Semantic Classification and Hash Code Accelerated Detection of Design Changes in BIM Models

Jia-Rui Lin[a,b,*], Yu-Cheng Zhou[a,b]

[a] Department of Civil Engineering, Tsinghua University, Beijing, China 100084
[b] Tsinghua University-Glodon Joint Research Centre for Building Information Model (RCBIM), Tsinghua University, Beijing, China 100084

Abstract

As the design process of a building or infrastructure is complex with iterative steps and multidisciplinary collaboration, design changes are inevitable and efficient change detection based on Building Information Modeling (BIM) is essential for design collaboration and quality control. Although some detection methods have been proposed to tackle the problem, majority of them regard syntactic data changes as design changes, which always yield incorrect and meaningless results. For instance, exchanging the locations of two identical columns is meaningless from the viewpoint of a designer, which the current methods take it as a design change. Otherwise, additional time is needed to avoid these issues, thus leading to high time consumption. To address this problem, we first propose a semantic classification of design changes with three categories of data changes and three levels of design changes. Then, an efficient detection method considering the proposed semantic classification of design changes is proposed with hash-code-based acceleration. After implementing the proposed and existing methods, the detection performance of our method is compared with the performances of existing methods and invited designers based on 4 groups of BIM models. Results show that our method could: 1) detect semantic changes with 100% correctness while reducing up to 98.1% of the detection time of existing methods, and 2) improve the average detection rate of designers from 81.3% to 100% while saving 99.88% of the design reviewing time (detecting 7 changes from 5800 instances). This research contributes to the body of knowledge a new definition of design changes and an approach to efficient design change detection with hash-code-based acceleration. Meanwhile, the proposed classification of design changes and developed BIM models in this research can also be taken as a baseline for developing and validating new detection methods.

Keywords: building information modeling (BIM); change detection; semantic classification; hash code acceleration; design automation; collaboration; smart design

1 Introduction

The Architecture, Engineering, and Construction (AEC) design process is complex and involves many iterative steps

* Corresponding author, email: lin611@tsinghua.edu.cn, jiarui_lin@foxmail.com

and multidisciplinary design work that can be performed sequentially, concurrently or in parallel [1]. Changes in design are inevitable, and come from multiple sources at any time, which lead to extensive impacts [2,3]. Over the past decade, Building Information Modeling (BIM) has increasingly been adopted as a digital representation, as a knowledge resource and as an integrated method for building lifecycle management [4-7]. However, it is challenging to detect design changes as they are often documented in meeting memos, mail correspondences, on a post-it on the project manager's table, or solely in the memory of the project participants [8-10]. Furthermore, substantial information losses occur whenever an employee leaves the project [11]. Consequently, the effects of design changes are so opaque that the consequences are hard to judge, sometimes resulting in critical after-effects even in current BIM-based workflows [12]. On the other hand, this situation means that design change detection plays an important role in design collaboration. For instance, designers must share and exchange data among diverse software tools [13,14]; the ability to detect design changes can reduce the amount of transferred information and improve the efficiency of partial model exchange [15]. Change detection also helps designers, engineers, and managers focus on changes during the design process. However, manually detecting or checking for design changes is time consuming and error prone. Therefore, automated design change detection is highly important.

Design change detection began when 2D-based drawings were used to represent a facility. In the CAD environment, a common way to detect changes is using three primitive operators that capture the essence of available CAD drawing and editing operators: insert, delete and replace [16]. In 3D object-based BIM models, additional information, such as 3D shapes and properties, can be expressed in a model, and design change detection has been extensively studied by researchers. However, semantic design change, which represents meaningful design change from a designer's point of view in this research, is not widely considered in many studies, thus leading to meaningless detection results and wasting a lot of time. Currently, many detection methods usually match instances of two design versions by their IDs (e.g., GUID) and comparing all properties of each matched pair in BIM models [17,18]. These methods failed to consider the semantics of instances in BIM models, and they may detect meaningless change results because they regard data changes as design changes. For example, exchanging the locations of two identical columns will be detected as a change by such methods, which actually should not be considered as a change from a designer's point of view. Similarly, deleting and recreating the same beam should not be considered a change. In addition, these methods are sensitive to the IDs of instances. When the IDs change unexpectedly, the detection result will be incorrect. Many studies have noted these issues and proposed improved detection methods, such as the geometry-based [19] and content-based [20] automatic comparison approaches. However, these methods are still not perfect because the detection results also contain meaningless changes, and they are usually time consuming.

Obviously, the definition of semantic design changes is not clear, and is always mixed with data changes. To address

these issues, this research focuses on two aspects of design change: 1) classification, which provides a definition of semantic design changes and is a criterion for design change detection, and 2) detection, which is a method as well as an algorithm to identify semantic design changes according to the classification. First, we propose a semantic classification of design changes that divides data changes into three categories and classifies design changes into three levels from a designer's viewpoint. Next, we improve the detection method by considering the proposed classification criterion to enable the detection of semantic design changes. Finally, the hash-code-based encoding of instance properties is utilized to accelerate the comparison process of design change detection.

The remainder of this paper is structured as follows. Section 2 reviews the related works, including those on the classification and detection of design changes, and highlights potential research gaps. Section 3 describes the methodology of this research. Section 4 proposes a semantic classification of design changes. Section 5 proposes a hash-code-accelerated method that can detect semantic changes with high efficiency. Section 6 implements both the proposed and existing methods, establishes BIM models, and tests the detection performances of both these methods and of designers. Section 7 discusses the applicability and novelty of this contribution and point out some complicated work that can expand the application of the proposed method. Finally, Section 8 concludes this research and discusses future work.

2 Related Works

In the AEC design process, design changes occur rapidly during the design, engineering and construction phases [12]; the changes cannot always be avoided or fully anticipated, and are also identified as one of the major causes of construction project failure and a key source of project delays [21,22]. To respond to the market demands, project owners may request a series of design changes to project designs at any phase of the project [23]. Currently, in the BIM-based workflows, extensive studies of automatic design change detection have been done by many researchers.

In the BIM model data representation standard Industry Foundation Classes (IFC) and mostly used BIM platforms (e.g., Autodesk Revit), each instance has an identifier that can be used to track its information changes such as GUID in IFC or ElementId in Revit; this research calls these identifiers as ID. IFC defines an EXPRESS-based entity-relationship model that consists of several hundred entities organized into an object-based inheritance hierarchy [24]. In Revit, the data in a document consists primarily of a collection of elements [25] in which each element has many properties. Generally, the BIM model structure can be described by instances (objects) that possess many properties. The mostly used method of design change detection is matching instances of two design versions by their IDs and comparing all properties of each matched pair in BIM models. This method has been adopted by many studies [17,18] and by most commercial BIM platforms, such as Autodesk Revit, Navisworks, and Graphisoft ArchiCAD [20]. However, as mentioned above, this method may detect meaningless change results and is sensitive to the ID. To address the issue of this method, many studies

have proposed improved detection methods. Lee, G. et al. [26] proposed a flattening-based method that utilizes a recursive strategy to flatten the instances and then it compares the flattened instances. Daum, S. and Borrmann, A. [19] proposed an approach for detecting equivalences in IFC datasets based on a geometrical comparison first and a value comparison second. Shi, X. et al. [20] proposed a content-based automatic comparison approach for IFC files that constructs a hierarchical structure for both files and then uses an iterative bottom-up procedure to compare them. Shafiqa, M. T. and Lockleyb, S. R. [27] proposed a signature-based model comparison approach for IFC models that uses instance characteristics to define the corresponding signatures, which can be used to establish candidates for comparison. However, these methods are suffering from the fact that their detection results also contain meaningless changes, and they are usually time consuming.

Next, this section will first describe the current classification of design changes, and then discuss two types of design change detection methods, and finally summary the defects of current detection methods.

2.1 Classification of Design Changes

Design changes usually involve two files (documents); this research calls them File A (the old file) and File B (the new file). Currently, design changes are classified into three types by most studies: added (newly created), modified (revised), and deleted (removed) changes [1,19,23,26,28,29]. This research summarizes their definitions as follows.

- Added. An instance exists only in File B.
- Deleted. An instance exists only in File A.
- Modified. An instance exists in Files A and B, but they are not equal (some properties changed).

2.2 Detection of Design Changes

According to the current classification of design changes, this research groups detection methods into two categories based on the order in which they execute the following two processes.

- Matching. Determine whether an instance in File A also exists in File B or vice versa. This process usually uses the ID of an instance to find its “identical” instance in another file. That is, matched instances exist in both Files A and B, while unmatched instances exist in only one file.
- Comparison. Determine whether two instances are equal or whether there are two equal instances exist in both Files A and B. This process usually compares all the properties of instances (except ID).

Therefore, the two categories of detection methods are as follows:

- Matching-first method. The matching process is executed first and then comparison is performed.
- Comparison-first method. The comparison process is executed first and then matching (optional) is performed.

Most of the time, ID (e.g., GUID in IFC or ElementId in Revit) is the only property used for matching, so we define matching as matching two instances by ID in this research. Some studies may call the comparison of two instance properties other than ID "matching" but that process is called comparison in this research.

2.2.1 Matching-first Method

Figure 1 shows the flowchart of the matching-first method for design change detection. In this figure, files that "inst1" and "inst2" are from can be switched so that the output "Deleted" can be changed to "Added", that is, if all the File A (B) are changed to File B (A) in the flowchart, the output "Deleted" should be "Added". This rule also applies to flowcharts below.

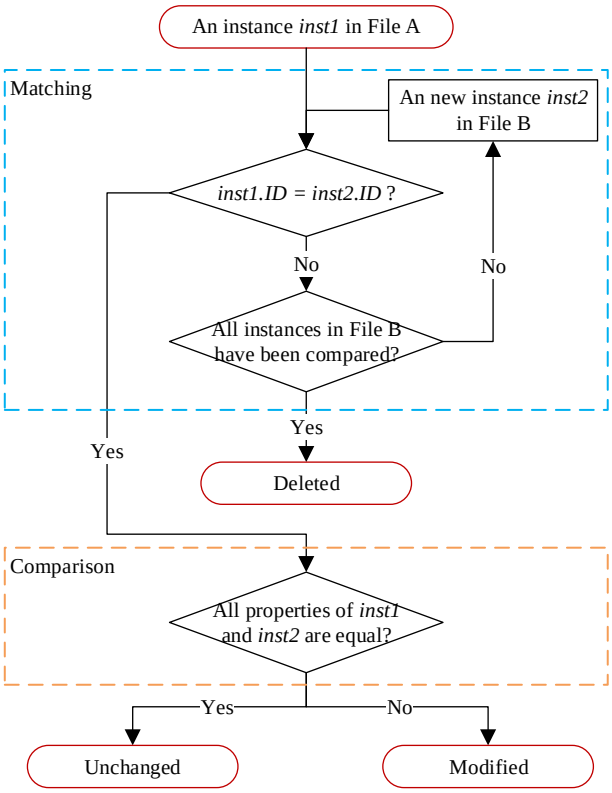


Figure 1. Flowchart of the matching-first method for design change detection.

The matching-first method is the most commonly used method for detecting design changes between two BIM models; its comparison process usually compares all the properties (except ID) of each instance. For convenience in the subsequent analysis, this research defines two types of detection errors as follows.

- Major error. A changed (added/deleted/modified) instance is detected as unchanged, or an unchanged instance is detected as changed.
- Minor error. A changed instance is detected as a different change type (e.g., a modified instance is detected as added).

The advantages of the matching-first method are simple and fast, but it is sensitive to ID and may yield meaningless results. Regarding the sensitivity to ID, an unexpected change of ID (which is called unreliable) can occur when copying models, saving as, work sharing and syncing in Revit [30]; once this happens, the matching-first method will no longer match any instances in Files A and B. Thus, even two instances whose properties (except ID) are all equal will be detected as deleted and added. Consequently, when the ID is unreliable, the matching-first method will result in many major errors.

Meaningless results may result from the matching-first method regardless of whether the ID is reliable. Consider an exchange of locations for two identical columns as an example. If all properties (except ID) of these two columns are equal, then this exchange should not be considered a change because it is meaningless from a design viewpoint. However, the matching-first method will detect it as a change of the locations of the two columns. Thus, the matching-first method may result in major errors regardless of whether the ID is reliable.

2.2.2 Comparison-first Method

Figure 2 shows the flowchart of the comparison-first method for design change detection.

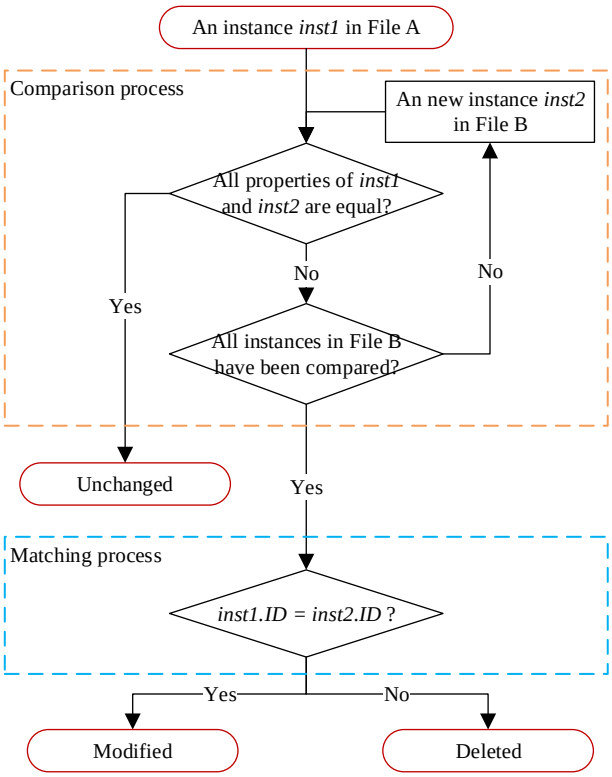


Figure 2. Flowchart of the comparison-first method for design change detection.

To improve the detection correctness of the matching-first method, many studies have proposed new methods, most of which are comparison-first methods. In fact, the flattening-based method [26], geometry-based method [19], content-based method [20], and signature-based method [27] mentioned above are all comparison-first method without matching processes. The former two methods consider some semantics in the comparison process; thus, they avoid comparing IDs

in IFC files. The latter two methods include some measures to accelerate the comparison process; specifically, the third method constructs two hierarchical structures for two IFC files and uses an iterative bottom-up procedure to compare them, while the fourth method uses a hash function to calculate signatures for each instance and then uses the signatures to establish candidates for comparison. Note that signatures (or fingerprints) have been widely used in instance detection and classification[31], and it is a simplified representation that often carries the most important information of an instance.

The comparison-first method first uses a comparison process to classify instances as changed or unchanged, which can avoid major errors. Therefore, the example of exchanging the locations of two identical columns will not be detected as a change by this method. However, most current comparison-first methods have no matching process, which may result in minor errors. For example, if the geometry of an instance has been modified and its ID is not changed, the current geometry-based (comparison-first) method may detect this as deleted and added changes, but actually, it should be a modified change. In fact, when the ID is reliable, ID can be used to identify a changed instance in File B was new or modified from File A, which is the only way to eliminate minor errors. Thus, if the matching process is executed after the comparison process, minor errors can be eliminated immediately when the ID is reliable.

A significant defect of the comparison-first method is that it is time consuming. The comparison process usually consumes much more time than matching because instances in BIM models possess many properties, such as parameters, location, geometry, etc., making the comparison-first method much slower. This is an important reason why this method is not commonly used.

2.3 Summary

To illustrate the difference of detection results between the matching-first and comparison-first methods clearly, Figure 3 shows a typical design change in a BIM model and the detection results of these two methods. In this figure, the ID and other properties have blue and yellow backgrounds, respectively; C, G, and L are the category, geometry, and location, respectively. In Figure 3, the detection result of the comparison-first method is correct, but the result of the matching-first method has some errors.

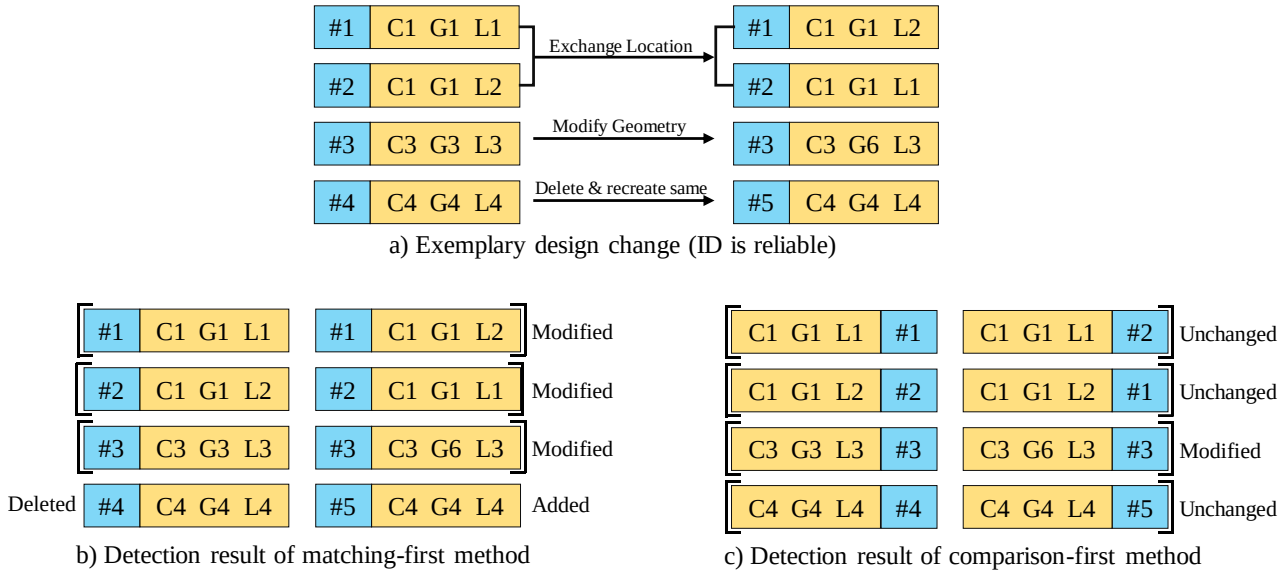


Figure 3. Detection results of the matching-first and comparison-first methods for a typical change.

To sum up the detection performances of these two types of methods, Table 1 shows the detection correctness and time consumption of them. It can be seen from Table 1 that the comparison-first method is the only one can achieve high correctness in all cases, but it has the lowest efficiency. Actually, minor errors are unavoidable when the ID is unreliable; consequently, in this research, these unavoidable minor errors will be ignored when considering the correctness of a detection method, so the comparison-first method is considered correct. However, because the current comparison-first method is based on the classification criterion (added/deleted/modified) which considers only syntactic changes, it can only obtain syntactic correctness but cannot deal with semantics. Therefore, this method will still detect meaningless design change results when used in practice. For example, in IFC, the geometric expression of an instance can be changed from a solid model to a surface model while maintaining the meaning, and the local coordinate system and the placement of an instance can be changed simultaneously while keeping its global location fixed, however, these two example changes cannot be identified semantically by the current comparison-first method. Additionally, the change of an instance property which is meaningless for designers should not be regarded as a semantic design change, and it cannot be handled correctly by the current comparison-first method.

In summary, the definition of semantic design changes is not clear, and is always mixed with data changes. The matching-first method regards data changes as design changes, and it results in wrong and meaningless results. The comparison-first methods proposed in other researches addresses the issue of ID-sensitivity and considers some semantics of instances, but they still cannot categorize design changes and data changes correctly; thus, meaningless design changes still exist in its detection results. Additionally, the time consumption of the comparison-first method is high.

Table 1. Detection performances of the matching-first and comparison-first methods.

		M	*C* (no m)	*C*
ID is reliable	Major error	X	O	O
	Minor error	O	X	O
ID is unreliable	Major error	X	O	O
	Minor error	X	X	X
Time consumption		Low	High	High

Note. *M* is the Matching-first method; *C* (no m) is the Comparison-first method (without the matching process); O means the method can eliminate this type of error, and X means cannot eliminate.

3 Research Methodology

The objective of this research is to detect semantic design changes with high correctness and efficiency, and a five-step methodology is utilized (Figure 4). The methodology consists of five parts: review, semantic change classification, detection method, implementation, and validation. Each part is introduced as follows.

In the review of related works, we concluded that the unclear definition of design changes is the reason for meaningless detection results. Therefore, although the current comparison-first methods address some issues in the matching-first method, they may still detect meaningless results, because the classification criterion for design changes they adopt does not consider the semantics in BIM models. Additionally, the time consumption of the current comparison-first methods is high.

To address these issues and achieve our objectives, we focus on two aspects: 1) classification, which provides the definition of semantic design changes and also works as a criterion and guide for design change detection, and 2) detection, which is a method used to identify changes according to the classification. First, we propose a semantic classification of design changes that divides data changes into three categories and classifies design changes into three levels from a design viewpoint. Next, we improve the comparison-first method by considering the proposed classification criterion, thus enabling the detection of semantic design changes. Finally, to further improve the efficiency, we adopt hash codes of instances to accelerate the comparison process of the detection method, and the time complexities of our method and previously proposed methods are also analyzed.

To verify the performance of the proposed method, this research implements our method and previously proposed methods in Revit. At the beginning, all the implemented methods are tested with manually created design changes in multiple BIM models (which include almost all types of changes and have different complexity levels), and their efficiency and correctness are compared. Meanwhile, detection performances of the proposed method and invited designers are also measured and compared to further show its benefit. First, a group of junior designers was invited to make design changes

in a BIM model individually, the modified models and documents recording design changes are saved in two different repositories. Then, each designer randomly chooses one model (except for the one he/she created) from the model repository and try his/her best to find the design changes manually. Finally, our method is utilized to detect design changes automatically, and the performance of our method and the designers is compared.

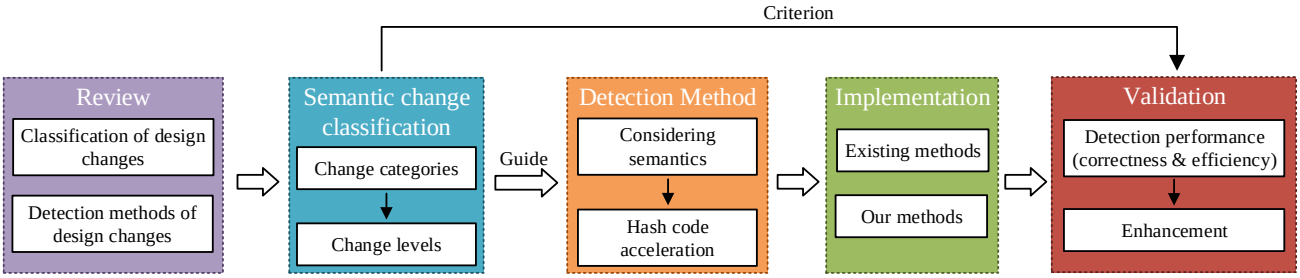


Figure 4. Overview of the research methodology.

4 Semantic Classification of Design Changes

As mentioned above, design changes and data changes are mixed used currently and what kind of changes are valuable or semantic for designers is not clear. Therefore, this research will first propose a semantic classification of design changes [32] to illustrate and define the semantic design changes. First, the data changes are divided into three categories; then, the design changes are classified into three levels from a design viewpoint to identify the semantic design changes in BIM models.

4.1 Categories of Data Changes

As two different model schemas, both IFC and Revit data models are based on the object-oriented modeling method, representing the same design data of a building or facility in different ways. In either of these two models, data of building elements can be described by instances (objects) that possess many properties. Generally, for each instance of a BIM model, its data information can be divided into three categories: 1) the characteristics and specifications of an instance such as the name, material, size, etc., which is called property data in this research; 2) the information shows the location, shape, geometry, etc. of an instance, which is called appearance data; 3) the information to connect other instances, such as the level of an instance, the relationship of a wall and a door, etc., which is called relationship data. Accordingly, we can classify the data changes in each category. A summary of the data changes in each category is shown in Table 2.

Property data represents instance properties, such as parameters or other user-specified attributes. In IFC, an IfcProperty can be added to or deleted from an IfcPropertySet; the value of an IfcProperty can be modified; the name of a user-defined IfcProperty can be modified; and the order of IfcProperty in an IfcPropertySet can be changed.

Appearance data represents the 3D appearances of instances, such as their geometry and location. In IFC, the attribute

Representations in IfcProductRepresentation is a list of IfcRepresentation (including shape representation), where each member defines a valid representation of a particular type within a particular representation context. Therefore, instance geometry can be added, deleted, and modified via IfcRepresentation. The representation method of an IfcRepresentation can be changed by changing the subtype of an IfcRepresentationItem in Items of the IfcRepresentation, e.g., from IfcFaceBasedSurfaceModel to IfcSolidModel. Additionally, the transformation property of an IfcProduct can be modified by its ObjectPlacement.

Relationship data represents the relationships between two or more instances, such as the relationship between a door and a wall. In IFC, instance relationships are represented by IfcRelationship, which connects instances through RelatingObject and/or RelatedObjects attributes. Classes such as IfcBuildingElement have many attributes defined by a set of IfcRelationship (or its subtype), e.g., IsDecomposedBy, HasAssociations, and IsDefinedBy. Relationships can be added, deleted, and modified in these attributes, and there are two types of modifications: modifying the related instance or the relationship itself.

Table 2. Data changes in each category

Category	Data Change	Description	Example in IFC
Property data	Added	Add a new instance property	Add an IfcProperty in HasProperties of an IfcPropertySet
	Deleted	Delete an existing instance property	Delete an IfcProperty in HasProperties of an IfcPropertySet
	Value Modified	Modify a property's value	Modify the NominalValue of an IfcPropertySingleValue
	Name Modified	Modify a property's name	Modify the Name of an IfcProperty
	Order Changed	Change the order of instance properties	Change the order of IfcProperty in HasProperties of an IfcPropertySet
Appearance data	Added	Add a new instance geometry	Add an IfcRepresentation in Representations of an IfcProductRepresentation (e.g., Representation of IfcWall)
	Deleted	Delete an existing instance geometry	Delete an IfcRepresentation in Representations of an IfcProductRepresentation
	Geometry Modified	Modify the geometric shape of an instance	Change the IfcRepresentation in Representations of an IfcProductRepresentation
	Transformation Modified	Modify the transformation property of an instance, such as translation or rotation	Change the IfcObjectPlacement in ObjectPlacement of an IfcProduct
	Representation Modified	Change the geometric representation, such as converting a solid model to a surface model	Change the subtype of an IfcRepresentationItem in Items of an IfcRepresentation
Relationship data	Added	Add a new relationship between two instances	Add an IfcObject in RelatedObjects of an IfcRelDecomposes
	Deleted	Delete an existing relationship between two instances	Delete an IfcObject in RelatedObjects of an IfcRelDecomposes
	Instance Modified	Modify a related instance	Modify the data of an IfcObject in RelatedObjects of an IfcRelDecomposes
	Relationship Modified	Modify a relationship between two instances (e.g., change the relationship between A& B to A& C)	Change an IfcObject in RelatedObjects of an IfcRelDecomposes

4.2 Levels of Design Changes

As mentioned above, data changes are insufficient to be regarded as design changes. Therefore, we propose a three-

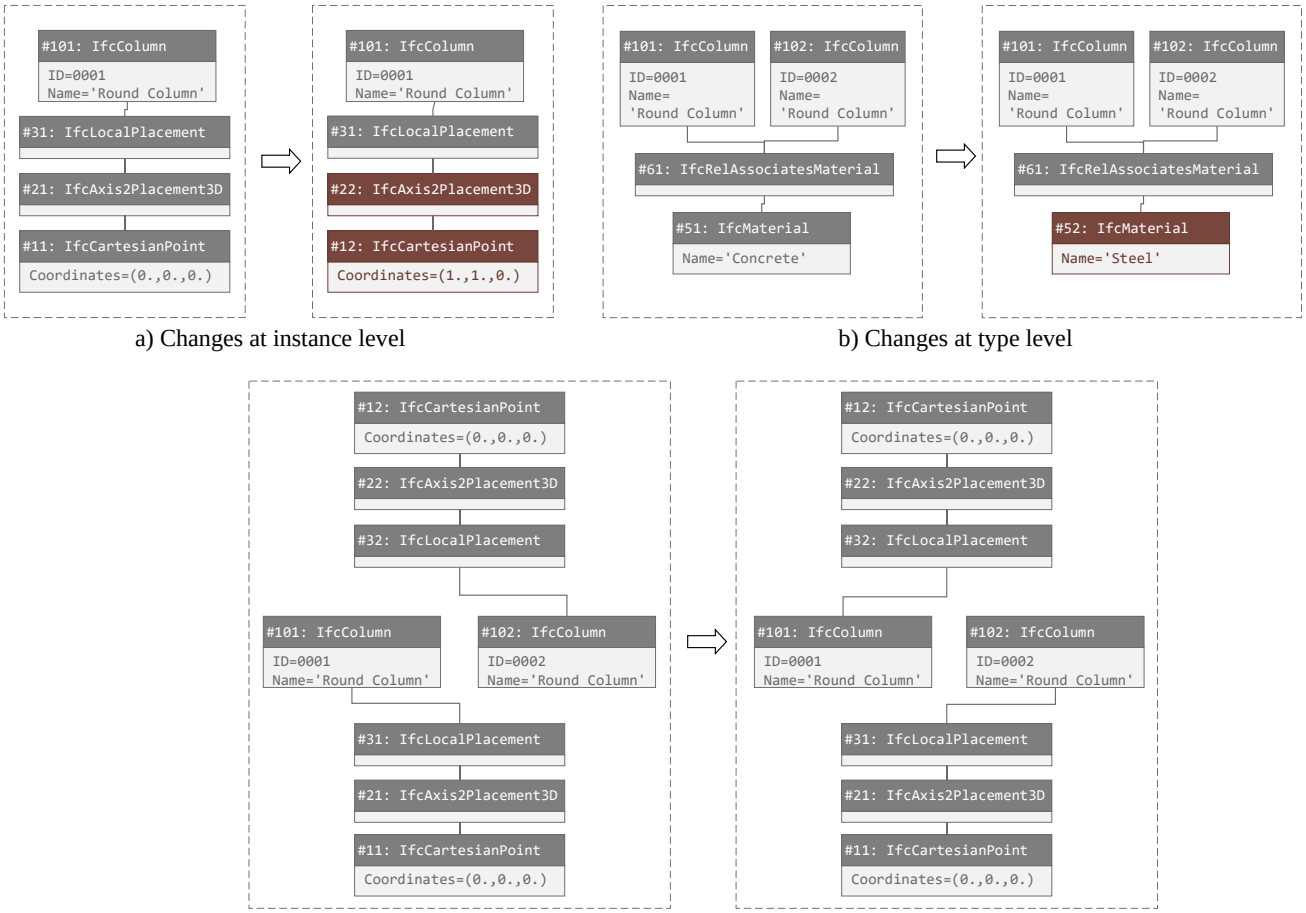
level semantic classification of design changes (example changes of each level based on IFC and Revit are shown in Figure 5 and Figure 6 respectively).

First, all data changes should be semantically identified at the instance level as discussed in Section 4.1.

Next, from the perspective of the type level, changes in instances may be caused by changes in a type. For example, the geometric changes of multiple instances may be caused by a geometry change of the type from which these instances inherit. To optimize the detection result, if changes in some instances are caused by changes in a type, the source of the change should be identified, and the changed instances should be grouped by the type.

Finally, from the perspective of the model level, changes in instances may be meaningless. For example, exchanging the locations of two identical columns should not be detected as a change because the model is the same before and after the change from a designer's viewpoint. Such meaningless changes should not be flagged as changes. In addition, similar to the type level, instance modifications in relationship data should be detected at the model level to identify the source of the change. For example, modifying the elevation of a level may result in many changes in the related instances; therefore, the source of the change (i.e., the modified elevation) should be identified, and the related instances should be grouped by that change.

In summary, this research defines the semantic design change as follows: a change is regarded as a semantic design change if and only if the change is semantic at all levels, namely, instance level, type level, and model level. It can be seen, data changes can only involve semantic changes at the instance level, such changes do not involve semantics at the higher two levels. Therefore, we can conclude that the semantic classification criterion for the detection method involves identifying semantic changes at three levels: the instance level (where changes are data changes), the type level, and the model level.

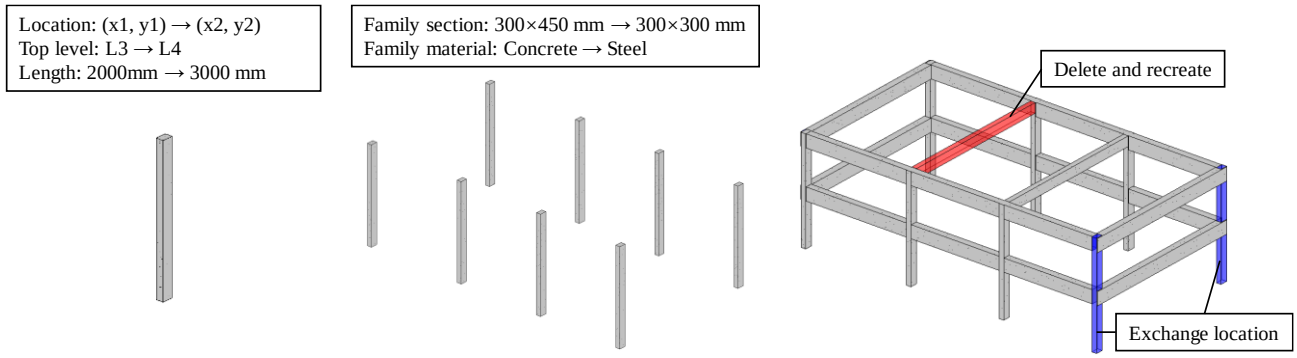


a) Changes at instance level

b) Changes at type level

c) Meaningless changes at model level

Figure 5. Example changes at three levels in IFC



a) Changes at instance level

b) Changes at type level

c) Meaningless changes at model level

Figure 6. Example changes at three levels in Revit

5 Detection of Semantic Design Changes

5.1 Improved Method by Considering Semantics of Design Changes

Detecting semantic design changes can be achieved by detecting design changes under the semantic classification criterion, which can be obtained via the comparison-first method considering the following requirements as described

below.

1. Meaningful properties

When comparing property data, only the meaningful properties should be compared; meaningless properties should be ignored. Some properties of an instance are meaningless for designers, such as the IfcOwnerHistoryInfo. Therefore, a change in these properties should not be regarded as an instance semantic change, and such properties should be ignored during comparison of two instances. In addition, properties that can be derived (or inferred) from other properties should be ignored during the comparison. In this research, the above two kinds of properties are called meaningless properties; the remaining properties of an instance are called meaningful properties. Thus, when comparing two instances, only the meaningful properties should be compared.

2. Order of properties

When comparing property data, this research assumes that the order change in property data should not be regarded as a semantic change. Some properties are collections of many other properties, such as the IfcPropertySet in IFC and the Parameters in Revit. Therefore, a detection method should compare these property sets regardless of their orders because the order is meaningless.

3. Semantics of geometry

When comparing appearance data, the geometry of instances should be compared semantically. In BIM models, the geometry of an instance can be represented in different ways [33,34]. For example, the geometry of an instance can be represented as sweeping, constructive solid geometry (CSG), or B-Rep in IFC. Therefore, geometries with different syntactic data should be compared by their semantic meanings.

4. Reference ID

When comparing relationship data, the related instances should be compared rather than the reference IDs. Some properties are references to other instances, such as the STEP ID (e.g., “#124”) in IFC [35] and the LevelId in Revit. The instances associated with these properties are semantic, but their reference IDs are not. Thus, when comparing these properties between two instances, the instances referred to by the properties should be compared instead of the properties themselves.

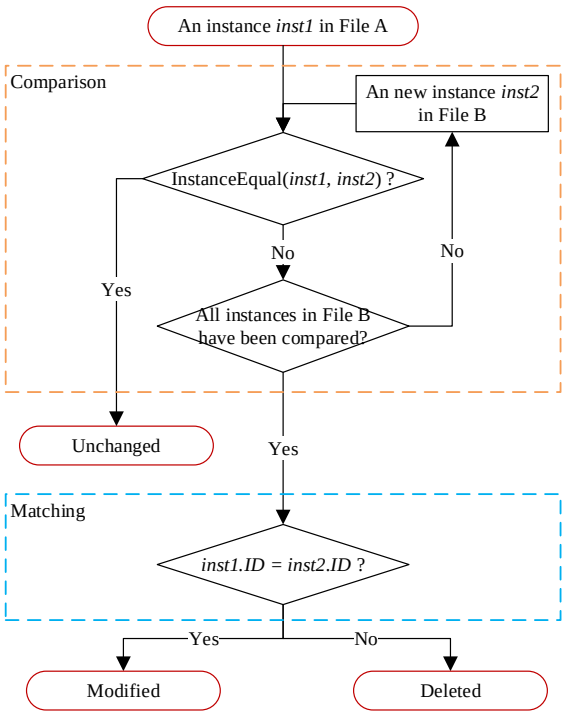
5. Changes in types

Changes in instances may be caused by changes in a type; these changes occur at the type level. In IFC, one approach for detecting changes at the type level is using the RelatingObject and/or RelatedObjects attributes of IfcRelationship. If changes occur to an instance that is related to many other instances, the changes in the related instances caused by this changed instance should be grouped by this instance. However, detecting changes at the type level is difficult in Revit

because types (family types) are not stored as instances (elements). An alternative to detecting changes at the type level is to list all the categories of changed instances in a hierarchical view allowing users to view any single category's changed instances.

After considering all the requirements above, the comparison-first method can detect design changes under the semantic classification criterion. The flowchart of this improved comparison-first method for semantic design change detection is shown in Figure 7, and the InstanceEqual function used in Figure 7a is shown in Figure 7b. In Figure 7b, the InstanceEqual function takes only the meaningful properties of two instances and then compares the equality of these properties semantically, which includes ignoring the order of properties, comparing geometries in a unified data format, and comparing the referred instances instead of the reference properties. Note that change in types is not considered in Figure 7b because it has an alternative to detect.

So far, only one significant disadvantage remains for the comparison-first method: its high time consumption. This issue will be addressed in the following section.



a) Comparison-first method (improved)

Algorithm 1 Compare the semantic equality of two instances

Input: two instances *inst1, inst2*
Output: whether *inst1* is equal to *inst2*

```
function InstanceEqual(inst1, inst2)
    props1, props2 ← all meaningful properties of inst1, inst2 in the same order
    if props1.length ≠ props2.length
        return false
    end if
    for i = 1 to props1.length
        if not PropertyEqual(props1[i], props2[i])
            return false
        end if
    end for
    return true
end function

function PropertyEqual(prop1, prop2)
    if typeof(prop1) ≠ typeof(prop2)
        return false
    end if
    if prop1 is a geometry property
        geom1, geom2 ← geometries of prop1, prop2 in a unified data format
        return geom1 == geom2
    else if prop1 is a reference property
        rinst1, rinst2 ← referred instances of prop1, prop2
        return InstanceEqual(rinst1, rinst2)
    else
        return prop1 == prop2
    end if
end function
```

b) Algorithm of comparing semantic equality of instances used in a)

Figure 7. Flowchart of the improved comparison-first method for semantic design change detection.

5.2 Acceleration Based on Hash Code

A hash function is any function that can be used to map data of arbitrary size onto data of a fixed size [36]. The values returned by a hash function are called hash codes. An important feature of hash functions is that the same data will be

assigned the same hash code; nevertheless, having the same hash code does not mean that the original data are the same (but they likely are). Different data with the same hash code forms a hash collision. A good hash function will have a very low probability of hash collisions.

The time consumption of the comparison process during detection is directly proportional to the number of times of data (e.g., properties) reading. However, if we calculate hash codes for all instances and use those to filter out unequal instances in the comparison process, the time consumption can be greatly reduced; subsequently, the meaningful properties of two instances need to be read and compared only when their hash codes are equal. Moreover, if the probability of hash collisions is zero or very low, we can directly assert that two instances are equal if their hash codes are equal, which further reduces the time consumption.

To detect semantic design changes, the key point is that the hash code of an instance should correspond to the semantic classification criterion. Therefore, the hash code of an instance should change if and only if that instance is changed semantically, which also implies that the result of comparing two instances' hash codes (ignoring hash collision) should be identical to that produced using the comparison process proposed in Section 5.1. Generally, instance hash code calculation involves calculating the hash codes for all the meaningful properties of the instance and then combining them. Therefore, only hash codes of the meaningful properties of an instance should be calculated; the property set order should be ignored; geometries with the same semantics should have the same hash code; and the reference to another instance should be replaced by the hash code of the referenced instance. Meanwhile, hash codes of different properties of a single instance can be combined into one instance hash code using a parity-preserving operator such as XOR [37]. In this way, the order change of properties is omitted automatically, thus creating a one-shot solution for comparing two instances. The algorithm of calculating the hash code of an instance considering semantics proposed in this research is shown in Figure 8.

Figure 9 shows the flowcharts of the hash-code-accelerated (improved comparison-first) methods proposed in this research for semantic design change detection. In this figure, hash codes of all instances in both File A and B have been calculated previously by the algorithm in Figure 8. In Figure 9a, the InstanceEqual function is also the Algorithm 1 shown in Figure 7b, and the "No" with red color means the occurrence of the hash collision. In Figure 9b, the quick hash-code-accelerated method does not include the comparison process because it assumes that the probability of hash collisions is zero.

Algorithm 2 Calculate the hash code of an instance considering semantics

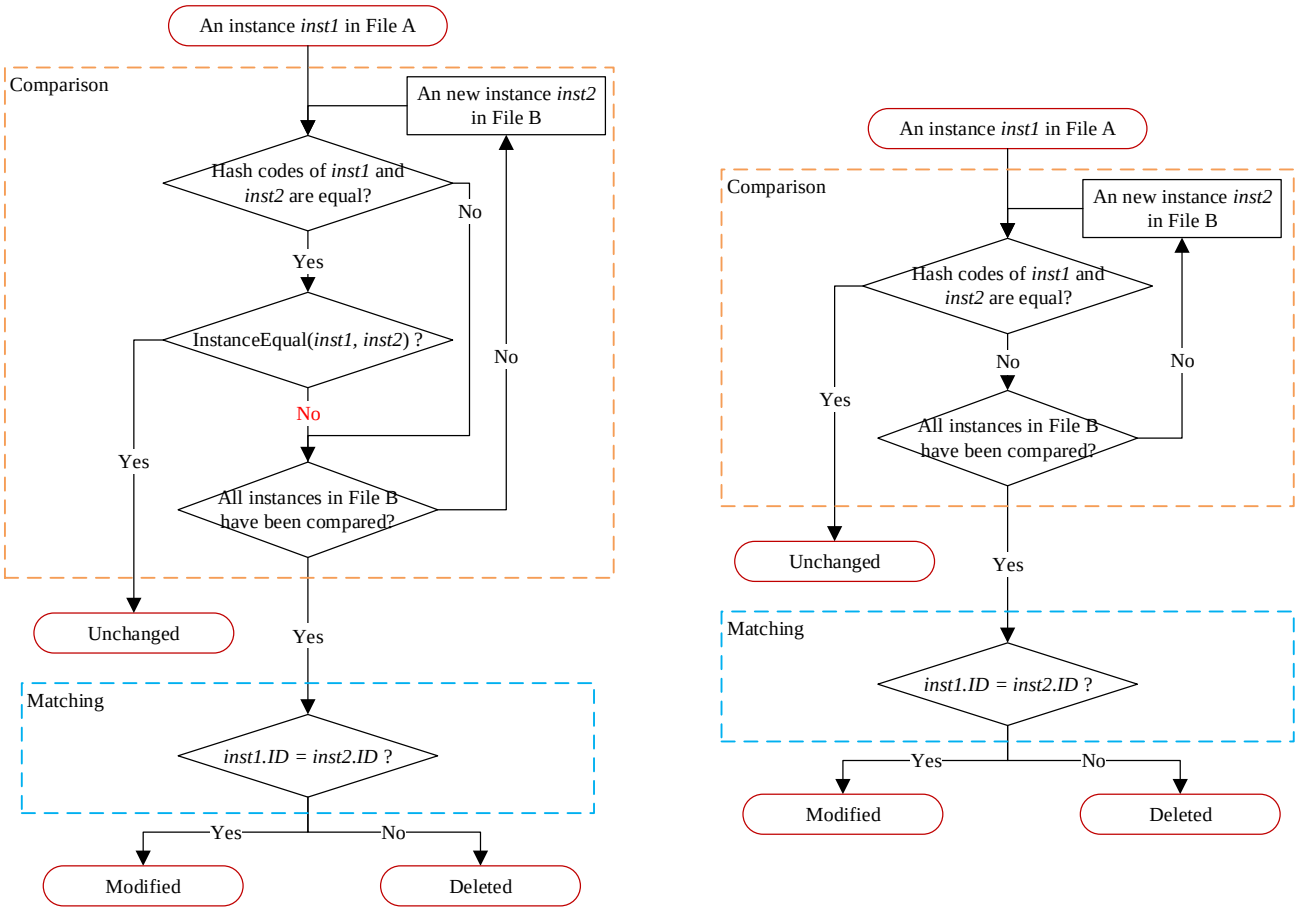
Input: an instance *inst1*

Output: the hash code of *inst1*

```
function GetInstanceHashCode(inst1)
    hashcode ← 0
    props1 ← all meaningful properties of inst1
    for i = 1 to props1.length
        hashcode ← hashcode XOR GetPropertyHashCode(props1[i])
    end for
    return hashcode
end function

function GetPropertyHashCode(prop1)
    if prop1 is a geometry property
        hashcode ← hash code of geometry of prop1 in a unified data format
    else if prop1 is a reference property
        hashcode ← hash code of the referred instances of prop1
    else
        hashcode ← hash code of prop1
    end if
    return hashcode
end function
```

Figure 8. Algorithm of calculating the hash code of an instance considering semantics.



a) Hash-code-accelerated method

b) Quick hash-code-accelerated method

Figure 9. Flowcharts of the hash-code-accelerated (improved comparison-first) methods for semantic design change detection.

5.3 Time Complexity Analysis

For convenience, this research assumes that the total number of instances in both Files A and B is N and that the time consumption of comparing two instances' equality is 1 s, while comparing the IDs or hash codes of two instances requires $1/m$ s.

The matching-first method executes the matching process $O(N^2)$ times and the comparison process $O(N)$ times; thus, its total time cost is $O\left(\frac{N^2}{m}+N\right)=O\left(N\left(1+\frac{N}{m}\right)\right)$. In our test, m is approximately 56000, which is much larger than N; therefore, it can be assumed that $\frac{N}{m}<1$. Thus, the time complexity of the matching-first method is $O(N)$. However, it should be noted that this method is infeasible when the ID is unreliable because its detection result contains both major and minor errors. Therefore, its time complexity is only applicable when the ID is reliable.

The comparison-first method executes the matching process $O(N)$ times and the comparison process $O(N^2)$ times, but the latter can be reduced to $O(N)$ in one case. When the ID is reliable and the change is small, as shown in Figure 7, almost every first comparison of two instances will return "Yes" because the order of instance comparisons is usually based on the ID. Therefore, the execution times of the comparison process will be reduced to $O(N)$ in this case, and the total time consumption of the comparison-first method will be $O\left(N+\frac{N}{m}\right)=O(N)$. However, when the ID is unreliable or the change is large, the execution times of the comparison process are $O(N^2)$ and cannot be reduced. In this case, the total time consumption of the comparison-first method will be $O\left(N^2+\frac{N}{m}\right)=O(N^2)$.

For the hash-code-accelerated method, the equality of two instances will be compared if and only if their hash codes are equal. Considering the probability of hash collisions is very low, almost all unequal instances are filtered out before the equality comparison occurs. Therefore, its execution times of the comparison are always $O(N)$. Because the time consumption of calculating the hash codes is also $O(N)$, the total time consumption of the hash-code-accelerated method is $O\left(N+N+\frac{N}{m}\right)=O(N)$. As for the quick hash-code-accelerated method, it does not compare the equality of two instances, so its total time consumption is $O\left(N+\frac{N}{m}\right)=O(N)$.

Table 3 summarizes the time complexities of the four methods in two conditions and shows that the two hash-code-accelerated methods perform best overall. Actually, the quick hash-code-accelerated method is faster than the hash-code-accelerated method because the former does not double-check the equality of two instances. In addition, when the ID is reliable, the time consumption of the quick hash-code-accelerated method and the comparison-first method may be close because both will read the meaningful properties of all instances at once—the hash-code-accelerated method may even be slowest in this case.

A user-friendly method exists to eliminate any possible hash collisions in the quick hash-code-accelerated method, that

is, executing the hash-code-accelerated method to check the detection result in the background or another thread after the quick hash-code-accelerated method is complete. Since the hash codes have been calculated by the quick hash-code-accelerated method, the double-checking process can save that calculation time. Finally, the total time consumption of this method is close to that of the hash-code-accelerated method, but the user can obtain the detection result as soon as the quick hash-code-accelerated method is complete.

Table 3. Time complexities of the four methods in two conditions.

Condition	Time Complexity			
	M	*C*	*Hc*	*Qhc*
ID is reliable and change is small	$O(N)$	$O(N)$	$O(N)$	$O(N)$
ID is unreliable or change is large	*N/A*	$O(N^2)$	$O(N)$	$O(N)$

Note. *M*, *C*, and *(Q)Hc* are the Matching-first, Comparison-first, and (Quick) Hash-code-accelerated methods, respectively.

6 Implementation and Results

Both the proposed method in this research and existing methods are implemented in Revit for testing purpose, and then 4 groups of BIM models are developed as a baseline dataset for validating the detection methods. Finally, performances of the proposed method and existing methods as well as invited designers are compared to highlight the advantages of our method.

6.1 Implementation of the Detection Methods

According to the four detection methods: matching-first, comparison-first, hash-code-accelerated, and quick hash-code-accelerated methods, this research develops four algorithms under the semantic classification criterion based on Revit API using the C# language. In this implementation, the four algorithms share the same function of instance comparison which considers the semantics of design changes. Thus, both the matching-first and comparison-first algorithms represent existing methods with an improved comparison process.

Figure 10 shows the core code listings for these four algorithms and the key differences between the last three are indicated by the red rectangles. The target of these algorithms is to classify all the element IDs into four lists: `addEIds`, `delEIds`, `modifyEIds`, and `unchangedEIds`; these four lists represent added, deleted, modified, and unchanged elements, respectively. In Figure 10 c) and 10d), the two hash-code-accelerated algorithms compare the hash codes of two elements by using `EIdHashCodes1/2`, which stores the precalculated hash codes for all the elements in one file, and `eIdHc1/2.Value` represents the hash code of an element in File A/B.

```
//List<Element> Eles1/2: all elements in File A/B, ordered by Id.
int i = 0, j = 0, eId1i, eId2j;
while (i < Eles1.Count && j < Eles2.Count)
{
    eId1i=Eles1[i].Id.IntegerValue;eId2j=Eles2[j].Id.IntegerValue;
    if (eId1i < eId2j)
    { delEIds.Add(eId1i); i++; }
    else if (eId1i == eId2j)
    {
        if (eleEqual(Eles1[i], Eles2[j])) unchangedEIds.Add(eId2j);
        else modifyEIds.Add(eId2j);
        i++; j++;
    }
    else //if(eId1 > eId2)
    { addEIds.Add(eId2j); j++; }
}
while (i < Eles1.Count)
{ delEIds.Add(Eles1[i].Id.IntegerValue); i++; }
while (j < Eles2.Count)
{ addEIds.Add(Eles2[j].Id.IntegerValue); j++; }
```

a) Matching-first algorithm

```
//use eIdEles_ in foreach loop to use Remove()
var eIdEles1_ = new Dictionary<int, Element>(EIdEles1);
var eIdEles2_ = new Dictionary<int, Element>(EIdEles2);

foreach (var ee1 in EIdEles1)
    foreach (var ee2 in eIdEles2_)
        if (eleEqual(ee1.Value, ee2.Value))
        {
            unchangedEIds.Add(ee2.Key);
            eIdEles1_.Remove(ee1.Key); eIdEles2_.Remove(ee2.Key);
            break;
        }
foreach (var ee1 in eIdEles1_)
{
    if (eIdEles2_.ContainsKey(ee1.Key))
    { modifyEIds.Add(ee1.Key); eIdEles2_.Remove(ee1.Key); }
    else
        delEIds.Add(ee1.Key);
}
foreach (var ee2 in eIdEles2_)
    addEIds.Add(ee2.Key);
```

b) Comparison-first algorithm

```
//EleHashCodes1/2 have been initialized and calculated previously
var eIdHCs1_ = new Dictionary<int, long>(EIdHashCodes1);
var eIdHCs2_ = new Dictionary<int, long>(EIdHashCodes2);

foreach (var eIdHc1 in EIdHashCodes1)
    foreach (var eIdHc2 in eIdHCs2_)
        if (eIdHc1.Value == eIdHc2.Value &&
            eleEqual(EIdEles1[eIdHc1.Key], EIdEles2[eIdHc2.Key]))
        {
            unchangedEIds.Add(eIdHc2.Key);
            eIdHCs1_.Remove(eIdHc1.Key);eIdHCs2_.Remove(eIdHc2.Key);
            break;
        }
foreach (var eIdHc1 in eIdHCs1_)
{
    if (eIdHCs2_.ContainsKey(eIdHc1.Key))
    { modifyEIds.Add(eIdHc1.Key); eIdHCs2_.Remove(eIdHc1.Key); }
    else
        delEIds.Add(eIdHc1.Key);
}
addEIds = eIdHCs2_.Select(idhc => idhc.Key).ToList();
```

c) hash-code-accelerated algorithm

```
//quick: assuming no hash collision will happen
var eIdHC1_ = new Dictionary<int, long>(EIdHashCodes1);
var eIdHC2_ = new Dictionary<int, long>(EIdHashCodes2);

foreach (var eIdHc1 in EIdHashCodes1)
    foreach (var eIdHc2 in eIdHC2_)
        if (eIdHc1.Value == eIdHc2.Value)
        {
            unchangedEIds.Add(eIdHc2.Key);
            eIdHC1_.Remove(eIdHc1.Key); eIdHC2_.Remove(eIdHc2.Key);
            break;
        }
foreach (var eIdHc1 in eIdHC1_)
{
    if (eIdHC2_.ContainsKey(eIdHc1.Key))
    { modifyEIds.Add(eIdHc1.Key); eIdHC2_.Remove(eIdHc1.Key); }
    else
        delEIds.Add(eIdHc1.Key);
}
addEIds = eIdHC2_.Select(idhc => idhc.Key).ToList();
```

d) Quick hash-code-accelerated algorithm

Figure 10. Core code listings for the four algorithms

The `eleEqual()` function in Figure 10 is used to judge whether all the meaningful properties of two instances are semantically equal. It selects 5 of the 22 element properties in Revit as the meaningful properties, as shown in Table 4. Note that, generally, the geometry and location of an instance need to be considered together. However, in Revit, the location can also store element geometric information such as LocationCurve, and considering location while ignoring the geometry of an element is sufficient to obtain accurate detection results. Therefore, Table 4 regards element geometry as a meaningless property for the convenience of implementation.

Finally, all these algorithms are integrated into a WPF application for Revit change detection. This application is used to test these algorithms in the next section. The alternative approach for detecting changes at the type level (as described in Section 5.1) is adopted in this application. Therefore, all the categories of changed instances will be listed in a tree view, allowing users to view any single category's changed elements.

Table 4. Meaningful and meaningless properties of an element in Revit

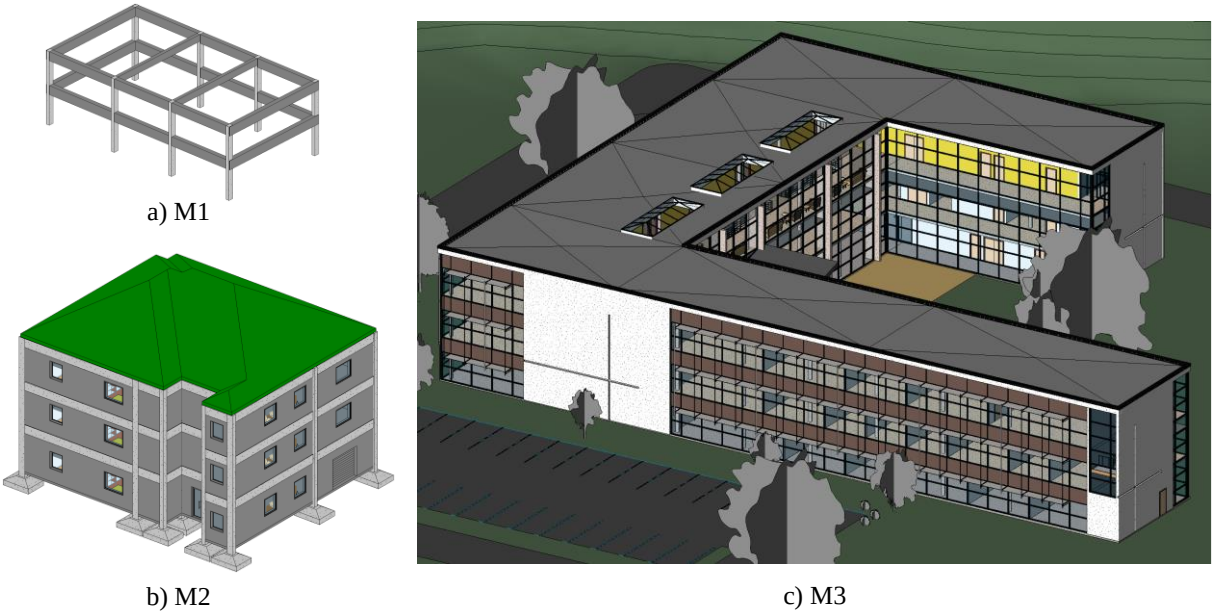
Meaningful properties	Meaningless properties

Category	Id	GroupId
Location	Geometry	IsTransient
Parameters	LevelId	IsValidObject
Name	AssemblyInstanceId	OwnerViewId
	BoundingBox	ParametersMap
	CreatedPhaseId	Pinned
	DemolishedPhaseId	UniqueId
	DesignOption	ViewSpecific
	Document	WorksetId

6.2 Developed BIM Models

Lin et al. [32] developed 11 exemplary BIM models, named M1, that can be used as test cases for validating detection methods. Because these BIM models are simple and contain only a few instances, this research develops two other groups of BIM models that are more complex and can also be used as test cases. All these three groups of Revit BIM models follow the same development rules; therefore, each group of BIM models has 11 models: one of them is the original model, and the other 10 are derived models that include only one type of change from the original model as defined in Section 4. The names of the original models of these three groups are M1, M2, and M3. M1 is a simple structure frame; M2 is an example house and contains architectural and structural building information; and M3 is a Revit sample project named rac_advanced_sample_project (http://www.autodesk.com/revit-rac-advanced-sample-project-2018-enu). Figure 11 shows screenshots of the three original BIM models. Table 5 shows the design change information of these BIM models. A repository containing these three groups of BIM models and their detailed descriptions was established on GitHub and can be found at https://github.com/Zhou-Yucheng/Design-Change-BIM-Models.

The architectural model of a classroom building of a primary school, named M4, is also used in the test to compare the detection performances of designers and the implemented methods. Figure 12 shows a screenshot of this model.



a) M1

b) M2

c) M3

Figure 11. Screenshots of the original BIM models M1, M2, and M3.

Table 5. Design change information of the three groups of BIM models M1, M2, and M3.

Model Name [a]	Category of Data Change [b]	Level of Design Change
Mx	-	-
Mx_All-A	Added in P, A and R	Instance
Mx_All-D	Deleted in P, A and R	Instance
Mx_All-DA(M) [c]	Deleted and then added in P, A and R	Model
Mx_A-MG	Geometry modified in A	Instance
Mx_A-MG(T)	Geometry modified in A	Type
Mx_A-ML	Location modified in A	Instance
Mx_A-ML(M) [c]	Location modified in A	Model
Mx_P-MV	Value modified in P	Instance
Mx_R-MI(M)	Instance modified in R	Model
Mx_R-MR	Relationship modified in R	Instance

[a] Mx denotes M1, M2, or M3.

[b] P, A, and R denote Property data, Appearance data, and Relationship data, respectively.

[c] These two models contain only meaningless changes (delete and recreate the same instances or exchange locations).

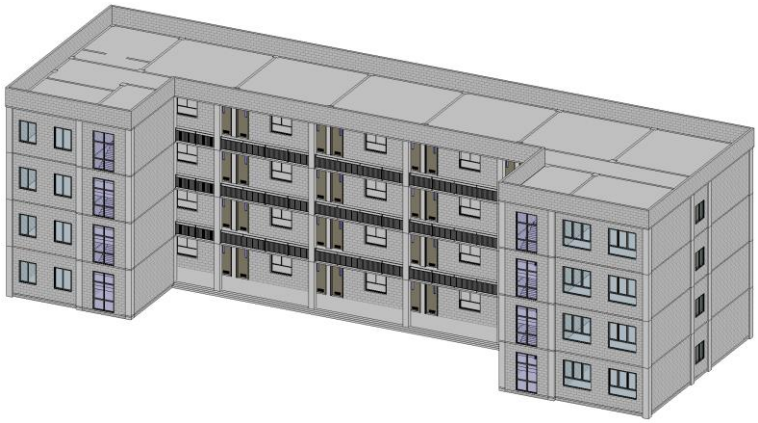


Figure 12. Screenshot of the BIM model M4.

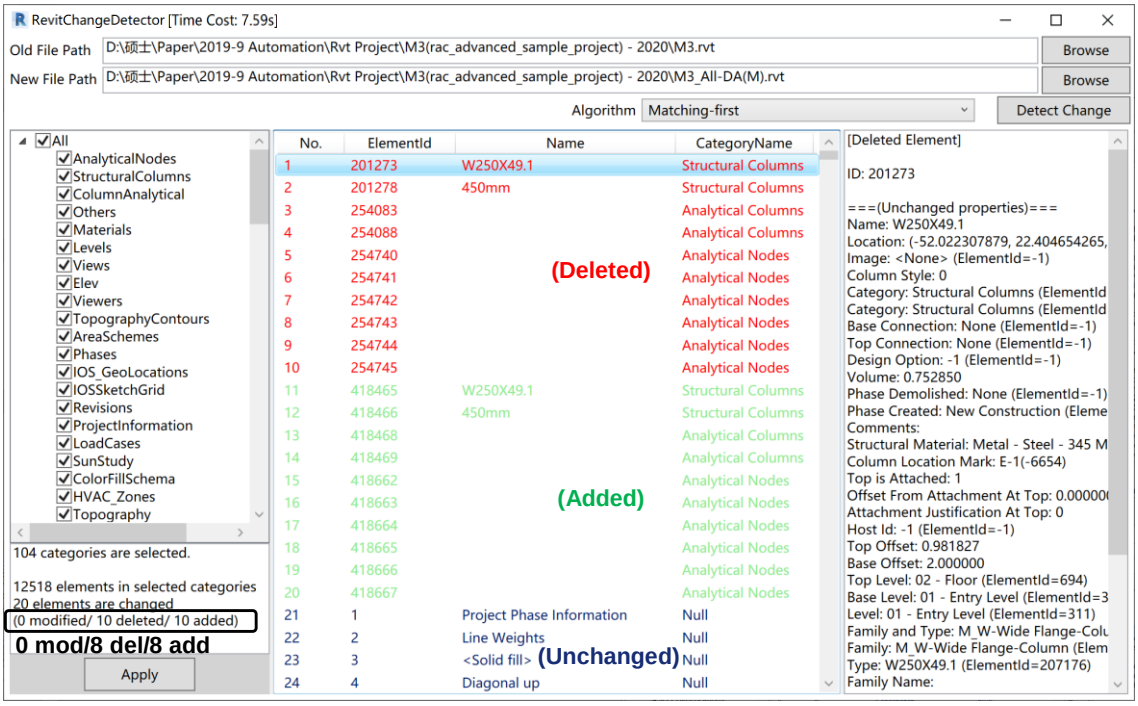
6.3 Performance Comparison

6.3.1 Proposed Method and Existing Methods

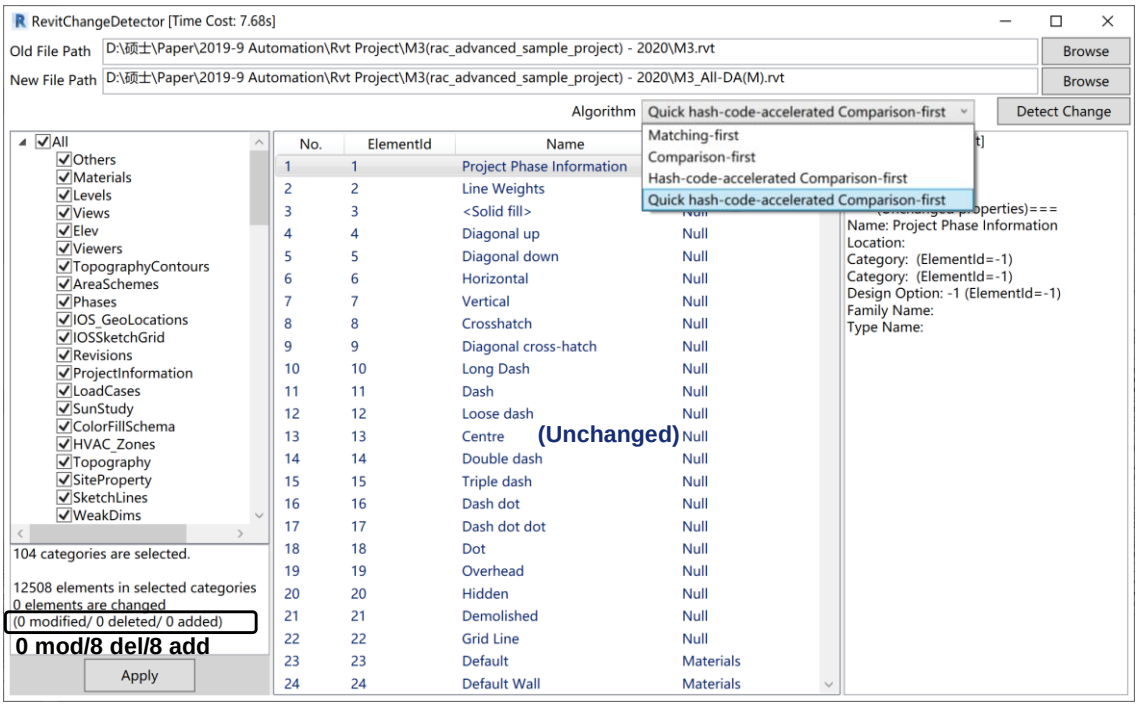
With the developed application, this research tests the performances (correctness & time consumption) of the four detection algorithms by detecting the changes between derived and original models in the first three groups of BIM models (M1–M3). Each test was performed twice in two different conditions: the ID is reliable or unreliable, where the ID of every element was changed randomly under the unreliable ID condition.

Figure 13 shows the detection results of M3_All-DA(M) when the ID is reliable. In Figure 13a, the matching-first algorithm detects all changes but includes meaningless changes. However, in Figure 13b, which shows the detection result of the other three algorithms (since their detection results are the same), the detection result is that all elements are unchanged, which is correct. Because the model M3_All-DA(M) (deletes some elements and recreates the same one) only contains meaningless changes. Moreover, it can be concluded that hash collisions do not occur in the two hash-code-accelerated algorithms because the detection result of the quick hash-code-accelerated algorithm is the same as those of the comparison-first and hash-code-accelerated algorithms.

When the ID is unreliable, the detection result of the matching-first algorithm is totally wrong, because almost all elements will be judged as changed. However, the detection results of the other three algorithms are still correct and are the same as shown in Figure 13b. However, in this condition, the time consumption of the comparison-first algorithm is high to the point where it is unacceptable, while the time consumed by the other two hash-code-accelerated algorithms is only slightly higher than under the reliable-ID condition.



a) Detection result of matching-first algorithm



b) Detection result of comparison-first, hash-code-accelerated and quick hash-code-accelerated algorithms

Figure 13. Screenshots of the results of the four algorithms for detecting M3_All-DA(M) when ID is reliable.

Table 6 summarizes the average time consumed by each of the four tested algorithms. The tests were performed on a laptop with an Intel i7-9850H CPU (2.60 to 4.60 GHz). As Table 6 shows, when the ID is reliable, the time consumption of the matching-first algorithm is the lowest, and the comparison-first algorithm consumes almost the same amount of time. The time consumption of the quick hash-code-accelerated algorithm is approximately 5%~14% larger than that of the

comparison-first algorithm, and the difference decreases as the number of elements increases (from M1 to M3). However, the time consumption of the hash-code-accelerated algorithm is approximately twice that of the comparison-first algorithm, and the difference increases with the number of elements.

When the ID is unreliable, the matching-first algorithm is not applicable. The time consumption of the comparison-first algorithm increases dramatically because its detection time complexity is $O(N^2)$ (where N is the number of elements in the models). When the element number is approximately 5000 or 13000, the time consumption increases by a factor of approximately 10 or 50, which is unacceptable. In contrast, the time consumed by the other two hash-code-accelerated algorithms is only 2%~5% larger than the time they consume when the ID is reliable.

Therefore, we can conclude that the time consumed by the two hash-code-accelerated algorithms is stable and ID-independent, while the time consumption of the comparison-first algorithm is sensitive to ID. When the ID is unreliable, the quick hash-code-accelerated algorithm can save up to $1 - 8.4/450.1 = 98.1\%$ of time compared to the comparison-first algorithm. Therefore, in most cases, the quick hash-code-accelerated algorithm is recommended. The hash-code-accelerated algorithm can be optionally executed as a double-check; in this way, the hash codes are calculated by the former, which saves time for the latter double-check process.

Table 6. Average time consumption of the four tested algorithms.

Condition	Model Group	Element Number	Average Time Consumption (s)			
			M	*C*	*Hc*	*Qhc*
ID is reliable	M1	2007	0.49	0.50	0.97	0.57
	M2	4869	1.9	2.0	3.8	2.2
	M3	12896	7.6	7.7	15.2	8.1
ID is unreliable	M1	2007	-	2.8	0.99	0.59
	M2	4869	-	21.2	4.0	2.3
	M3	12896	-	450.1	15.5	8.4

Note. *M*, *C*, and *(Q)Hc* are the Matching-first, Comparison-first, and (Quick) Hash-code-accelerated algorithms, respectively.

In addition to the above four algorithms, this research also tests IFCdiff [20]. As mentioned in section 2.2.2, IFCdiff is a software application used to detect design changes in IFC files using a content-based comparison-first method. Table 7 shows the detection performances of IFCdiff and that of the quick hash-code-accelerated algorithm. Note that the IFC files detected in IFCdiff are exported by the corresponding Revit files in Revit.

To measure the detection correctness, we adopt the similarity rate ($= N_{A\cap B}/N_B$, where $N_{A\cap B}$ is the number of equal

instances between Files A and B and N_B is the number of instances in File B). As Table 7 shows, the similarity rate between two files given by IFCdiff is 80%-99%. However, the similarity rate should be 100% because no semantic changes exist between these two files, and that is the result of the quick hash-code-accelerated algorithm. In our opinion, this result may have occurred because IFCdiff includes some meaningless properties when comparing two instances: it ignores only some typical meaningless properties, such as the GUID, Owner History Info, and Property set order.

Regarding time consumption, IFCdiff increases quickly and dramatically. The time consumption for M1 detection is less than 0.1 s, but for M3 it is 19.4 s. Shi, X. et al. [20] have calculated that the time complexity of IFCdiff is $O(nlog(n))$, but the number of instances in the IFC file may increase considerably when the file is exported from Revit (as shown in the Instance Number column in Table 7). Therefore, the time complexity of IFCdiff may be greater than expected because the number of instances in the IFC files may increase much faster than in the Revit files for the same models.

In addition, the content-based method used by IFCdiff is based on the IFC hierarchical structure, where most instances will cite or be cited by other instances. However, the hierarchical structure is also a constraint for this method. In contrast, the hash-code-accelerated method proposed in this research has no dependence on the file structure: it can detect changes in any file that composed of instances consist of properties.

Table 7. Detection performances of IFCdiff and the quick hash-code-accelerated algorithm

Model	Instance Number		Similarity Rate		Time Consumption (s)	
	IFC	Revit	IFCdiff	*Qhc*	IFCdiff	*Qhc*
M1_All-DA(M)	578	2007	80.1%	100%	<0.1	0.57
M2_All-DA(M)	113703	4869	99.4%	100%	2.1	2.2
M3_All-DA(M)	976881	12896	99.2%	100%	19.4	8.1

Note. *Qhc* is the quick hash-code-accelerated algorithm.

6.3.2 Proposed Method and Invited Designers

In this test, 30 junior designers were invited to make design changes to the BIM model M4 individually. They were asked to make changes until they thought that others would need 30-45 minutes to find all the changes without being given any clue about the changes. Then, each designer was asked to randomly choose another changed model and detect the design changes manually without any clue about the changes. Finally, time consumption and detection rate were calculated for each designer. The complete data of this test are listed in Appendix A. The detection performances of the designers are shown in Figure 14, drawn from left to right in ascending detection rate order.

In Figure 14, most of the designers (73%) have detection rates of 78% ±12%, and only a few designers (13%) achieve a 100% detection rate. In this test, the average detection rate is 79.8%, and the average time for each manual detection is

5.18 min; thus, each designer requires an average of 5.18 minutes to determine one design change.

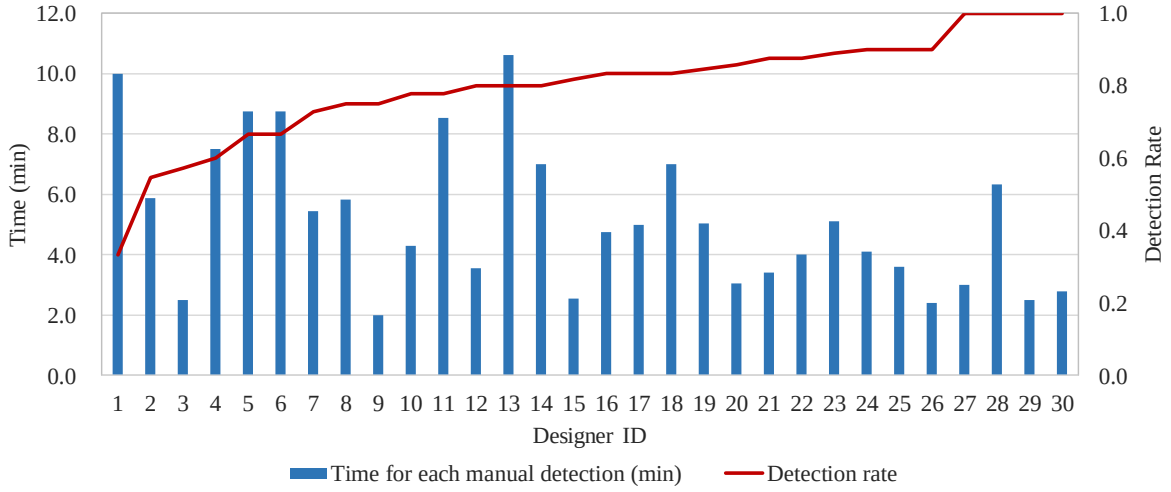


Figure 14. Detection performances of the designers in M4.

To compare the detection performance differences between the proposed method and the designers, we randomly select 10 of the changed M4 models and use the developed application to detect their changes. In this test, the detection results of the comparison-first, hash-code-accelerated, and quick hash-code-accelerated algorithms are all the same for all the 10 models (which implies that hash collisions did not occur). The ratios of time consumption of these three algorithms are consistent with Table 6 (note that the ID is reliable in this test). The detection result of the quick hash-code-accelerated algorithm for detecting one of the changed models is shown in Figure 15.

In Figure 15, changes in the categories SketchLines, WeakDims, Views, and Cameras are intentionally not shown by unchecking the corresponding boxes in the left tree view in the application to filter out meaningless changes that designers are not concerned about. The detection result of the quick hash-code-accelerated algorithm is that 37 elements are changed (14 modified, 20 deleted, and 3 added); this result includes all the changes in this model. Therefore, this algorithm achieves a 100% detection rate in this case. Additionally, the user can right-click one of the changes to select this item in the Revit UI, such as the modified window shown in Figure 15. Therefore, the efficiency of design change detection is greatly enhanced via this automatic and clear approach, and the time needed for detection involves only time needed to run the algorithm in the application. In contrast, the detection result of the matching-first algorithm is 167 changed elements (14 modified, 85 deleted, 68 added), which includes all the changes but also includes many meaningless changes. For example, some materials, analytical spaces, system zones, etc. were deleted and recreated between the two files, but they are identical except for the ID change. Nevertheless, these meaningless changes were still detected by the matching-first algorithm.

The detection performances of the designers and the quick hash-code-accelerated algorithm for the 10 changed M4 models are shown in Figure 16, and the average results are summarized in Table 8. It can be seen, the detection rates of the

quick hash-code-accelerated algorithm are always 100%, while that of the designers' manual detecting are varied from 57% – 100% and the average value is 81.3%. The time consumption of the algorithm is only 2 – 3 s and it is much lower than the manual detecting, which varied from 150 – 525 s with an average value of 291 s. Note that the manual detection time is the time for each detection but the algorithm detection time is the time for total detection in the changed model, which has an average of approximately 7 changes and 5800 instances. Thus, the design reviewing time can be saved by $1 - 2.4/(291 \times 7) = 99.88\%$.

Finally, we can conclude that using the proposed algorithm, the detection rate can be improved from 81.3% to 100% which helps to find out approximately 1/5 design changes missed by manual detections, and the design reviewing time can be saved by 99.88% for detecting 7 changes from 5800 instances.

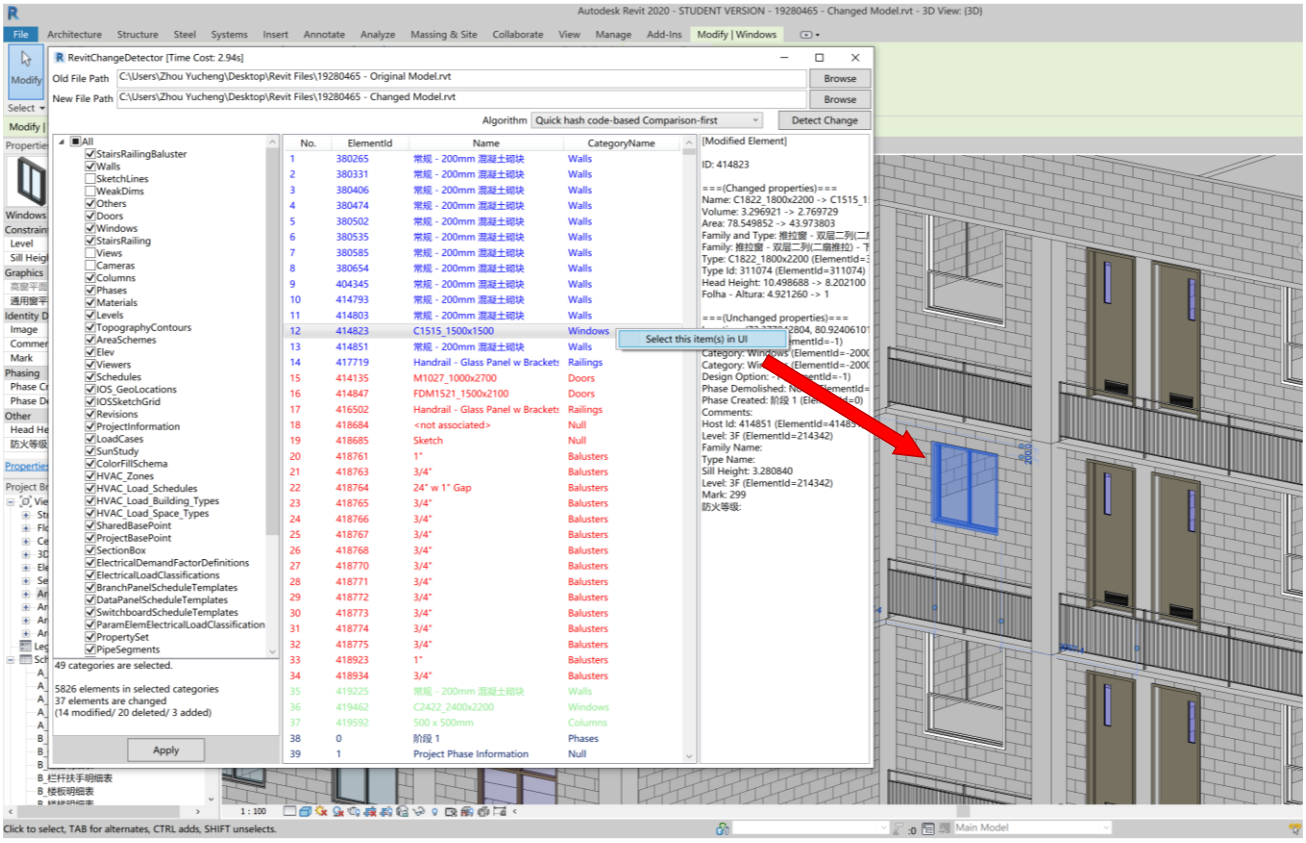


Figure 15. Screenshot of detecting changes by the quick hash-code-accelerated algorithm in M4.

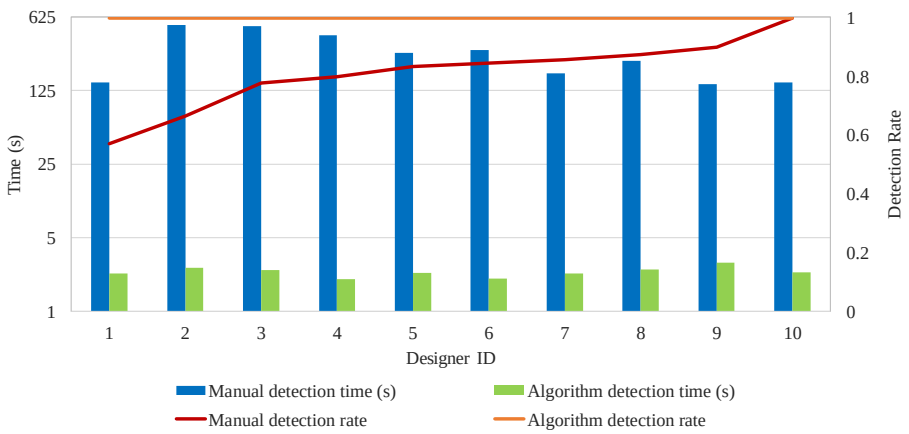


Figure 16. Detection performances of the designers and the quick hash-code-accelerated algorithm.

Table 8. Average detection performances of the designers and the quick hash-code-accelerated algorithm.

	Average Detection Rate	Average Detection Time (s)
Designers	81.3%	291
Algorithm	100%	2.4

7 Discussion

In this research, the detection methods are implemented and tested in Revit because it provides rich documents and APIs as well as an easy environment for implementation. However, the proposed method and the results are platform independent. First, our semantic classification is universal, which could apply to different BIM data models, including IFC, Revit, and even other models, and it provides the criterion to judge the correctness of detection methods and works as a guide to develop a detection method. Then, our detection method is platform independent because it: 1) adopts our semantic classification, which is platform independent, and 2) bases on the general structure of the BIM model which is composed of instances that consist of properties. Therefore, our method can be extended to handle other data models like IFC as long as implementing the methods for categorized instance properties extraction and the hash-code-based encoding of properties. Finally, the verification results are also platform independent because our method is compared with both existing methods implemented in Revit and another method based on IFC, and the results are also consistent with the theoretical analysis.

The novelty of our method is utilizing the hash code of properties to greatly improve the efficiency along with eliminating meaningless design changes at the model level by combining the hash code of properties. In this research, a basic hash code calculation approach is considered for properties and instances. To further improve the efficiency, more robust hash functions for calculating the hash codes of instances with zero hash collisions could be investigated in the future. This approach would save the currently optional double-check process in the quick hash-code-accelerated method,

which should further reduce the time consumption in some cases. However, instead of hash code, other signature generation approaches can also be applied in our method, and can be further studied to adjust to mover complex situations.

The implementation of a detection method may be quite complicated to cover more application cases. Detection methods are implemented only in Revit in this research, however, the implementation in IFC will be more complicated because IFC schema provides different ways (e.g., sweeping, CSG, and B-Rep) for representing the geometry of an instance. Consequently, geometries with different syntactic data may have the same semantic meaning. A possible approach to compare geometry semantically is to first discretize the solid models into 3D meshes and then use existing 3D shape comparison methods to compare the discretized shapes [20,38]. Additionally, when comparing two BIM models of different versions (e.g., IFC schema version or software version), the syntactic definition of an instance may be changed, which increases the complexity of considering its semantics. A feasible approach is converting the model with the lower version to the higher version, however, data loss should be properly handled if it occurs in this conversion.

In the workflow of the AEC industry, our method and can be applied to many scenarios to help the real world design process. For instance, a designer can use our method to compare the similarities and differences between two different design versions quickly after he/she makes some modifications based on a certain design. Meanwhile, when an architect makes some modifications in a BIM model and shares it with a structural engineer, the latter can use our method to detect the design changes to evaluate the impact on the structural design as well as analyzing the adjustment of the design. In addition, other professionals can also analyze changed cost, quantity take-off, and other specific constructions and possible construction schedule adjustments by detecting design changes after obtaining the changed BIM model. This research also helps investigate the generation mechanism of design changes and analyze subsequent design changes. For example, design changes in multiple BIM models can be detected by our method to provide support for further summarizing and analyzing potential design changes.

8 Conclusion and Future Work

Currently, the definition of semantic design changes is not clear, and is always mixed with data changes, thus leading to meaningless results and wasting a lot of time. To address these issues, we first propose a semantic classification of design changes that divides data changes into three categories and classifies design changes into three levels from a designer's viewpoint. Next, we improve the detection method by considering the proposed classification criterion to enable the detection of semantic design changes. Finally, the hash-code-based encoding of instance properties is utilized to accelerate the comparison process of design change detection. Results show that our method could: 1) detect semantic changes with 100% correctness while reducing up to 98.1% of the detection time of existing methods, and also be much faster than IFCdiff when the model instance number is large, and 2) improve the average detection rate of designers from 81.3% to

100% while saving 99.88% of the design reviewing time (detecting 7 changes from 5800 instances). Moreover, our semantic classification and detection method of design changes are platform independent, and can be the basis of future studies of detection methods. In summary, the significant contributions of this research are as follows.

- We propose a semantic classification of design changes, which can be the criterion to judge the correctness of detection methods and a guide to develop a method for detecting semantic design changes.
- We propose the (quick) hash-code-accelerated method, which can detect semantic changes with 100% correctness and high and stable efficiency.
- We establish three groups of BIM models that each contain the same number of models and change types but have different complexity levels. These models can be used as baseline datasets for validating future detection methods.

Future work remains to be done to expand and enhance the contribution of this research. First, as mentioned in Section 7, the implementation of detection methods in other platforms such as IFC can be further studied. Second, more application of detection methods in the real world design process such as assisting the analysis of subsequent construction changes can be further investigated. Third, more robust and efficient hash functions for calculating the hash codes can be further studied. Finally, to further illustrate the performance enhancement of the detection methods compared to human designers, factors that may affect the designers' production performances can be explored and studied [39-41]. These aspects will constitute our future work.

9 Acknowledgments

This research was funded by the Beijing Natural Science Foundation (No. 8194067), the Natural Science Foundation of China (No. 51908323) , the National Key R&D Program of China (No. 2018YFD1100900) and the Tsinghua University Initiative Scientific Research Program (No. 2019Z02UOT).

Appendix A. Complete Data of Detection in the Test of Designers

Table 9 gives the complete detection data in the test of the designers as described in Section 6.3.2.

Table 9. Complete detection data in the test of the designers.

Designer	Number of changes	Number of detections	Total time cost (min)	Time cost for each detection (min)	Detection rate
#1	9	7	30	4.29	0.778
#2	9	9	27	3.00	1.000
#3	10	8	28.5	3.56	0.800
#4	11	8	43.6	5.45	0.727
#5	9	7	59.8	8.54	0.778
#6	10	9	37	4.11	0.900
#7	12	9	52.5	5.83	0.750
#8	10	10	63.2	6.32	1.000

#9	5	4	42.5	10.63	0.800
#10	6	5	23.8	4.76	0.833
#11	7	6	18.3	3.05	0.857
#12	13	11	55.3	5.03	0.846
#13	8	8	20	2.50	1.000
#14	9	8	40.8	5.10	0.889
#15	6	5	25	5.00	0.833
#16	6	4	35	8.75	0.667
#17	10	6	45	7.50	0.600
#18	8	7	23.8	3.40	0.875
#19	6	4	35	8.75	0.667
#20	9	9	25	2.78	1.000
#21	6	5	35	7.00	0.833
#22	7	4	10	2.50	0.571
#23	8	7	28.1	4.01	0.875
#24	10	9	32.5	3.61	0.900
#25	11	9	23	2.56	0.818
#26	8	6	12	2.00	0.750
#27	11	6	35.3	5.88	0.545
#28	5	4	28	7.00	0.800
#29	9	3	30	10.00	0.333
#30	10	9	21.7	2.41	0.900
Mean	8.6	6.9	32.9	5.18	0.798
SD	2.1	2.1	12.8	2.38	0.146

References

[1] M. Nour, K. Beucke, Object versioning as a basis for design change management within a BIM context, International Conference on Computing in Civil and Building Engineering (ICCCBE-XIII), Nottingham University Press, Nottingham, UK, 2010, pp. 147-152. https://www.semanticscholar.org/paper/Object-versioning-as-a-basis-for-design-change-a-Nour-Beucke/999b8f25c0266a6e24812dc28bbcbd14f5af64f5 (Jan 10, 2020).

[2] I.A. Motawa, C.J. Anumba, S. Lee, F. Peña-Mora, An integrated system for change management in construction, Automation in Construction 16 (3) (2007) 368-377. DOI: 10.1016/j.autcon.2006.07.005.

[3] V. Likhitruangsilp, T.N. Handayani, N. Yabuki, A BIM-Enabled Change Detection System for Assessing Impacts of Construction Change Orders, 17th International Conference on Computing in Civil and Building Engineering, Tampere, Finland, 2018. DOI: 10.1061/9780784481264.061.

[4] A. Borrmann, M. König, C. Koch, J. Beetz, Building Information Modeling: Why? What? How? in: A. Borrmann, M. König, C. Koch, J. Beetz (Eds.), Building Information Modeling: Technology Foundations and Industry Practice, Springer International Publishing, Cham, 2018, pp. 1-24. DOI: 10.1007/978-3-319-92862-3_1.

[5] J. Zhang, Q. Liu, Z. Hu, J. Lin, F. Yu, A multi-server information-sharing environment for cross-party collaboration on a private cloud, Automation in Construction 81 (2017) 180-195. DOI: 10.1016/j.autcon.2017.06.021.

[6] C. Eastman, P. Teicholz, R. Sacks, K. Liston, BIM handbook: A guide to building information modeling for owners, managers, designers, engineers and contractors, John Wiley & Sons, 2008. ISBN: 111802169X.

[7] G. Gao, Y. Liu, P. Lin, M. Wang, M. Gu, J. Yong, BIMTag: Concept-based automatic semantic annotation of online BIM product resources, Advanced Engineering Informatics 31 (2017) 48-61. DOI: 10.1016/j.aei.2015.10.003.

[8] A. Deshpande, S. Azhar, S. Amireddy, A Framework for a BIM-based Knowledge Management System, Procedia Engineering 85 (2014) 113-122. DOI: 10.1016/j.proeng.2014.10.535.

[9] A. Kiviniemi, M. Fischer, Requirements Management Interface to Building Product Models, (2004). DOI: 10.25643/bauhaus-universitaet.242.

[10] H.P. Tserng, Y. Lin, Developing an activity-based knowledge management system for contractors, Automation in Construction 13 (6) (2004) 781-802. DOI: 10.1016/j.autcon.2004.05.003.

[11] D.E. O'Leary, Enterprise knowledge management, Computer 31 (3) (1998) 54-61. DOI: 10.1109/2.660190.

[12] M.H. Rasmussen, M. Lefrançois, P. Pauwels, C.A. Hviid, J. Karlshøj, Managing interrelated project information in AEC Knowledge Graphs, Automation in Construction 108 (2019) 102956. DOI: 10.1016/j.autcon.2019.102956.

[13] H. Lai, X. Deng, Interoperability Analysis of IFC-based Data Exchange Between Heterogeneous BIM Software, Journal of Civil Engineering and Management 24 (7) (2018) 537-555. DOI: 10.3846/jcem.2018.6132.

[14] I.J. Ramaji, A.M. Memari, Interpreted information exchange: Systematic approach for BIM to engineering analysis information transformations, Journal of Computing in Civil Engineering 30 (6) (2016) 4016028. DOI: 10.1061/(ASCE)CP.1943-5487.0000591.

[15] M. Nour, Manipulating IFC sub-models in collaborative teamwork environments, Proc. of the 24th CIB W-78 Conference on Information Technology in Construction, Maribor, Slovenia, 2007, pp. 26-29. http://itc.scix.net/paper/w78_2007_53 (Jan 10, 2020).
[16] K. Krishnamurthy, K.H. Law, A Data Management Model for Collaborative Design in a CAD Environment, Engineering with Computers 13 (2) (1997) 65-86. DOI: 10.1007/BF01350751.
[17] Y.S. Jeong, C.M. Eastman, R. Sacks, I. Kaner, Benchmark tests for BIM data exchanges of precast concrete, Automation in Construction 18 (4) (2009) 469-484. DOI: 10.1016/j.autcon.2008.11.001.
[18] H. Ma, K.M.E. Ha, C.K.J. Chung, R. Amor, Testing semantic interoperability, Proceedings of the Joint International Conference on Computing and Decision Making in Civil and Building Engineering, Montreal, Canada, 2006, pp. 1216-1225. http://itc.scix.net/paper/w78-2006-tf193 (Jan 10, 2020).
[19] S. Daum, A. Borrmann, Enhanced Differencing and Merging of IFC Data by Processing Spatial, Semantic and Relational Model Aspects, 23rd International Workshop of the European Group for Intelligent Computing in Engineering, Kraków, Poland, 2016. https://mediatum.ub.tum.de/1324278 (Jan 10, 2020).
[20] X. Shi, Y.S. Liu, G. Gao, M. Gu, H. Li, IFCdiff: A content-based automatic comparison approach for IFC files, Automation in Construction 86 (2018) 53-68. DOI: 10.1016/j.autcon.2017.10.013.
[21] P.E. Love, H. Li, Quantifying the causes and costs of rework in construction, Construction Management & Economics 18 (4) (2000) 479-490. DOI: 10.1080/01446190050024897.
[22] J.L. Burati Jr, J.J. Farrington, W.B. Ledbetter, Causes of quality deviations in design and construction, Journal of Construction Engineering and Management 118 (1) (1992) 34-49. DOI: 10.1061/(ASCE)0733-9364(1992)118:1(34).
[23] V. Moayeri, O. Moselhi, Z. Zhu, BIM-based model for quantifying the design change time ripple effect, Canadian Journal of Civil Engineering 44 (8) (2017) 626-642. DOI: 10.1139/cjce-2016-0413.
[24] Wikipedia, Industry Foundation Classes. https://en.wikipedia.org/wiki/Industry_Foundation_Classes (Jan 10, 2020).
[25] Autodesk, Element Class. http://www.revitapidocs.com/2018.1/eb16114f-69ea-f4de-0d0d-f7388b105a16.htm (Jan 10, 2020).
[26] G. Lee, J. Won, S. Ham, Y. Shin, Metrics for quantifying the similarities and differences between IFC files, Journal of Computing in Civil Engineering 25 (2) (2011) 172-181. DOI: 10.1061/(ASCE)CP.1943-5487.0000077.
[27] M.T. Shafiqa, S.R. Lockleyb, Signature-based matching of IFC Models, ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction, Vol. 35, IAARC Publications, Berlin, Germany, 2018, pp. 1-9. DOI: 10.22260/ISARC2018/0138.
[28] A. Jaly-Zada, C. Koch, W. Tizani, IFC Extension for Design Change Management, 32nd CIB W78 Conference 2015, Eindhoven, The Netherlands, 2015. https://www.semanticscholar.org/paper/IFC-Extension-for-Design-Change-Management-Jaly-Zada-Koch/e0a154951eccb19f3ac07d33392da83f4498149e (Jan 10, 2020).
[29] G. Lee, C.M. Eastman, R. Sacks, Eliciting information for product modeling using process modeling, Data & Knowledge Engineering 62 (2) (2007) 292-307. DOI: 10.1016/j.datak.2006.08.005.
[30] J. Kunkel, Revit Element ID - How To Get It and What To Do With It - CADD Microsystems. https://www.caddmicrosystems.com/blog/2018/08/revit-element-id-how-to-get-it-and-what-to-do-with-it/ (2019/1/13).
[31] J. Wu, J. Zhang, Introducing Geometric Signatures of Architecture, Engineering, and Construction Objects and a New BIM Dataset, Computing in Civil Engineering 2019, 2019, pp. 264-271. DOI: 10.1061/9780784482421.034.
[32] J.R. Lin, Y.C. Zhou, J.P. Zhang, Z.Z. Hu, Classification and Exemplary BIM Models Development of Design Changes, 36th International Symposium on Automation and Robotics in Construction, Banff, Canada, 2019, pp. 122-127. DOI: 10.22260/ISARC2019/0017.
[33] Y.W. Zhou, Z.Z. Hu, J.R. Lin, J.P. Zhang, A Review on 3D Spatial Data Analytics for Building Information Models, Archives of Computational Methods in Engineering (2019). DOI: 10.1007/s11831-019-09356-6.
[34] T. He, J. Zhang, J. Lin, Y. Li, Multiaspect Similarity Evaluation of BIM-Based Standard Dwelling Units for Residential Design, Journal of Computing in Civil Engineering 32 (5) (2018) 4018032. DOI: 10.1061/(ASCE)CP.1943-5487.0000774.
[35] T. Liebich, IFC 2x Edition 2 model implementation guide, International Alliance for Interoperability (2004). https://standards.buildingsmart.org/documents/Implementation/IFC2x_Model_Implementation_Guide_V2-0b.pdf (Jan 10, 2020).
[36] P. Schueffel, N. Groeneweg, R. Baldegger, The crypto encyclopedia: coins, tokens and digital assets from A to Z, Growth publisher, 2019. ISBN: 9782940384471.
[37] V. H., D.B. K., XOR-based hash functions, IEEE Transactions on Computers 54 (7) (2005) 800-812. DOI: 10.1109/TC.2005.122.
[38] A. Vonthron, C. Koch, M. König, Removing duplicated geometries in IFC models using rigid body transformation estimation and flyweight design pattern, Visualization in Engineering 6 (1) (2018). DOI: 10.1186/s40327-018-0061-x.
[39] L. Zhang, B. Ashuri, BIM log mining: Discovering social networks, Automation in Construction 91 (2018) 31-43. DOI: 10.1016/j.autcon.2018.03.009.
[40] S. Yarmohammadi, R. Pourabolghasem, D. Castro-Lacouture, Mining implicit 3D modeling patterns from unstructured temporal BIM log text data, Automation in Construction 81 (2017) 17-24. DOI: 10.1016/j.autcon.2017.04.012.

[41] L. Zhang, M. Wen, B. Ashuri, BIM log mining: measuring design productivity, Journal of Computing in Civil Engineering 32 (1) (2017) 4017071. DOI: 10.1061/(ASCE)CP.1943-5487.0000721.