis.net/doc/AS/FG/P1/FM/1.0

Hogan, A., Blomqvist, E., Cochez, M., D'Amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., & Zimmermann, A. (2021, July). Knowledge Graphs. In ACM Computing Surveys (pp. 1–37, Vol. 54). Association for Computing Machinery (ACM). https://doi.org/10.1145/3447772

Hopcroft, J. E., & Wong, J. K. (1974, April). Linear Time Algorithm for Isomorphism of Planar Graphs (Preliminary Report). In Proceedings of the Sixth Annual ACM Symposium on Theory of Computing (pp. 172–184). Association for Computing Machinery (ACM). https://doi.org/10.1145/800119.803896

Horowitz, E., & Sahni, S. (1983, January). Fundamentals of Data Structures (4th ed.). Computer Science Press.

Hsieh, S.-M., Hsu, C.-C., & Hsu, L.-F. (2006). Efficient Method to Perform Isomorphism Testing of Labeled Graphs [Proceedings, Part V]. In M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganá, Y. Mun & H. Choo (Eds.), Computational Science and Its Applications - ICCSA 2006 (pp. 422–431). Springer-Verlag Berlin Heidelberg.

Huan, J., Wang, W., & Prins, J. (2003, December). Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In Third IEEE International Conference on Data Mining (pp. 549–552). IEEE Comput. Soc. https://doi.org/10.1109/icdm.2003.1250974

Hunt, J. W., & Szymanski, T. G. (1977, May). A Fast Algorithm for Computing Longest Common Subsequences. In Communications of the ACM (pp. 350–353, Vol. 20). Association for Computing Machinery (ACM). https://doi.org/10.1145/359581.359603

Institute for Automation and Applied Computer Science (IAI), Karlsruhe Institute of Technology (KIT). (2017). CityGML Example FZK-Haus. Retrieved November 15, 2023, from https://www.citygmlwiki.org

Institute for Automation and Applied Computer Science (IAI), Karlsruhe Institute of Technology (KIT). (2024). KITModelViewer. Retrieved August 5, 2024, from https://www.iai.kit.edu/4561.php

International Organization for Standardization (ISO). (2019a, February). ISO 8601-1:2019 Date and Time - Representations for Information Interchange - Part 1: Basic Rules (1st ed., Vol. 01.140.30) [International Standard]. https://www.iso.org/standard/70907.html

International Organization for Standardization (ISO). (2019b, February). ISO 8601-2:2019 Date and Time - Representations for Information Interchange - Part 2: Extensions (1st ed., Vol. 01.140.30) [International Standard]. https://www.iso.org/standard/70908.html

Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009). A Classification of Object-Relational Impedance Mismatch. In 2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications (pp. 36–43). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/dbkda.2009.11

Jaccard, P. (1912, February). The Distribution of the Flora in the Alpine Zone. In New Phytologist (pp. 37–50, Vol. 11). Wiley. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

Jang, H., Yu, K., & Park, S. (2023). Managing 3D GIS Data for Indoor Environment Using Property Graph Database. In IEEE Access (pp. 37216–37228, Vol. 11). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/access.2023.3266519

Jensen, K. (1987, May). Coloured Petri Nets. In W. Brauer, W. Reisig & G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties (1st ed., pp. 248–299). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-47919-2_10

Kabsch, W. (1976, September). A Solution for the Best Rotation to Relate Two Sets of Vectors. In Acta Crystallographica Section A (pp. 922–923, Vol. 32). International Union of Crystallography (IUCr). https://doi.org/10.1107/s0567739476001873

Kaden, R. (2014). Berechnung der Energiebedarfe von Wohngebäuden und Modellierung energiebezogener Kennwerte auf der Basis semantischer 3D-Stadtmodelle [Doctoral dissertation, Technical University of Munich].

Kajiya, J. T. (1986, August). The Rendering Equation. In ACM SIGGRAPH Computer Graphics (pp. 143–150, Vol. 20). Association for Computing Machinery (ACM). https://doi.org/10.1145/15886.15902

Kawaguchi, K., Vajjhala, S., Fialli, J., & Grigoriadi, R. (2017). The Java Architecture for XML Binding (JAXB) 2.3 Specification. Oracle Corporation. Retrieved September 27, 2023, from https://www.jcp.org/en/jsr/detail?id=222

Keller, W. (1997). Mapping Objects to Tables - A Pattern Language. In F. Buschmann & D. Riehle (Eds.), Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP) '97. Siemens AG.

Kleppe, A. G., Warmer, J., & Bast, W. (2003, April). MDA Explained: The Model Driven Architecture: Practice and Promise (Revised). Addison-Wesley Longman Publishing Co., Inc.

Kolbe, T. H. (2009). Representing and Exchanging 3D City Models with CityGML. In J. Lee & S. Zlatanova (Eds.), 3D Geo-Information Sciences (pp. 15–31). Springer. https://doi.org/10.1007/978-3-540-87395-2

Kolbe, T. H., & Donaubauer, A. (2021). Semantic 3D City Modeling and BIM. In W. Shi, M. F. Goodchild, M. Batty, M.-P. Kwan & A. Zhang (Eds.), Urban Informatics (pp. 609–636). Springer Singapore. https://doi.org/10.1007/978-981-15-8983-6_34

Kolbe, T. H., Gröger, G., & Plümer, L. (2008). CityGML - 3D City Models and their Potential for Emergency Response. In S. Zlatanova & J. Li (Eds.), Geospatial Information Technology for Emergency Response (pp. 257–274). Taylor & Francis. https://www.taylorfrancis.com/books/9780429224065

Kolbe, T. H., Kutzner, T., Smyth, C. S., Nagel, C., Roensdorf, C., & Heazel, C. (2021, September). OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard [20-010, Version 3.0.0, International Standard]. Open Geospatial Consortium (OGC). https://www.opengis.net/doc/IS/CityGML-1/3.0

Kuramochi, M., & Karypis, G. (2004, September). An Efficient Algorithm for Discovering Frequent Subgraphs. In IEEE Transactions on Knowledge and Data Engineering (pp. 1038–1051, Vol. 16). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/tkde.2004.33

Kutzner, T. (2016, December). Geospatial Data Modelling and Model-driven Transformation of Geospatial Data based on UML Profiles [Doctoral dissertation, Technical University of Munich].

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., & Vitalis, S. (2019, June). CityJSON: A Compact and Easy-to-use Encoding of the CityGML Data Model. In Open Geospatial Data, Software and Standards (Vol. 4). Springer Science and Business Media LLC. https://doi.org/10.1186/s40965-019-0064-0

Lee, J., Li, K.-J., Zlatanova, S., Kolbe, T. H., Nagel, C., Becker, T., & Kang, H.-Y. (2020, November). OGC(R) IndoorGML 1.1 [19-011R4, Version 1.1, Standard Implementation]. Open Geospatial Consortium (OGC)". http://www.opengis.net/doc/IS/indoorgml/1.1

Lei, B., Janssen, P., Stoter, J., & Biljecki, F. (2023, March). Challenges of Urban Digital Twins: A Systematic Review and a Delphi Expert Survey. In Automation in Construction (p. 104716, Vol. 147). Elsevier BV. https://doi.org/10.1016/j.autcon.2022.104716

Leroux, J. (2022, February). The Reachability Problem for Petri Nets is Not Primitive Recursive. In 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS) (pp. 1241–1252). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/focs52979.2021.00121

Lipton, R. (1976). The Reachability Problem Requires Exponential Space. In Research Report (Vol. 62). Department of Computer Science, Yale University.

Luks, E. M. (1982, August). Isomorphism of Graphs of Bounded Valence can be Tested in Polynomial Time. In Journal of Computer and System Sciences (pp. 42–65, Vol. 25). Elsevier BV. https://doi.org/10.1016/0022-0000(82)90009-5

Mäntylä, M. (1988, January). An Introduction to Solid Modeling (1st ed.). W. H. Freeman & Co.

Mao, B., & Li, B. (2019, December). Graph-Based 3D Building Semantic Segmentation for Sustainability Analysis. In Journal of Geovisualization and Spatial Analysis (Vol. 4). Springer Science and Business Media LLC. https://doi.org/10.1007/s41651-019-0045-y

Masri, N., Sultan, Y. A., Akkila, A. N., Almasri, A., Ahmed, A., Mahmoud, A. Y., Zaqout, I., & Abu-Naser, S. S. (2019). Survey of Rule-based Systems. In International Journal of Academic Information Systems Research (IJAISR) (pp. 1–22, Vol. 3). International Journal of Academic Information Systems Research (IJAISR).

McCluskey, G. (1998, January). Using Java Reflection. Retrieved October 8, 2023, from https://www.oracle.com/technical-resources/articles/java/javareflection.html

McCulloch, W. S., & Pitts, W. (1943, December). A Logical Calculus of the Ideas Immanent in Nervous Activity. In The Bulletin of Mathematical Biophysics (pp. 115–133, Vol. 5). Springer Science and Business Media LLC. https://doi.org/10.1007/bf02478259

McKay, B. D. (1978). Computing Automorphisms and Canonical Labellings of Graphs. In Combinatorial Mathematics (pp. 223–232, Vol. 686). Springer Berlin Heidelberg. https://doi.org/10.1007/bfb0062536

McKay, B. D. (1980). Practical Graph Isomorphism. In Congressus Numerantium (pp. 45–87, Vol. 30). Combinatorial Press.

McKay, B. D., & Piperno, A. (2014, January). Practical Graph Isomorphism, II. In Journal of Symbolic Computation (pp. 94–112, Vol. 60). Elsevier BV. https://doi.org/10.1016/j.jsc.2013.09.003

McKay, B. D., & Piperno, A. (2023, November). nauty and Traces User's Guide (Version 2.8.8). Retrieved August 9, 2024, from https://pallini.di.uniroma1.it

Metaver Metadatenverbund, Landesbetrieb Geoinformation und Vermessung (LGV) Hamburg. (2024). 3D-Gebäudemodell LoD2-DE Hamburg. Retrieved January 18, 2024, from https://metaver.de

Michelson, B. M. (2006, February). Event-driven Architecture Overview - Event-driven SOA is Just Part of the EDA Story. Patricia Seybold Group and Elemental Links, Inc. https://doi.org/10.1571/bda2-2-06cc

Miller, G. (1980, April). Isomorphism Testing for Graphs of Bounded Genus. In Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing (pp. 225–235). Association for Computing Machinery (ACM). https://doi.org/10.1145/800141.804670

Miyazaki, T. (1997). The Complexity of McKay's Canonical Labeling Algorithm. In L. Finkelstein & W. M. Kantor (Eds.), Groups and Computation II, DIMACS Series on Discrete Mathematics and Theoretical Computer Science (pp. 239–256, Vol. 28). American Mathematical Society.

Murthy, R., Liu, Z. H., Krishnaprasad, M., Chandrasekar, S., Tran, A.-T., Sedlar, E., Florescu, D., Kotsovolos, S., Agarwal, N., Arora, V., & Krishnamurthy, V. (2005, June). Towards an Enterprise XML Architecture. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (pp. 953–957).

Association for Computing Machinery (ACM). https://doi.org/10.1145/1066157.1066300

Myers, E. W. (1986, November). An O(ND) Difference Algorithm and Its Variations. In Algorithmica (pp. 251–266, Vol. 1). Springer Science and Business Media LLC. https://doi.org/10.1007/bf01840446

Neo4j, Inc. (2023). Neo4j Licensing. Retrieved August 27, 2023, from https://neo4j.com/licensing

Neo4j, Inc. (2023). The Neo4j Operations Manual v5. Retrieved October 22, 2023, from https://neo4j.com/docs/operations-manual/5

Neuen, D. (2016). Graph Isomorphism for Unit Square Graphs. In P. Sankowski & C. D. Zaroliagis (Eds.), 24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark (70:1–70:17, Vol. 57). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ESA.2016.70

Newman, M. E. J. (2003, January). The Structure and Function of Complex Networks. In SIAM Review (pp. 167–256, Vol. 45). Society for Industrial & Applied Mathematics (SIAM). https://doi.org/10.1137/s003614450342480

Nguyen, S. H., & Kolbe, T. H. (2020, September). A Multi-Perspective Approach to Interpreting Spatio-Semantic Changes of Large 3D City Models in CityGML using a Graph Database [15th International 3D GeoInfo Conference 2020, University College London (UCL), London, UK]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 143–150, Vol. VI-4/W1-2020). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-vi-4-w1-2020-143-2020

Nguyen, S. H., & Kolbe, T. H. (2021, October). Modelling Changes, Stakeholders and their Relations in Semantic 3D City Models [16th International 3D GeoInfo Conference 2021, New York University (NYU), NY, USA]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 137–144, Vol. VIII-4/W2-2021). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-viii-4-w2-2021-137-2021

Nguyen, S. H., & Kolbe, T. H. (2022, October). Path-tracing Semantic Networks to Interpret Changes in Semantic 3D City Models [17th International 3D GeoInfo Conference 2022, University of New South Wales (UNSW), Sydney, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 217–224, Vol. X-4/W2-2022). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-x-4-w2-2022-217-2022

Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2017, October). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database [12th International 3D GeoInfo Conference 2017, University of Melbourne, Melbourne, Australia]. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information

Sciences (pp. 99–106, Vol. IV-4/W5). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-iv-4-w5-99-2017

Nguyen, S. H. (2017, May). Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database [Master's thesis, Technical University of Munich].

Nguyen, S. H., & Kolbe, T. H. (2024, September). Identification and Interpretation of Change Patterns in Semantic 3D City Models [18th 3D GeoInfo Conference 2023, Technical University of Munich (TUM), Munich, Germany]. In T. H. Kolbe, A. Donaubauer & C. Beil (Eds.), Recent Advances in 3D Geoinformation Science (pp. 479–496). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43699-4_30

Nguyen, S. H., Yao, Z., & Kolbe, T. H. (2018). Spatio-Semantic Comparison of Large 3D City Models in CityGML Using a Graph Database. In gis.Science (pp. 85–100, Vol. 3). Wichmann Verlag. https://gispoint.de/artikelarchiv/gis/2018/gisscience-ausgabe-32018.html

Nicola, M., & van der Linden, B. (2005, August). Native XML Support in DB2 Universal Database. In Proceedings of the 31st International Conference on Very Large Data Bases (pp. 1164–1174). VLDB Endowment.

Obe, R. O., & Hsu, L. S. (2017). PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database (3rd ed.). O'Reilly Media, Inc.

Olbrich, F. (2023, November). Multimodal Navigation Applications for CityGML 3.0 using a Graph Database [Master's thesis, Technical University of Munich].

Olbrich, F., Beil, C., Nguyen, S. H., & Kolbe, T. H. (2024, March). Multimodale Navigationsanwendungen für CityGML 3.0-konforme 3D-Straßenraummodelle mittels Graphdatenbanken. In T. P. Kersten & N. Tilly (Eds.), DGPF-Jahrestagung 2024 - Stadt, Land, Fluss - Daten vernetzen, 44. Wissenschaftlich-Technische Jahrestagung der DGPF (pp. 357–369, Vol. 32). Deutsche Gesellschaft für Photogrammetrie, Fernerkundung und Geoinformation (DGPF) e.V. https://doi.org/10.24407/KXP:1885708890

Pearson, K. (1900, July). X. On the Criterion That a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such That It Can Be Reasonably Supposed to Have Arisen from Random Sampling. In The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science (pp. 157–175, Vol. 50). Informa UK Limited. https://doi.org/10.1080/14786440009463897

Pédrinis, F., Morel, M., & Gesquière, G. (2014, November). Change Detection of Cities. In M. Breunig, E. Al-Doori Mulhim and Butwilowski, P. V. Kuper, J. Benner & K.-H. Häfele (Eds.), 3D Geoinformation Science (pp. 123–139). Springer International Publishing. https://doi.org/10.1007/978-3-319-12181-9_8

Petri, C. A. (1962, June). Kommunikation mit Automaten [Doctoral dissertation, Technical University of Darmstadt].

Pfaltz, J. L., & Rosenfeld, A. (1969, May). Web Grammars. In Proceedings of the 1st International Joint Conference on Artificial Intelligence (pp. 609–619). Morgan Kaufmann Publishers Inc.

Piperno, A. (2011). Search Space Contraction in Canonical Labeling of Graphs. arXiv. https://doi.org/10.48550/ARXIV.0804.4881

Ponomarenko, I. (1988, January). The Isomorphism Problem for Classes of Graphs that are Invariant with Respect to Contraction. In Zap. Nauchn. Sem. Leningrad. Otdel. Mat. Inst. Steklov. (LOMI) (pp. 147–177, Vol. 174).

Powałka, L., Poon, C., Xia, Y., Meines, S., Yan, L., Cai, Y., Stavropoulou, G., Dukai, B., & Ledoux, H. (2024). cjdb: A Simple, Fast, and Lean Database Solution for the CityGML Data Model. In T. H. Kolbe, A. Donaubauer & C. Beil (Eds.), Recent Advances in 3D Geoinformation Science (pp. 781–796). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-43699-4_47

Pratt, T. W. (1971, December). Pair Grammars, Graph Languages and String-to-Graph Translations. In Journal of Computer and System Sciences (pp. 560–595, Vol. 5). Elsevier BV. https://doi.org/10.1016/s0022-0000(71)80016-8

Python Software Foundation. (2023). marshal - Internal Python object serialization. Retrieved September 24, 2023, from https://docs.python.org/3/library/marshal.html

Redweik, R., & Becker, T. (2014, November). Change Detection in CityGML Documents. In M. Breunig, M. Al-Doori, E. Butwilowski, P. V. Kuper, J. Benner & K.-H. Häfele (Eds.), 3D Geoinformation Science (pp. 107–121). Springer International Publishing. https://doi.org/10.1007/978-3-319-12181-9_7

Reisig, W. (2013, July). Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies (1st ed.). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33278-4

Ryan, V., Seligman, S., & Lee, R. (1999, October). RFC2713: Schema for Representing Java(TM) Objects in an LDAP Directory. Network Working Group, The Internet Society. Retrieved August 7, 2024, from https://www.rfc-editor.org/rfc/rfc2713.html

Rys, M. (2005, June). XML and Relational Database Management Systems: Inside Microsoft(R) SQL Server(TM) 2005. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (pp. 958–962). Association for Computing Machinery (ACM). https://doi.org/10.1145/1066157.1066301

Schade, S., & Cox, S. J. D. (2010, July). Linked Data in SDI or How GML is not about Trees. In M. Painho, M. Y. Santos & H. Pundt (Eds.), Proceedings of the 13th AGILE International Conference on Geographic Information Science - Geospatial Thinking. Association of Geographic Information Laboratories for Europe (AGILE).

Schwab, B., Beil, C., & Kolbe, T. H. (2020, May). Spatio-Semantic Road Space Modeling for Vehicle-Pedestrian Simulation to Test Automated Driving Systems. In Sustainability (p. 3799, Vol. 12). MDPI AG. https://doi.org/10.3390/su12093799

Sellis, T. K., Roussopoulos, N., & Faloutsos, C. (1987, September). The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In P. M. Stocker, W. Kent & P. Hammersley (Eds.), Proceedings of the 13th International Conference on Very Large Data Bases (pp. 507–518). Morgan Kaufmann Publishers Inc.

Shafto, M., Conroy, M., Doyle, R., Glaessgen, E., Kemp, C., LeMoigne, J., & Wang, L. (2010, November). DRAFT Modeling, Simulation, Information Technology & Processing Roadmap - Technology Area 11. National Aeronautics and Space Administration (NASA). https://www.emacromall.com/reference/NASA-Modeling-Simulation-IT-Processing-Roadmap.pdf

Sharma, A., Kosasih, E., Zhang, J., Brintrup, A., & Calinescu, A. (2022, November). Digital Twins: State of the Art Theory and Practice, Challenges, and Open Research Questions. In Journal of Industrial Information Integration (p. 100383, Vol. 30). Elsevier BV. https://doi.org/10.1016/j.jii.2022.100383

Shields, R. (2012, July). Cultural Topology: The Seven Bridges of Königsburg, 1736. In Theory, Culture & Society (pp. 43–57, Vol. 29). SAGE Publications. https://doi.org/10.1177/0263276412451161

Skiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer London. https://doi.org/10.1007/978-1-84800-070-4

Stadler, A., & Kolbe, T. H. (2007). Spatio-semantic Coherence in the Integration of 3D City Models. In A. Stein (Ed.), Proceedings of the 5th International ISPRS Symposium on Spatial Data Quality ISSDQ 2007 in Enschede, The Netherlands, 13-15 June 2007. International Society for Photogrammetry and Remote Sensing (ISPRS).

Stadler, A., Nagel, C., König, G., & Kolbe, T. (2009, January). Making Interoperability Persistent: A 3D Geo Database Based on CityGML. In 3D Geo-Information Sciences (pp. 175–192). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-87395-2_11

Stouffs, R., Tauscher, H., & Biljecki, F. (2018, August). Achieving Complete and Near-Lossless Conversion from IFC to CityGML. In ISPRS International Journal of Geo-Information (p. 355, Vol. 7). MDPI AG. https://doi.org/10.3390/ijgi7090355

Trudeau, R. J. (1994, February). Introduction to Graph Theory (2nd Revised ed.). Dover Publications Inc.

Turing, A. M. (1936, November). On Computable Numbers, with an Application to the Entscheidungsproblem. In Proceedings of the London Mathematical Society (pp. 230–265, Vol. s2-42). Wiley. https://doi.org/10.1112/plms/s2-42.1.230

Ullmann, J. R. (1976, January). An Algorithm for Subgraph Isomorphism. In Journal of the ACM (pp. 31–42, Vol. 23). Association for Computing Machinery (ACM). https://doi.org/10.1145/321921.321925

Umeyama, S. (1991, April). Least-squares Estimation of Transformation Parameters between Two Point Patterns. In IEEE Transactions on Pattern Analysis and Machine Intelligence (pp. 376–380, Vol. 13). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/34.88573

United Nations Department of Economic and Social Affairs (UN DESA). (2019). World Urbanization Prospects: The 2018 Revision. United Nations. https://www.un-ilibrary.org/content/books/9789210043144

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010, April). A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In Proceedings of the 48th Annual Southeast Regional Conference. Association for Computing Machinery (ACM). https://doi.org/10.1145/1900008.1900067

Vinasco-Alvarez, D., Samuel, J., Servigne, S., & Gesquière, G. (2024, May). Towards an Automated Transformation of an nD Urban Data Model to a Computational Ontology Network: From UML to OWL, From CityGML 3.0 to "CityOWL". In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences (pp. 231–238, Vol. X-4/W4-2024). Copernicus GmbH. https://doi.org/10.5194/isprs-annals-x-4-w4-2024-231-2024

Volk, R., Stengel, J., & Schultmann, F. (2014, March). Building Information Modeling (BIM) for Existing Buildings - Literature Review and Future Needs. In Automation in Construction (pp. 109–127, Vol. 38). Elsevier BV. https://doi.org/10.1016/j.autcon.2013.10.023

Vukotic, A., Watt, N., Abedrabbo, T., Fox, D., & Partner, J. (2014). Neo4j in Action (1st ed.). Manning Publications Co.

Wagner, R. A., & Fischer, M. J. (1974, January). The String-to-String Correction Problem. In Journal of the ACM (pp. 168–173, Vol. 21). Association for Computing Machinery (ACM). https://doi.org/10.1145/321796.321811

Wang, Y., DeWitt, D. J., & Cai, J. Y. (2003, March). X-Diff: An Effective Change Detection Algorithm for XML Documents. In Proceedings 19th International Conference on Data Engineering 2003 (pp. 519–530). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/icde.2003.1260818

Werbos, P. (1974, August). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences [Doctoral dissertation, Harvard University].

West, D. (2000, August). Introduction to Graph Theory (Subsequent ed.). Pearson.

Willenborg, B., Sindram, M., & Kolbe, T. H. (2017, October). Applications of 3D City Models for a Better Understanding of the Built Environment. In M. Behnisch

& G. Meinel (Eds.), Trends in Spatial Analysis and Modelling (pp. 167–191, Vol. 19). Springer International Publishing. https://doi.org/10.1007/978-3-319-52522-8_9

Wood, L., Hors, A. L., Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Nicol, G., Robie, J., Sutor, R., & Wilson, C. (2000, September). Document Object Model (DOM) Level 1 Specification (Second Edition) - W3C Working Draft. Retrieved August 7, 2024, from https://www.w3.org/TR/2000/WD-DOM-Level-1-20000929

World Wide Web Consortium (W3C). (2006, May). XML Linking Language (XLink) 1.1. Retrieved August 7, 2024, from https://www.w3.org/TR/xlink11

Wysocki, O., Schwab, B., & Willenborg, B. (2022, January). Awesome CityGML: The Ultimate List of Open Data Semantic 3D City Models (Version 1.0). Zenodo. https://doi.org/10.5281/zenodo.5899096

Yao, Z. (2020, January). Domain Extendable 3D City Models - Management, Visualization, and Interaction [Doctoral dissertation, Technical University of Munich].

Yao, Z., & Kolbe, T. H. (2017). Dynamically Extending Spatial Databases to Support CityGML Application Domain Extensions using Graph Transformations. In T. P. Kersten (Ed.), Kulturelles Erbe erfassen und bewahren - Von der Dokumentation zum virtuellen Rundgang, 37. Wissenschaftlich-Technische Jahrestagung der DGPF (pp. 316–331, Vol. 26). Deutsche Gesellschaft für Photogrammetrie, Fernerkundung und Geoinformation (DGPF) e.V.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., & Kolbe, T. H. (2018, May). 3DCityDB - A 3D Geodatabase Solution for the Management, Analysis, and Visualization of Semantic 3D City Models based on CityGML. In Open Geospatial Data, Software and Standards (pp. 1–26, Vol. 3). Springer Science and Business Media LLC. https://doi.org/10.1186/s40965-018-0046-7

Zhang, K. (1993). A New Editing based Distance between Unordered Labeled Trees. In A. Apostolico, M. Crochemore, Z. Galil & U. Manber (Eds.), Combinatorial Pattern Matching (pp. 254–265, Vol. 684). Springer-Verlag. https://doi.org/10.1007/BFb0029810