A DATA STRUCTURE TO INCORPORATE VERSIONING IN 3D CITY MODELS

S. Vitalis[1,*], A. Labetski[1], K. Arroyo Ohori[1], H. Ledoux[1], J. Stoter[1]

[1] 3D Geoinformation Group, Delft University of Technology, the Netherlands -
(s.vitalis, a.labetski, g.a.k.arroyoohori, h.ledoux, j.e.stoter)@tudelft.nl

KEY WORDS: Versioning, 3D City Models, CityJSON, CityGML, Git

ABSTRACT:

A 3D city model should be constantly updated with new versions, either to reflect the changes in its real-world counterpart, or to improve and correct parts of the model. However, the current standards for 3D city models do not support versioning, and existing version control systems do not work well with 3D city models. In this paper, we propose an approach to support versioning of 3D city models based on CityJSON and the concepts behind the Git version control system, including distributed and non-linear workflows. We demonstrate the benefits of our approach in two examples and in our software prototype, which is able to extract a given version of a 3D city model and to display its history.

1. INTRODUCTION

3D city models are increasingly being used to represent the complexity of today's urban areas, as they aid in understanding how different aspects of a city can function, such as emergency response, noise propagation and solar power potential (Biljecki et al., 2015). When a 3D city model is created, it represents its real-world counterpart at a snapshot in time. However, as time passes, its model needs to be updated and to evolve, much like its real-world counterpart.

In order to keep a 3D city model up to date, new versions of it should be regularly created due to three main causes. First, cities themselves are constantly changing (e.g. a new building is built, a road is closed, etc.). Second, the modelling aspect of a project may change, such as when new information becomes available for existing city objects through the acquisition of a new dataset or the output of a new simulation. Last, certain maintenance processes may cause changes to a dataset (e.g. geometric errors are fixed, or the classification used in an attribute is changed).

A commonly used standard for storing 3D city models is CityGML (Open Geospatial Consortium, 2012). While a number of CityGML models for different cities exist, these represent only static snapshots in time and most often do not evolve alongside their real-world counterparts. Of the openly available 3D city model datasets, the vast majority have not been updated since their creation[1]. For the cities that have had multiple versions, we have observed that municipalities and governments tend to completely recreate the 3D city models, rather than update them (e.g. the city of Helsinki (Airaksinen et al., 2019)). There are, in our opinion, two factors that make CityGML datasets difficult to update. First, CityGML files are notoriously complex and difficult to edit, and thus the number of software packages supporting their editing is very low; Ledoux et al. (2019) explain this issue in detail. Second, the current version of CityGML (2.0) does not have any mechanism for the versioning of files. While there is a proposal to remedy this situation (Chaturvedi et al., 2017), as we explain in Section 2.2.3, we believe this proposal has some shortcomings, especially because it jumbles versioning with the semantic modelling of changes in time.

In a more general context, generic solutions to the problem of versioning data are widely available in the form of version control systems (VCS). These systems provide a robust mechanism for storing different versions of files and for tracking changes (and their associated metadata) in a structured and meaningful way. VCS are often the key aspect of a project for two reasons. First, they provide enough information so that contributors can track and review changes. For instance, when a bug is identified in a software project, its source can be tracked. Second, they enable easier collaboration between multiple authors. This is due to the fact that authors can keep track of who changed what and when through metadata.

In this paper, we propose a new approach to deal with the versioning of 3D city models. Our approach is based on the most used/popular VCS, which is called Git (Spinellis, 2012). Git was chosen as the basis for our approach because it offers a distributed architecture and offers several advantages in practice, e.g. authors can work locally and working with different versions is handled elegantly; we further expand on Git in Section 2.1.1. We have incorporated the successful characteristics of Git's data structure in our solution, which we described in Section 3. Given the shortcomings of a GML implementation, as mentioned above (difficulties in editing and manipulating files), we have implemented our approach for CityJSON, which is a JSON-based implementation of the CityGML data model (v2.0) (Ledoux et al., 2019). We present in Section 4 two concrete examples of how our versioning approach could be applied, and we demonstrate the benefits it would have by showing the prototype software we have developed in Section 5.

2. RELATED WORK

2.1 Version control systems

Version control systems (VCS) are utilised for the management of changes in information (Spinellis, 2005) this can be documents, websites, computer code and even geospatial data. Changes (also known as revisions) can be identified with a unique id and contain information such as a time stamp and the

*Corresponding author
[1] https://3d.bk.tudelft.nl/opendata/opencities/

person responsible for the revision (Ball et al., 1997). Versioning is already well established in the realm of computer programming where it is often used to track changes in programming code (Spinellis, 2005; Ball et al., 1997). In large projects, multiple programmers can work on different branches to develop their portion of a particular project. Versioning is also useful for dealing with errors and reverting a change in cases where an update may have a bug.

VCS's main component is a *repository*, which contains all informations about all the save iterations of files as well as the metadata information of every version (Loeliger, McCullough). They utilise *branches* within a repository which make it easier for users to work on different aspect of a project at the same time (Tichy, 1985). That means, that users can have multiple variations of the evolution of files and they can switch from one to another (Pilato et al., 2008). For example, this can be used to develop new experimental features before incorporating them into the main project. Conceptually, the repository can be perceived as a graph where nodes represent versions and branches are leafs of the graph. Normally, one branch is considered the main as it contains the stable version of the project. When a user is satisfied with the state of a branch, it can be merged to the trunk, or discarded if it is unsatisfactory. These merges can also be tagged to identify milestones in a project.

Given the above description, VCS can be divided into two categories regarding their architecture: *centralised* and *distributed*. Centralised VCS, such as Concurrent Versions System (CVS) (Grune et al., 1986), Revision Control System (RCS) (Tichy, 1985), and Apache Subversion (SVN) (Pilato et al., 2008), use a single repository where every author has to commit changes. This means that in order to interact with the VCS, one has to be online in order to access this single repository. In addition, this architecture discourages many small contributions to the project, as normally a commit conflicts with other contributions that might have been done meanwhile by another author. Distributed VCS, such as Git (Loeliger, McCullough), utilise an non-centralised architecture where every contributor has its own repository. This means that users commit in their own local repository and they can share multiple versions with other remote repositories. Such an approach allows users to work locally, without being destructed by other authors' work. Then changes can be shared at a later stage, through the process of *pushing* or *pulling* versions between repositories.

2.1.1 Git. Git is one of the most popular VCS (Spinellis, 2012). Its success can be attributed to its distributed architecture and its internal structure. The distributed architecture allows teams to be flexible on how to incorporate Git in their workflows. This is due to the fact that authors can work locally and share changes only when they are complete. In addition, the minimal and versatile internal structure of Git makes it fast and robust when executing VCS operations, such as merging or checking out.

Git is composed of two components: the data structure, which is used for the storage of all versions of a file and the metadata related to these versions; and the software, which provides the functionality related to tracking changes of a file and facilitating the interaction between a user and the data structure.

In order to undertake the complicated operations required by such an architecture, Git uses a flexible internal data structure. Versions are organised in a directed acyclic graph (DAG) (Figure 1). In this graph, a version is a node which is called a *commit*. Every commit derives from the previous one, therefore it

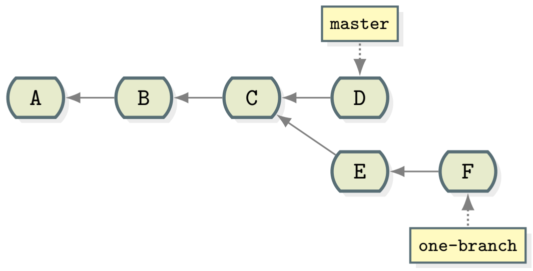


Figure 1. Example of a directed acyclic graph as defined by the versioning mechanism. Every node knows it's parent, therefore the graph can be traversed from the leafs to the root. Branches, which are only pointers to a version, act as starting points of a traversal, therefore defining the actual line of nodes.

is responsible for knowing its parents. It is, also, identified by the hash[2] of its content: the metadata information (author, date, message), the id of its parents and the content of the files in this commit. Thus, it is possible to populate the history of the project by starting from the "last" version and traversing back following its parents until the initial commit is reached.

In order to avoid ambiguity, Git does not denote one commit as the last one. Instead, special pointer objects, called *refs*, can be used as nodes to initiate the traversal of Git's DAG, therefore the project's history is populated on-the-fly.

Branches are the main refs in Git, as they only store a key-value pair where key is a name and value is a commit id (i.e. its hash). A branch can, therefore, be denoted as the last of a chain of commits.

2.2 3D city models

There are many documented usages of 3D city models across a diverse array of applications ranging from disaster management, urban planning, navigation and noise emission, to name a few (Baig, Rahman; Biljecki et al., 2016). This is further reflected in the number of national mapping agencies producing 3D city models (Stoter et al., 2015). The OGC (Open Geospatial Consortium) open standard for the storage and exchange of 3D city models is CityGML (Open Geospatial Consortium, 2012), and includes geometry, semantics, and graphical appearance. It is both the name of the model and the XML encoding. CityGML is modular with modules representing elements of a city including buildings, transportation, water bodies, vegetation, etc.

2.2.1 CityJSON. CityJSON is a JSON-based exchange format for the CityGML data model (version 2.0.0). Its current version (v1.0) implements most of the CityGML data model, and all of the CityGML modules have been mapped. This means that in practice, one can convert automatically CityJSON to CityGML, and vice-versa.

CityJSON was designed with programmers in mind, and it was also designed to be compact and friendly for web and mobile development. This means a simpler structure is used to represent a city: the data model of CityGML has been "flattened out". A CityJSON file simply contains all the city objects (such as buildings, roads, or trees) indexed under one point of entry, and hierarchical nature of CityGML is removed.

A simple file contains a few JSON properties:

[2]Hashes, a concept borrowed from mathematics, are unique identifiers composed of a series of letters and numbers.

```
{
  "type": "CityJSON",
  "version": "1.0",
  "CityObjects": {},
  "vertices": [],
  "appearance": {}
}
```

and the property "CityObjects" would contain the city objects, for instance:

```
"CityObjects": {
  "id-1": {
    "type": "Building",
    "attributes": {...},
    "children": ["id-2", "id-3"],
    "geometry": [{...}]
  },
  "id-2": {
    "type": "Road",
    "geometry": [{...}]
    ...
  }
}
```

The core structure of CityJSON can be extended by adding new properties (to the city model and to each city objects), and in Section 3 we exploit this in our approach to add the versioning capabilities.

2.2.2 Versioning of 3D city models. There have been various attempts at better integrating versioning with GIS data. For example, GeoGig[3] aims to enable versioning for multiple data formats, including Shapefile, GeoJSON, PostgreSQL and Oracle Database, following Git's distributed architecture and workflow. Another approach to the problem is through providing versioning in GIS applications, such as the *QGIS versioning plugin*[4] and *FastVersion*[5] for QGIS[6]. While these tools could potentially work with 3D city model datasets, mapping CityGML and CityJSON can be particularly challenging. This is due to the fact that the CityGML data model is not easily mapped to relational models, such as the ones used in the data formats that GeoGig or QGIS uses.

One potential solution to the problem of 3D city models' versioning could be to simply store CityGML or CityJSON files in a Git repository. While Git is a powerful and robust VCS, there are several elements that make VCSs inefficient for versioning 3D city model files. First, the order of elements is insignificant for 3D city models, i.e. a model can change the order or city objects or their bounding surfaces and this would be tracked as a change from Git. Given that with traditional Git versioning systems, it is easy for changes to be detected that are actually simply due to the reordering of data or elements, or the introduction of new lines, this approach makes comparisons between different versions more difficult. Additionally, in order to interact with this tool you would need to interact with Git which increases the complexity of usage. Using this application would mean that versioning information is not stored with the file and the versioning repository would need to be shared separately.

2.2.3 Proposed CityGML versioning module. While the current version of CityGML (2.0) does not have support for storing versions of objects, there is a proposed versioning module for CityGML 3.0, which is based on Chaturvedi et al. (2017). As Figure 2 shows, the proposal aims at storing the 'creation' and 'termination' date and time for a feature as well as the 'valid from' and 'valid to' date and time. Then, versions are aggregations of these timestamped features, and they can also have a `VersionTransition` property which records the reason for the transition and the type (this is borrowed from the INSPIRE standard). These transitions are composed of a number of transactions, which represent individual changes linking old and new features.

The proposed CityGML versioning module does achieve some support for versioning, but it is a problematic solution because it conflates versioning with life-cycle modelling. This can be most easily seen in the `TransactionValue` enumeration, which contains both versioning operations (fork and merge) and life-cycle stages (planned and realized). Modern VCS (e.g. Git) support powerful automatic operations based on how they can track changes in data in a time-independently way (i.e. the nodes in the DAG represent changes in the data and can be rearranged by operations). By contrast, life-cycle modelling is a lot more restrictive since the operations have a semantic meaning and their order matters. These two aspects are thus incompatible and should not be implemented in the same mechanism.

In addition, the proposed CityGML versioning module appears to store the results obtained from using VCS explicitly, as opposed to just storing the much simpler internal structure used in VCS, from which the rest can be obtain using simple operations in software. This introduces both unnecessary complexity (e.g. modelling of versions, transitions and transactions) and unnecessary redundancy (e.g. linking features from both versions and transactions). The resulting model thus makes software implementation needlessly difficult and discourages practitioners from using it.

3. METHODOLOGY

This section describes our approach to store multiple versions of a 3D city model in a CityJSON file. It has to be clarified that although in Section 3.1 we do investigate how this can potentially interact with software, our solution only focuses on the description of the architecture and the data structure. In Section 5 we show the prototype software implementation which we developed in order to validate the proposed data structure, but this is not intended to be part of the proposed solution.

3.1 System design and architecture

We propose a method where all versions of all CityJSON objects are stored in a single file. In other words, a CityJSON file acts as a repository, which we here refer to as "versioned CityJSON" (vCityJSON). Users can interact with a vCityJSON in two ways. First, they can directly add new city object versions manually (e.g. by duplicating an existing city object, renaming it and making the necessary changes). In this approach, they would need to add a new version and define it accordingly (as defined in Section 3.2). Second, it is possible to develop a tool that extracts a "simple" city model based on the description of one version in a vCityJSON file. Then, changes can be made to this file, after which the tool can incorporate the changes in the versioned CityJSON by adding all of the necessary new objects (Figure 3).

[3]http://geogig.org/
[4]https://github.com/Oslandia/qgis-versioning
[5]http://www.fastversion.org/
[6]https://qgis.org

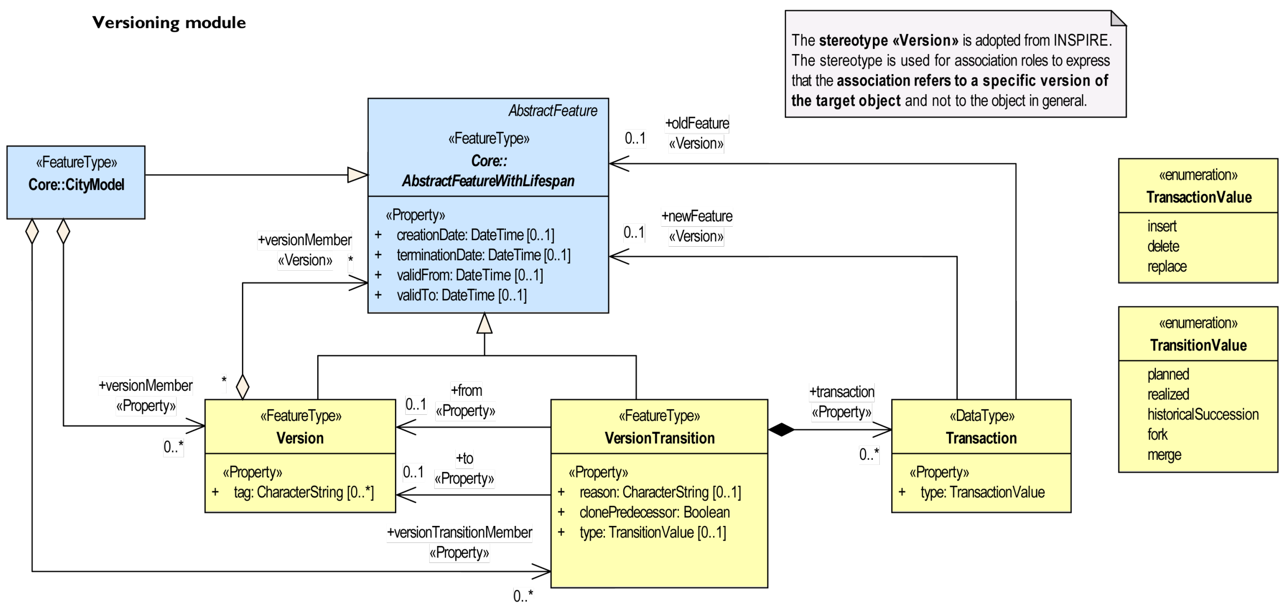


Figure 2. Proposed versioning for CityGML version 3, from OGC (2019).

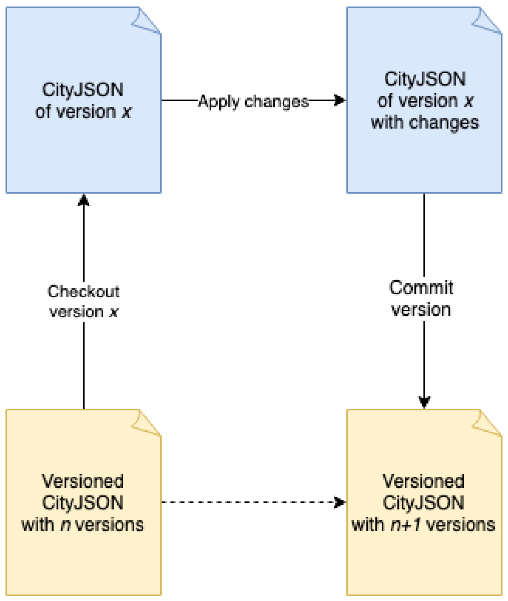

CityJSON of version x
Apply changes
CityJSON of version x with changes
Checkout version x
Commit version
Versioned CityJSON with n versions
Versioned CityJSON with n+1 versions

Figure 3. A suggested workflow indicating how a new version can be added by using an intermediate software. The user can extract (checkout) a version from the versioned CityJSON file, apply changes to the object and then commit a new version based on the changes. We need to clarify that in this article we do not propose how to implement such an operation.

3.2 Data structure

As described in 3.1, a vCityJSON acts like a repository for the city model. That means that the file contains all versions of city objects, as well as the versions' information with their metadata. The structure vCityJSON is similar to a regular CityJSON, with the addition of a "versioning" property at the root of the CityJSON object:

```
{
  "type": "CityJSON",
  "version": "1.0",
  "CityObjects": {},
  "versioning": {},
  "vertices": []
}
```

All versions of city objects are listed under the "CityObjects" property, as is the case in a regular CityJSON file. For example, if one vCityJSON file contains two versions of a building, then two city objects will be in the file as follows:

```
{
  ... // Start of CityJSON
  "CityObjects": {
    "building1": {
      "type": "building",
      "geometry": [ ... ]
    },
    "building1-renovated": {
      "type": "building",
      "geometry": [ ... ]
    },
  },
  ... // Rest of CityJSON
}
```

Every version of a city object has to have a different id in the file (in this example, they ids are "building1" and "building1-renovated"). Identifiers do not need, and are in fact not intended, to have a semantic or functional meaning in order for the versioning to work. In Section 6 we discuss potential candidates for generating identifiers, but different applications may have different requirements. Authors of city models may desire to implement their own strategy in relation to identifiers.

All versions of the city model are defined in the "versioning" property. This contains three properties: "versions", which defines the versions of a city model; "tags", which defines tag names for specific versions of a city model; and "branches", which defines the branches of the repository.

Under the "versions" property, all versions are listed with their metadata. A typical example of two versions, related to the building described earlier, would be:

```
{
  ...
  "CityObjects": {
    ... // Here we define "building1" and
    "building1-renovated"
  }
  "versioning": {
    "versions": {
      "version01": {
        "author": "John Doe",
        "date": "2019-03-04T18:34:12.24Z",
        "message": "First version of the city
     model with the building",
        "objects": [
          "building1"
        ]
      },
      "version02": {
        "author": "John Doe",
        "date": "2019-03-05T18:34:12.24Z",
        "message": "Building1 was renovated",
        "parents": [ "version01" ],
        "objects": [
          "building1-renovated"
        ]
      }
    }
  }
  ...
}
```

In this example, two versions are defined: "version01" and "version02". Every version has the "author", "date" (acting as a timestamp) and "message" property. Additionally, the ids of the city objects that exist in every version are defined in the "objects" array. Finally, every version defines the previous one (the one that it derives from) through the "parents" property.

It is important to emphasise that, similarly to city objects, the names of versions do not have a semantic or functional meaning. A city model's lineage is not defined by the name or date, but instead through the "parents". Normally, every version is expected to have one parent from which it derives. There are only two exceptions: the initial version of a city model, which has no parents; and a "merged" version which occurs when changes between two branches are merged and result in two parents (one for each branch).

The above description defines a DAG structure, similar to that of Git's internal structure (Section 2.1.1), to store a city model's versions. A traversal of versions in such a structure can only occur in a last-to-first order. Therefore, a version has to be denoted as the last one in order to populate the history of a city model by following its parents. For this reasons, branches or tags can be used.

Branches are effectively names that point to the last version of a specific chain of versions. This is defined through the "branches" property. Every branch is a key-value pair where the key is the name of the branch and the value is the name of the last version of this branch.

Tags can be used in order to signify versions that correspond to milestones or may have a significant meaning. For example, if a municipality is working on a city model for an extended period of time and chooses to release a version of it to the public, the version that was released can be named according (e.g. "Version 2019" or "First public release"). Tags are defined under the "tags" property as key-value pairs, similar to branches.

4. VERSIONING EXAMPLES

In this section we describe two examples of real-world changes and how they can be tracked by utilising versioning in CityJSON, as described above. The first example demonstrates the usage of versions in tracking the lineage of a building and its ownership. This demonstrates how a city object can be added, changed and then removed from the 3D city model. The second example illustrates the usage of versioning in order to describe temporal changes to the state of city objects. This example shows how a city object can change to one state and then revert back to a previous one.

4.1 Building lineage example

In this example we describe three major events that occur regarding a building and how they can be stored using versioning. We assume that this city model already contains 27 versions (named "v27", "v26", etc.) and they are all in a single branch named "master". For simplicity, no tags are used in this example.

First, the building is constructed with one storey above ground in 1973 and the ownership is assigned to "Bilbo Baggins":

```
{
...
  "CityObjects": {
    ...
      "building01-01": {
        "type": "Building",
        "attributes": {
          "storeysAboveGround": 1,
          "storeysBelowGround": 0,
          "yearOfConstruction": 1973,
          "owner": "Bilbo Baggins",
        },
        "address": {
          "CountryName": "Eriador",
          "LocalityName": "The Shire",
          "ThoroughfareNumber": "1",
          "ThoroughfareName": "Bag End",
          "PostalCode": "4DV3N TUR3"
        }
      }
    ...
  }
...
  "versioning": {
    "versions": {
      "v28": {
        "author": "J.R.R. Tolkien",
        "date": "1954-07-29",
        "message": "Add Baggins' new building",
        "parents": [ "v27" ],
        "objects": [
          "building01-01",
          ... // all other city object ids from
     v27
        ]
      },
      ... // all other 27 versions here
    },
    "branches": {
      "master": "v28"
    }
  }
}
```

The building is introduced to the city model and a new version, named "v28", is added. This version has "v27" as its parent and it has the same objects as before, plus the newly introduced building. Finally, the "master" branch is changed to now point to "v27".

Next, there is a change in ownership, as well as a change to the building structure. To represent these changes in the city model, we first introduce the city object with the new state under "CityObjects":

```
"building01-02": {
    "type": "Building",
    "attributes": {
      "storeysAboveGround": 2,
      "storeysBelowGround": 1,
      "yearOfConstruction": 1973,
      "owner": "Frodo Baggins",
    },
    "address": {
      "CountryName": "Eriador",
      "LocalityName": "The Shire",
      "ThoroughfareNumber": "1",
      "ThoroughfareName": "Bag End",
      "PostalCode": "4DV3N TUR3"
    }
}
```

Then, the new version, named "v29", is added to the "versioning" property:

```
"versions": {
  "v29": {
    "author": "J.R.R. Tolkien",
    "date": "1954-11-11",
    "message": "Change ownership and structure of
     building01",
    "parents": [ "v28" ],
    "objects": [
      "building01-02",
      ... // all other city object ids from v28
     except "building01-01"
    ]
  },
  ... // all other 28 versions here
},
"branches": {
  "master": "v29"
}
```

Similar to the previous step, a new version object is added which is derived from the previous version, which was "v28". Additionally, the "master" branch now moves to the newly created version.

Finally, the building is demolished and we wish to stop tracking it. Therefore we simply add a new version without the building in it at all:

```
"versions": {
  "v30": {
    "author": "J.R.R. Tolkien",
    "date": "1955-10-20",
    "message": "Remove building01",
    "parents": [ "v29" ],
    "objects": [
      ... // all other city object ids from v29
     except "building01-02"
    ]
  },
    ... // all other 28 versions here
},
"branches": {
  "master": "v30"
}
```

4.2 Temporary close of road example

In this example, we demonstrate the use of versioning in order to describe a road that closes temporarily. We assume that this city model already contains one single version, named "20180505", in the single branch "master". Initially, the model contains the road with its specification:

```
{
  ... // Beginning of CityJSON
  "CityObjects": {
    "road01-01": {
        "type": "Road",
        "attributes": {
            "speed": 50
            "unit": "km"
            "access": true
        }
    }
  },
  "versioning": {
    "versions": {
      "20180505": {
        "author": "John Doe",
          "date": "2018-05-05T00:00:00.00Z",
          "message": "Initial version with open
     road",
          "objects": [
            "road01"
          ]
      }
    },
    "branches": {
      "master": "20180505"
    }
  }
  ... // Rest of CityJSON
}
```

When the road closes, a new version of the city object is introduced in "CityObjects"

```
... // Inside "CityObjects"
road01-closed": {
  "type": "Road",
  "attributes": {
    "speed": 50,
    "unit": "km",
    "access": false
  }
}
...
```

Then a new version that replaces "road01" with "road01-closed" is added and the "master" branch is pointing to this one:

```
... // Inside "versioning"
"versions": {
  "20190123": {
    "author": "John Doe",
    "date": "2019-01-23T11:11:11.11Z",
    "message": "Change road01 to closed access",
    "objects": [
      "road01-closed"
    ]
  },
  "20180505": {
    "author": "John Doe",
    "date": "2018-05-05T00:00:00.00Z",
    "message": "Initial version with open road",
    "objects": [
      "road01"
    ]
  }
},
"branches": {
  "master": "20190123"
}
...
```

Finally, the road re-opens. As there is already a version of "road01" with the same attributes, we don't have to introduce it again. Instead, we just add a new version where we replace "road01-closed" with "road01":

```
... // Inside "versioning"
"versions": {
  "20190124": {
    "author": "John Doe",
    "date": "2019-01-24T22:22:22.22Z",
    "message": "Change road01 to open again",
    "objects": [
      "road01"
    ]
  },
  "20190123": {
    "author": "John Doe",
    "date": "2019-01-23T11:11:11.11Z",
    "message": "Change road01 to closed access",
    "objects": [
      "road01-closed"
    ]
  },
  "20180505": {
    "author": "John Doe",
    "date": "2018-05-05T00:00:00.00Z",
    "message": "Initial version with open road",
    "objects": [
      "road01"
    ]
  }
},
"branches": {
  "master": "20190123"
}
...
```

This example highlights the ability of the proposed data structure to re-use versioned city objects when an object reverts back to its original state.

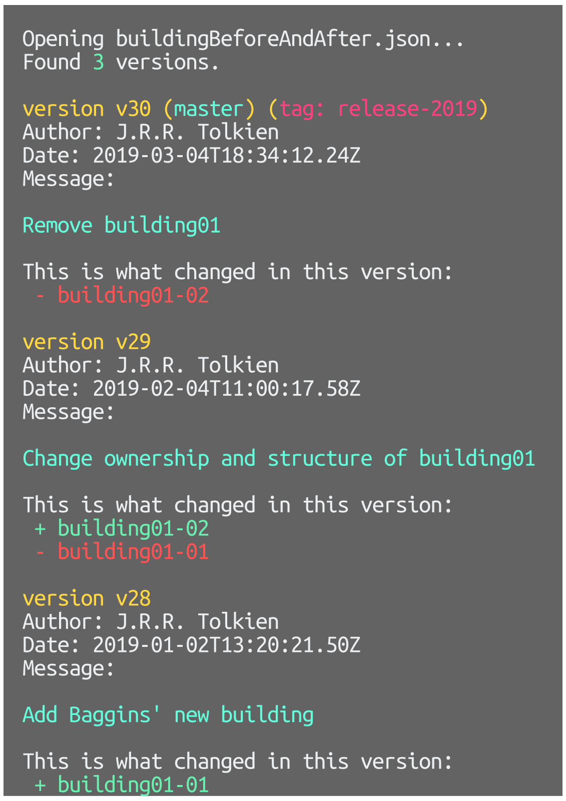
5. PROTOTYPE IMPLEMENTATION

In order to validate the completeness of the proposed solution we have developed a prototype Python 3 script[7]. The script can interact with a vCityJSON and execute two operations: `log`, which prints the history of versions of a vCityJSON; and `checkout`, which extracts a regular CityJSON for a specific version of a vCityJSON.

The `log` command shows the history of a branch in the vCityJSON to the terminal output (Figure 4). The successful execution of such an operation validates the completeness of the proposed solution. Therefore, it is possible to build the history of a 3D city model by using the data structure described in Section 3.2.

The `checkout` command extract a regular CityJSON file to reflect a specific version of the vCityJSON provided. For example, by applying checkout for v30 to the first example (Section 4.1) the output is:

[7]https://github.com/tudelft3d/cityjson-versioning-prototype



```
Opening buildingBeforeAndAfter.json...
Found 3 versions.

version v30 (master) (tag: release-2019)
Author: J.R.R. Tolkien
Date: 2019-03-04T18:34:12.24Z
Message:

Remove building01

This is what changed in this version:
 - building01-02

version v29
Author: J.R.R. Tolkien
Date: 2019-02-04T11:00:17.58Z
Message:

Change ownership and structure of building01

This is what changed in this version:
 + building01-02
 - building01-01

version v28
Author: J.R.R. Tolkien
Date: 2019-01-02T13:20:21.50Z
Message:

Add Baggins' new building

This is what changed in this version:
 + building01-01
```

Figure 4. Output of the `log` operation of our script

```
{
    "type": "CityJSON",
    "version": "1.0",
    "CityObjects": {
        "building01-02": {
            "type": "Building",
            "attributes": {
                "storeysAboveGround": 2,
                "storeysBelowGround": 1,
                "yearOfConstruction": 1973,
                "owner": "Frodo Baggins",
            },
            "address": {
                "CountryName": "Eriador",
                "LocalityName": "The Shire",
                "ThoroughfareNumber": "1",
                "ThoroughfareName": "Bag End",
                "PostalCode": "4DV3N TUR3"
            },
            "bbox": [ 84710.1, 446846, -5.3,
    84757.1, 446944, 40.9 ]
        }
    }
}
```

This validates the completeness of a version's internal representation of objects.

6. DISCUSSION

In this paper we propose a methodology for the representation of 3D city model versions in a CityJSON file. Our approach aims to incorporate the successful characteristics of Git's data structure in the context of CityJSON. Therefore, it focuses on only solving the problem of storing the versions of multiple city objects in CityJSON.

Beyond the two examples presented in Section 4 there are further possible applications: building modifications (e.g. changing the shape of a roof), fixing erroneous information in the model (e.g. misplaced boundary line), adding detail to a model, adding a newly generalised version of the model. etc.

The proposed solution is semantic-agnostic, because there are two aspects of tracking real-world changes that cannot be addressed with a general purpose approach: (i) the association between city objects and their real-world counterparts (e.g. using a cadastral id); and (ii) temporal semantics (i.e. information about why or when a real-world change occurred). Conversely, those problems are rather dependent on the application for which a 3D city model is developed for. Attempting to provide a complete model for all of these aspects could result in an overcomplicated and verbose solution that would be difficult to adapt by practitioners.

Although we chose not to address the problem of associating city objects with their real-world counterparts, certain best practices about how this can be done can be investigated in the future. This is also related to the question of how to assign city objects and version identifiers in vCityJSON which, as was previously mentioned in Section 3.2, do not have a semantic or functional meaning. In our examples (Section 4) we demonstrated two simple solutions: both using a real-world name and appending it with either an iterator or a semantic meaning. There is a plethora of other options that can be investigated by practitioners based on the requirements of a specific application.

We would like to specifically emphasize one potential common identifier that can be utilised for vCityJSON city objects and versions: hashes. This is the same approach used by Git's internal data structure for the representation of commits and versioned files. Hashes not only ensure that no duplicate identifiers are present in the file, but can provide an additional benefit for the validation of data.

We believe our proposed solution can be developed into a CityJSON extension or incorporated in a future version of the specification. Furthermore, it can open many possibilities for adding additional functionality in software that supports CityJSON, such as azul[8]. It would be possible to allow users of 3D viewer to navigate through multiple versions of a city model and identify differences between them. In addition, it makes it possible to write software that tracks the lineage of a specific city object throughout the history of the city model.

In the future, we would like to investigate the specification of versioning operations in vCityJSON, such as forking, merging, comparing versions and highlighting conflicts. Furthermore, a best-practice list, for elements such as commit messages, will be addressed in the future. A software that would interact with the user to allow them to *checkout* from a vCityJSON to a regular CityJSON and *commit* back, as described in 3.1, would be beneficial for practitioners.

ACKNOWLEDGEMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677312 UMnD).

[8]https://github.com/tudelft3d/azul

REFERENCES

Airaksinen, E., Bergström, M., Heinonen, H., Kaisla, K., Lahti, K., Suomisto, J., 2019. The Kalasatama digital twins project—The final report of the KIRA-digi pilot project. Technical report, City of Helsinki.

Baig, S. U., Rahman, A. A., 2012. Generalization and Visualization of 3D Building Models in CityGML. *Lecture Notes in Geoinformation and Cartography*, 63–77.

Ball, T., Kim, J.-M., Porter, A. A., Siy, H. P., 1997. If your version control system could talk. *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 11.

Biljecki, F., Ledoux, H., Stoter, J., 2016. An improved LOD specification for 3D building models. *Computers, Environment and Urban Systems*, 59, 25–37.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., Çöltekin, A., 2015. Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information*, 4(4), 2842–2889.

Chaturvedi, K., Smyth, C. S., Gesquière, G., Kutzner, T., Kolbe, T. H., 2017. Managing versions and history within semantic 3d city models for the next generation of citygml. *Advances in 3D Geoinformation*, Springer, 191–206.

Grune, D. et al., 1986. *Concurrent versions systems, a method for independent cooperation*. VU Amsterdam. Subfaculteit Wiskunde en Informatica.

Ledoux, H., Ohori, K. A., Kumar, K., Dukai, B., Labetski, A., Vitalis, S., 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(4).

Loeliger, J., McCullough, M., 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

OGC, 2019. CityGML 3.0 Conceptual Model. https://github.com/opengeospatial/CityGML-3.0CM.

Open Geospatial Consortium, 2012. OGC City Geography Markup Language (CityGML) Encoding Standard 2.0.0.

Pilato, C. M., Collins-Sussman, B., Fitzpatrick, B. W., 2008. *Version Control with Subversion: Next Generation Open Source Version Control*. " O'Reilly Media, Inc.".

Spinellis, D., 2005. Version control systems. *IEEE Software*, 22(5), 108–109.

Spinellis, D., 2012. Git. *IEEE Software*, 29(3), 100–101.

Stoter, J., Roensdorf, C., Home, R., Capstick, D., Streilein, A., Kellenberger, T., Bayers, E., Kane, P., Dorsch, J., Woźniak, P., Lysell, G., Lithen, T., Bucher, B., Paparoditis, N., Ilves, R., 2015. *3D Modelling with National Coverage: Bridging the Gap Between Research and Practice*. Springer International Publishing, Cham, 207–225.

Tichy, W. F., 1985. RCS—a system for version control. *Software: Practice and Experience*, 15(7), 637–654.