URBANCO2FAB: COMPREHENSION OF CONCURRENT VIEWPOINTS OF URBAN FABRIC BASED ON GIT

John Samuel[1], Sylvie Servigne[2], Gilles Gesquière[3]

[1]CPE Lyon, Université de Lyon, France - john.samuel@cpe.fr
[2]Université de Lyon, INSA-Lyon, LIRIS, UMR5205, F-69621, France - sylvie.servigne@liris.cnrs.fr
[3]Université de Lyon, LIRIS, UMR5205, F-69621, France - gilles.gesquiere@liris.cnrs.fr

KEY WORDS: Urban Fabric, Version Control Systems, City Information Model, Process Documentation

ABSTRACT:

The study of urban evolution requires representation and management of concurrent points of view using interoperable standards. An extension to CityGML has been recently proposed (Chaturvedi et al., 2017) to represent urban evolution. In this article, we further extend this work to represent different possible scenarios . The key characteristic of our approach is that it makes the most of existing technologies like version control system so that users can easily work with our proposed extension. Our approach also lets the users add and even modify information to versions and version transitions of scenarios in a given workspace.

1. INTRODUCTION

Cities undergo rapid changes. Some of them develop at a very past pace whereas some others face gradual deterioration. Towards understanding and developing sustainable and durable cities, both the categories of cities are being studied by historians and urban planners. Lessons from the past successes and failures serve as a guidance for the future planning of the cities. But what may seem as a failure by one may be considered as a changing point by another. Therefore understanding urban evolution requires the ability to represent and manage such different points of view.

2D maps have been used in the past to represent the urban footprints of the cities. Color, textures and even multiple maps have been used to compare the changing urban footprints. Models and simulations have also been created by archaeologists to understand the historical evolution of cities. The 4D visualization of Bastion fort (Rizvic et al., 2015), digital reconstruction of the city of Pompeii (Dell'Unto et al., 2013) and even the relief maps of the past augmented with various video displays (Priestnall et al., 2012) are commonly cited examples demonstrating the historical evolution. However their focus has been limited. Take for example, relief map models once built, offer very little scope for subsequent modifications. Similarly, focus on a particular zone is helpful for understanding its evolution, but cannot be extended to other zones.

With the growing usage of internet, desktop and handheld devices, several technologies are now available to build virtual urban models (Pfeiffer et al., 2013). It has become much easier to build large scale flexible city models in such a virtual environment compared to their equivalents in the real world. Adding to this, thanks to the growing availability of international standards for urban models like CityGML (Gröger et al., 2012), the focus has now shifted to building and sharing city models in an interoperable manner. Several libraries and applications are now currently available that can render 3D city objects like buildings, bridges, roads etc in an efficient manner. These developments have brought great hope in building generalized solutions that can be easily maintained and extended to very large scales.

However, rebuilding the past of a city is not an easy task. Historians and researchers all around the world have been working on the study of evolution making use of historical artefacts. For e.g., the Venice time machine project (Kaplan, 2015) aims to study 1000 years of Venice by building a very large document corpus by digitizing historical archives. Numerous documents are available today that enable them to virtually reconstruct one or more city objects and even reconstruct an entire town or city. These documents include administrative documents like project plans, aerial views, municipal council meetings and multimedia documents like videos, photographs, old postal cards, paintings etc. Some of these documents are evidences to the actual buildings of the past whereas others may be artistic renderings. Historians have found that unsuccessful projects that never came to full fruition have left lasting impacts on the cities. While studying the urban evolution, these traces are used to comprehend various socio-economic and political aspects of the given period. Urban planners and historians need simplified ways (Samuel et al., 2016) to represent, manage and share information concerning changes of city objects.

Much recently, CityGML has been extended in order to represent changes (Chaturvedi et al., 2017) in features of city objects. In this article, we present a proof-of-concept called UrbanCo2Fab by taking into consideration the above extension and further propose a simplified way to represent multiple viewpoints of urban evolution. In order to reduce the learning curve of the users of UrbanCo2Fab, we focus on using version control systems and their vocabulary to manage the relevant information. The goal is that any first-time user of the application does not need to waste a lot of time in getting acquainted to its different capabilities. For example, by making use of terms like versions, scenarios (branches), we ensure that the users are quickly able to understand the objectives of the tool.

In this article, we will discuss in detail the problem of representing and storing the different scenarios of urban evolution. We will also detail the implementation of our approach using existing version control system. Section 2 will present the existing state of the art and their limitations. Section 3 will present in detail UrbanCo2Fab, describing the various options to manage urban data and its evolution. We will also show the limitations of the GIT, a version control systems for studying the urban evolution. We also explain how we mapped our requirements to GIT and its options. We will see the implementation of the proof of concept in section 4. Section 5 presents the results using an example and we conclude our article in section 6, briefly describing our future course of actions.

2. STATE OF THE ART

Last several years have seen the growing focus on the use of desktop as well as mobile platforms and internet technologies for building virtual urban environment (Pfeiffer et al., 2013). A number of mapping services are now available with the capability to visualize 2D, 2.5D and 3D urban data. Incorporation of timeline (i.e., support of the fourth dimension, time) to these services help users to navigate through the historical past of a given area at any available period of time. Projects like Bastion 4D visualization project (Rizvic et al., 2015), Virtual Kyoto project (Yano et al., 2008), have demonstrated this capability to document and even visually narrate the historical evolution of important places. Nevertheless, all these works have focused on giving a 'one-shot' video, useful to get a single narration of urban evolution obtained after the consensus of different researchers working on the particular topic.

The goal of geographical information is not to limit to just four dimensions (Kaplan, 2015), but to be available to add additional information from diverse sources. These diverse sources may lead to alternate conflicting narratives. Many of these narratives are lost to the future generation. Therefore, we must be able to build a generic method that can be used to represent and organize urban evolution data, that can be used by several projects (and not just to create a one-time 3D video) and also give other viewpoints.

There is a growing demand by historians and even urban planners for easily manageable interoperable solutions for geographical information (De Roo et al., 2013). One possible way is to use and promote international standards. CityGML[1] is one such standard proposed by Open Geospatial Consortium (OGC) that can represent cities in 3D format. 3D urban data CityGML files for several major cities like Lyon, Berlin, New York are currently available.

CityGML is an evolving standard that has been extended[2] in several ways to incorporate several information like indoor facilities (Kim et al., 2014), dynamic properties(Chaturvedi and Kolbe, 2015), cultural heritage (Finat et al., 2010) etc. One recent work proposed an extension to represent urban evolution (Chaturvedi et al., 2017) by focusing on changes in features of city objects. But this work is limited in its capability to represent multiple points of views of urban evolution. Study of urban evolution requires representation and management of multiple points of view of historians and urban planners. In this article, we inspire from the work done by (Chaturvedi et al., 2017), particularly their proposition to use transaction timestamps and object existence timestamps in the real world, which in turn are inspired from the INSPIRE model (Craglia and Annoni, 2007) model. Our proposed approach tries to stay CityGML 2.0-compliant. With very little modification to CityGML 2.0, we demonstrate how urban evolution can be represented and even shared.

[1]https://www.citygml.org/
[2]https://www.citygml.org/ade/

Collaborative approaches in geographical information systems are not new. OpenStreetMap (Haklay and Weber, 2008) is a commonly cited example, where users edit maps adding, discussing and finally coming into a consensus on the final information to be shown. It does keep track of the user edits. However, once again, the users are not able to easily see other points of view, i.e., the users are only exposed to view obtained by consensus.

The ability to see and work with multiple possible scenarios is a common problem in software development. Version control systems (Spinellis, 2005) are now commonly used to develop softwares, which let users independently develop programs. Developers can switch to alternate view of development of code, for example, addition of new features. They review the code, come to a consensus and merge the proposed changes on the official branch, usually referred to as the master brunch or even trunk. Distributed version control systems (Milewski, 1997) even let users fork entire repositories, thereby letting multiple concurrent development by various business players. These solutions on first look may seem to match the requirements for managing information related to urban evolution. However, existing solutions including the latest version control system (VCS) like GIT (Loelinger and MacCullogh, 2012), Pijul[3] (inspired from categorical theory of patches (Mimram and Giusto, 2013)) have focused on managing line based changes. This line-based tracking approach, now almost a standard way is helpful in software development since VCS must cater to the needs of developers in different programming languages with varying syntax. Line-based differences between two codes targets human coders. This is not very useful to understand differences between city objects represented by CityGML (a structured data format using XML or JSON) that have undergone changes.

Geogig[4], an application for distributed versioning of geospatial data, tackles this problem for structured geographic data. It makes use of GIT vocabulary and shows the feature differences between two versions in a very user-friendly manner. One particular advantage of Geogig is that anybody familiar with GIT can use it. However, it does not currently support CityGML files and also it focuses on tracking user-made changes to geographical data files. It tracks only the transaction timestamps corresponding to the time when the user changes are actually stored in the repository. But the study of historical evolution requires working with historical dates. It also requires management of evidences and the proposed labels to significant events to support historians' claims of their different hypotheses of evolution proposed. In short, any proposed solution firstly must go beyond two timestamps, i.e., the beginnig and end of commit transaction time and must also ensure existence time of objects in real world. Secondly, GIT pointers track the changes and it is not possible to insert new versions between two already created versions, an important requirement while implementing evolving study of scenarios. Thirdly, it must be able to specify and even modify labels to events. Finally, it is not possible to directly use GIT unidirectional pointers for navigating the past and future of an urban area with respect to a given observation time.

In this article, we present UrbanCo2Fab, an application for comprehension of urban fabric for management of different points of view of city evolution. Its goal is to manage multiple concurrent scenarios of urban evolution. Historians can use it to represent different urban projects and study the impact on their proposition

[3]https://pijul.org/
[4]http://geogig.org/

in urban development. It is able to represent consensus scenario and multiple proposed scenarios of urban evolution. All these aspects have been detailed below.

3. URBANCO2FAB

A city object like a building undergoes several changes: from an initial empty plot to its construction to any possible modification and finally its possible destruction. These different states can be captured by versions shown in Figure 1. Consider the Figure 1 as the evolution of an administrative building between 1950 and 1972. There are five versions: *V1, V2, V3, V4* and *V5*. The duration of each version, i.e., the existence period of each version has been shown by the grey block. Every version has a starting period and a final period. Associated to each version, there is a label proposed by the user. For example, *V1* has an associate label *Building constructed*. Users can also propose a different label.

Period between two versions is called a version transition. In the Figure 1, there are four version transitions: *VT1, VT2, VT3* and *VT4*. These can also be referred by using version identifiers. I.e., *VT1, VT2, VT3* and *VT4* can also be referred to as *V1-V2, V2-V3, V3-V4* and *V4-V5* respectively. Like versions, version transitions can also be labeled. For example, the period between *V1* and *V2* is labeled as *Construction phase*, referring to the construction of a new floor. Version transitions also have a validity period. *VT1* is valid between 1957 and 1958.

A scenario is a linear narration of changes of one or more city objects. Therefore Figure 1 shows one possible scenario of construction, modification and ultimate destruction of a building. This scenario consists of *V1, V2, V3, V4, V5* with its associate version transitions *VT1, VT2, VT3, VT4* in the given order.

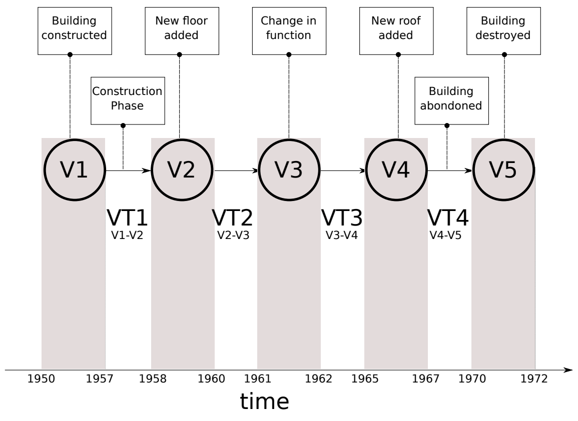

Figure 1. Versions and Version Transitions

There can be any number of versions depending on the evidences available. Take for example photographs showing the number of floors of a 10 storeyed building at different periods of time may be used to represent 10 different versions of a building. Another user, may not be interested in going in detail to such a granular level, may refer to only 2 versions: 0-floor building and 10-floor building. Thus, it is left to the user making use of the available evidences and the problem at hand for deciding the number of versions that need to be created. Nevertheless, a version, may be considered as a snapshot of one or more city objects at a given point of time or a state of one or more city objects during a given period of time.

In order to understand the details of a version, we start by first considering city objects. CityGML for example has different types of city objects, like building, bridge, vegetation, transportation etc. Every city object has one or more features. For example, a building or a bridge has a texture. As shown in Figure 2 consider a building with four features: its function, the number of floors, roof type and textures. We futher elaborate the example in Figure 1 with additional details. Feature values may change from time to time. Versionable features (Chaturvedi et al., 2017) capture these changes. In Figure 2, feature 'Function' has undergone changes three times, in version *V1*, it acted as as a public building, it had undergone a change in version *V3*, where it was used as a private appartment. Finally this building was abandoned after version *V4*. There are ten timestamps t1 to t10. Versions are shown by grey vertical boxes and cover a given time period.

Feature values also have a time period, i.e., the period during which the value of the given is valid. But there may be cases, when we do not have any information about a feature value. Like in version *V1*, we do not have the value for the feature 'Roof Type'. This is possible when we do not have any evidences concerning this value. Similarly, it is also possible, that a feature value is empty, even though we know its previous values. This is the case for the feature 'Texture' whose value was known for versions *V1* and *V2*, but not thereafter.

Whenever any of the feature value changes, either a version transition starts or a new version is created. This can be clearly seen in the given example. When the validity period of feature 'Number of storeys' is over, version *V1* is terminated and the version transition *VT1* starts. A set of feature value changes in a given version transition is called a **transaction**. Take for example, in version transition VT3, there are two transcations: change in roof type and change in the number of storeys.

Thus versions or version transitions are dependent on the feature value changes. A versionable feature therefore has two associate timestamps (existence in the real world): start date and end date for a feature value. Figure 3 can be generalized to more city objects, where features F1 to F4 may belong to one or more city objects. Thus, we assume that the features are uniquely identified.

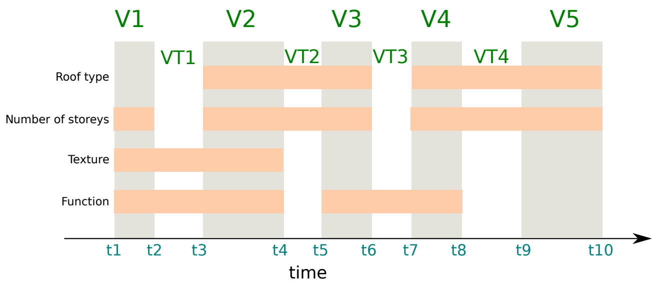

Figure 2. Change in features of a building. Every version V1-V5 has an associate existence time shown on the timeline. Versions are shown by grey vertical boxes. The validity of existence of a feature value is shown by the light-orange boxes.

Consider the scenario consisting of versions *V1, V2, V3, V4, V5*.

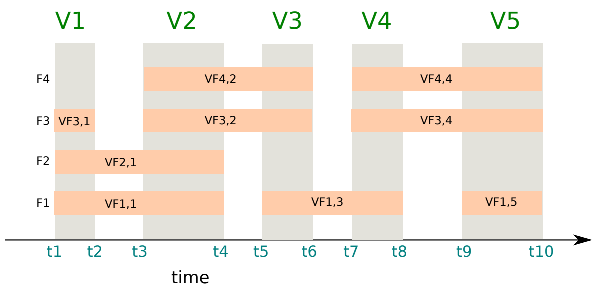

Figure 3. Versionable features of city objects. *F1, F2, F3* and *F4* are unique identifiers of features of city objects.

Let us assume that there is a consensus among all the researchers that these states of city objects existed, because of enough material evidences. These versions and associate version transitions are part of the space called **consensus space**. However, some historians propose a scenario of transition from *V1* to *V5* through versions *V7* and *V8*. Another group of historians propose a scenario of transition through version *V6*. There are not enough evidence to justifify their physical existence, hence they cannot be part of the consensus space. These two possible scenarios consisting of versions *V1, V7, V8, V5* and *V1, V6, V5* become part of the **propositions space**.

A workspace is a virtual space for the study of urban evolution used by one or more users to propose and save different possible scenarios. It consists of two spaces: consensus space and propositions space. A consensus space consists of only one scenario of versions and a consensus has been made that such a scenario existed. propositions space, on the other hand consists of one or more scenarios proposed by different historians, but a consensus has still not been made. It is possible that once a consensus has been made, a scenario from the propositions space enters to the consensus space. Furthermore, it is also possible to move one or more versions from a scenario in propositions space to consensus space.

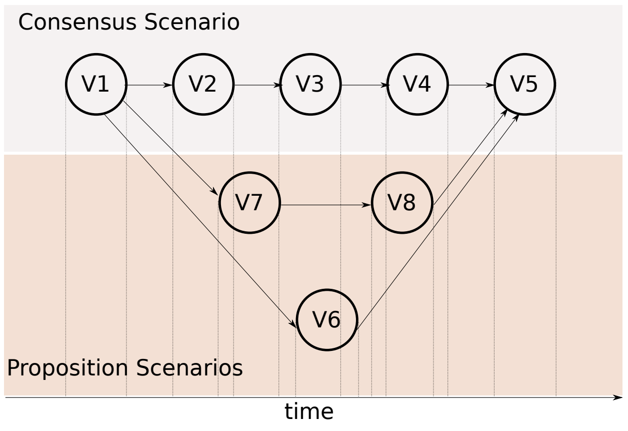

Figure 4. Workspace with two spaces for scenarios: Consensus space and propositions space

Looking at Figure 4, one may have an impression that it is similar to a version control system. This is true to a certain extent. A workspace is analogous to a repository in VCS, scenario is analagous to a branch, a consensus scenario is analogous to a main branch (or trunk) and proposition scenarios are analogous to feature branches or developer branches. This is in fact one of the reasons that we decided to make maximum use of VCS vocabulary for the management of workspaces. However, as discussed above, version control systems have several limitations. A comparison between our proposition, $UrbanCo^2Fab$ Workspaces and distributed version control systems is given in Table 1. As can be clearly seen, VCS does not capture the notion of transactions. $UrbanCo^2Fab$, however captures the changes between two versions (or simply a version transition) through transactions, where every transaction may refer to insertion, deletion or modification of a particular feature value. VCS only tracks the creation time of a commit, whereas, in $UrbanCo^2Fab$, we track the real-world existence of a city object as well as the transaction timestamps (when the object was created and completely saved to a database). Commit messages in VCS cannot be modified once a version has been created[5]. In $UrbanCo^2Fab$, labels can not only be assigned to a version, a version transition and a scenario, but they can also be changed.

4. DEVELOPMENT

Figure 5 shows the basic architecture of $UrbanCo^2Fab$. It is built over GIT as shown in Figure 5 and makes use of git-like commands to reduce the learning curve of new users (with slight changes in vocabulary described in Table 1). It is developed in Python using the Pygit2[6] library and tracks changes in city objects described using CityGML files. Pygit2 is a python interface over GIT letting developers easily access GIT related metadata.

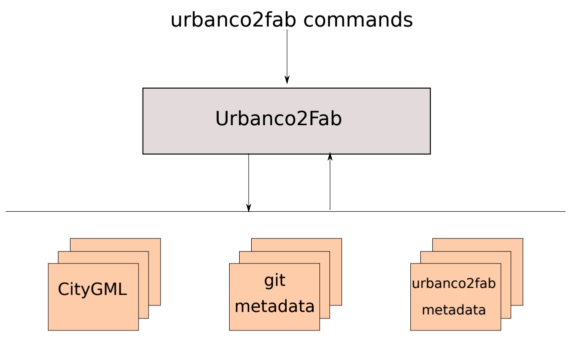

Figure 5. UrbanCo2Fab: Using urbanco2fab commands for managing CityGML files for the study of urban evolution

We use pygit2 to access GIT related metadata, especially the information concerning GIT commits. A commit in a GIT is used to give a timestamped snapshot of one or more files and directories. Every commit has an associate metadata: commit author, author signature (email), commit identifier (SHA1), pointers to previous version (or versions in case of a merge), timestamp (or creation date of a commit) etc. Commit identifier is used as a version identifier in $UrbanCo^2Fab$. Continuing with our example, version

[5] *git commit -m* in GIT lets a user modify a message, but it leads to the creation of a new version and the old version becomes an unreachable version.

[6] http://pygit2.org/

Characteristics	*$UrbanCo^2Fab$ Workspaces*	*Distributed Version Control Systems*
Basic Unit of Change	Feature of city object	Line or byte of a file
Change Tracking	**Transaction** and Version	Version
Transaction Types	Insert, Replace and Delete	N.A
Transaction Timestamps	Existence time and operation transaction times	N.A
Version	Timestamped collection of versionable features and associate transactions	Timestamped repository state (usually after a commit)
Version Timestamps	Existence time and operation transaction times	Time of creation
Version Transition Traversal	backward, **forward**	backward
Tag	User-defined name to version, **scenario and version transition**	User-defined name to version
Concurrent and Independent development	**Space, scenarios and workspaces**	Branches and repositories

Table 1. Comparison between $UrbanCo^2Fab$ Workspaces and Distributed Version Control Systems. Key differences are highlighted

identifiers *V1, V2, V3, V4, V5* correspond to the commit identifiers obtained after performing a 'git commit' operation. This has been shown in Figure 6. Note that for the sake of simplicity, we do not consider highly granular commit timestamps, but rather at monthly level (November 2017, January 2018 etc.) Another point to note that GIT does not have two timestamps for commit: start and end time, but rather one. Please note that the versions of a scenario need not be in a chronological order when considering the commit time. For example, version *V4* was created in November 2017 whereas the version *V3* was created in May 2018, however, while creating a scenario, *V3* appears before *V4*. This is quite normal because scenario versions are created when new evidences are available.

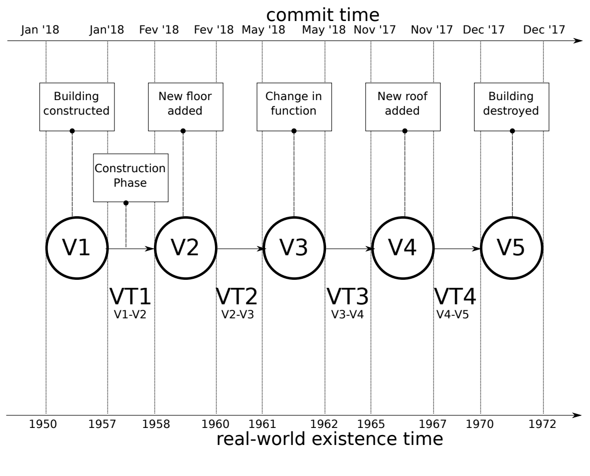

Figure 6. GIT commit time UrbanCo2Fab real-world existence time for versions and version transitions

As shown in Figure 7, UrbanCo2Fab works with two metadata directories: .git directory managed by GIT and .urbanco2fab managed by urbanco2fab. It reads and writes to the .git metadata directory using pygit2 interface whereas it directly interacts with the .urbanco2fab metadata directory, reading, writing and even updating information.

In order to limit the changes to CityGML, we make use of two additional attributes (timestamps): validFrom and validTo to demonstrate our idea. Every city object has an identifier (or a gml identifier). We call it the **major identifier**. This identifier along with the feature name is used to identify any particular feature of a city object and we call it the *minor identifier*. For example, in cases of a city object like a building having an identifier *CO1*, *CO1* is its major identifier and *CO1#function* is the minor identifier of feature 'function' of the building. Whenever there is a change in attribute (or feature value) of a city object, the user changes the feature value and specifies the validFrom and validTo timestamps to specify the validity of a version in the real-world. The user adds these changes and makes a commit. We make use of GIT commit to execute this step.

A commit created by the user corresponds to a version. A commit has several interesting features: commit identifier, commit author, commit message etc. When a user creates a scenario (similar to branch), We check for any new/updated/ versionable features in every CityGML file, verify existence timestamps and save these changes. The verification process ensures that the validity period of all the versionable features in a given scenario is well respected.

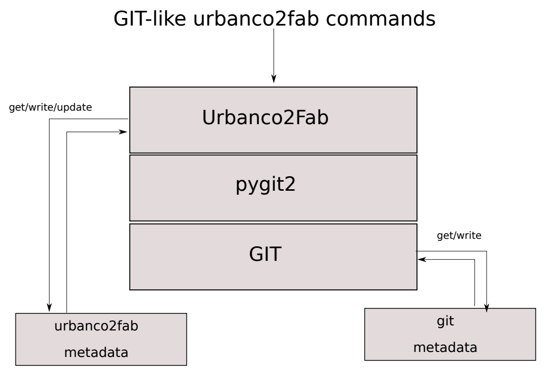

Figure 7. UrbanCo2Fab built over GIT using pygit2

In order to implement $UrbanCo^2Fab$, we now make use of the existing commands of VCS and propose extensions in order to integrate workspaces. Reutilisation of the existing VCS commands is also to reduce the learning curve of users who want to use $UrbanCo^2Fab$. A summary of proposed commands is described in Table 2.

These are the major subcommands *version*, *transaction*, *transition*, *scenario* and *workspace*. *version*, *scenario* and *workspace* have options for their creation, i.e., the users can use *add* option with appropriate parameter values to create a new version, scenario or a workspace. Similarly, these three subcommands also have the option *tag* to specify short phrases.

Note the use of [–time "..."] and [–document "..."] in *commit* option of *version* to specify the existence time of a version and the links to supporting evidences. We see the usage of time once again in *log* option of *transition* to be able to get historical or future list of transactions with respect to the user-specified date.

Following are the major commands of urbanco2fab.

urbanco2fab init initalize a workspace and creates all necessary files and folders required to work with urbanco2fab. Internally it makes use of GIT initialization with additional urbanco2fab information.

```
$ urbanco2fab init .
```

urbanco2fab add adds changes for creating a version. This is similar to git add, where the user changes are staged for commit. **urbanco2fab rm** removes files and **urbanco2fab mv** renames files.

```
$ urbanco2fab add file1 file2
$ urbanco2fab rm file1 file2
$ urbanco2fab mv oldname newname
```

urbanco2fab commit commits changes. This is a very important command of urbanco2fab. On completing this command, a new version is created, if the timestamps are correctly respected.

```
$ urbanco2fab commit -m "new version" --time ‘‘..."
```

creates a new version with the message.

```
$ urbanco2fab commit -m "new scenario" \
  -s "scenario name" -v v1 v4 v3 -vt v1-v4 v4-v3
```

creates a new scenario with three versions v1, v4 and v3 and version transitions v1-v4 and v4-v3.

urbanco2fab clone clones a remote workspace to the current workspace. **urbanco2fab pull** fetches and update changes to the current workspace of the user. **urbanco2fab push** pushes user changes to the remote workspace.

```
$ urbanco2fab clone URL
$ urbanco2fab pull
$ urbanco2fab push
```

urbanco2fab log gives the log of various changes with respect to a given timestamp.

```
$ urbanco2fab log -hs --time  ‘‘..."
```

shows all the versions before the given timestamp.

```
$ urbanco2fab log -fu --time  ‘‘..."
```

shows all the versions after the given timestamp.

urbanco2fab diff: shows the differences between two versions (useful for creating scenarios)

```
$ urbanco2fab diff version1 version2
```

where version1 and version2 are the SHA1 (commit signatures) of the versions and can be obtained from urbanco2fab log command discussed above.

urbanco2fab show shows the details of a scenario, version or a version transition.

```
$ urbanco2fab show -s scenario
$ urbanco2fab show -v v1
$ urbanco2fab show -vt v1-v4
```

urbanco2fab tag tags a version, version transition or a scenario.

```
$ urbanco2fab tag -s s1 -m "my tag"
```

5. RESULTS

In this section, we show the contents of various key aspects of $UrbanCo^2Fab$. Consider a user, working with a CityGML file concerning a building with the identifier *GMLID_BUI357978_1938_268*. The user specifies the function and also the time period of its validity through the command on making a commit. $UrbanCo^2Fab$ parses the CityGML files staged for commit, parses the XML files to find the (GML) identifiers of different city objects to find and save the changes in feature values.

```
<bldg:Building gml:id="GMLID_BUI357978_1938_268">
<bldg:outerBuildingInstallation>
<bldg:BuildingInstallation>
<bldg:function>1075</bldg:function>
...
```

A particular commit (version) in VCS stores only the timestamp at which a commit has been made along with the commit message ($title$) and a hash signature ($identifier$). We also make use of these attributes in $UrbanCo^2Fab$ along with some new attributes. There are two time periods, the transaction time corresponding to when the version was created ($storetransactionstarttime, storetransactionendtime$) and also the existence time ($existencestarttime, existenceendtime$) that stores the existence time of the given version in real-world. $description$ stores any detailed user-generated message and $document$ can be used to store links to a number of documents/evidence that describes the version (photographs, web URLs etc).

In code block 1, we see an example for version $V1$. *tag* are specified by the user and here the user has chosen *Building constructed.* Tags are useful to search certain specific themes. There are two document identifiers pointed to by *document*.

Subcommand	*Summary*
version	
add	Staging the data for the commit
mv	Moving or renaming city model (e.g., CityGML) file(s)
rm	Removing city model (e.g., CityGML) file(s)
commit [–time "..."] [–document "..."]	Commit the changes and create a new version of desired type along with its existence time and the links to supporting evidences
show	Show the details of a commit
tag	Tagging a particular version by user-specified phrases
transaction	
diff	Show the features that have changed between two version (or one transition)
transition	
log [–historical | –future] [–time "..."]	Show the history of changes going back to a particular time or the set of changes that have been made from a particular time (future)
scenario	
commit –version [] –versiontransition []	Create a new scenario with the given versions and version transitions
show	Show the details of a scenario
tag	Tagging a particular scenario by user-specified phrases
workspace	
commit –type [consensus,proposition] -s "..."	add a scenario to a space
tag	Tagging a particular workspace by user-specified phrases
clone	Clone a workspace
init	Initialize a workspace
pull	Fetch and integrate with another workspace
push	Update the changes to another workspace

Table 2. $UrbanCo^2Fab$: Summary of major subcommands

```
{
  "identifier": "V0",
  "title": "Version 0",
  "description": "Building constructed",
  "tag": ["Building constructed"],
  "document": ["doc31", "doc42"],
  "existencestarttime": "1950-01-01",
  "existenceendtime": "1957-01-01",
  "storetransactionstarttime":
      "2018-01-01 12:45:46",
  "storetransactionendtime":
      "2018-01-01 12:45:47",
  "useridentifier": "user1"
}
```

Code Block 1. Example Version in JSON

```
{
  "identifier": "VT1",
  "title": "Construction phase",
  "description":
     "Construction of a new storey",
  "tag": ["Construction phase"],
  "document": ["doc34"],
  "from": ["V1"],
  "to": ["V2"],
  "existencestarttime": "1957-01-01",
  "existenceendtime": "1958-02-01",
  "storetransactionstarttime":
       "2018-02-01 12:00:01",
  "storetransactionendtime":
       "2018-02-01 12:00:01",
   "transaction" : [
    {
      "idenitifier":
        "GMLID_BUI357978_1938_268#storeysAboveGround",
      "type": "Replace",
      "existencestarttime": "1957-01-01",
      "existenceendtime": "1958-01-01",
      "storetransactionstarttime":
         "2018-02-01 12:00:01",
      "storetransactionendtime":
         "2018-02-01 12:00:01",
    },
    ...
  ]
}
```

Code Block 2. Example Version Transactions in JSON format
In code block 2, we represent version transition VT1 (or V1-V2). Version transition stores the complete details of changes between two versions (in the form of transactions). Some features like $identifier, title, description, \ldots$ like those seen in $Version$ above serve the same purpose and are not described again here. $from$ and to are added to support forward and backward navigation of versions in a scenario. A transaction $transaction$ stores the details of changes that occurred to different feature. Compare the existence times of $V1$ in example from code block 1 and $VT1$ in code block 2 as well as the existence time of one of the transactions in VT1 concerning $storeysAboveGround$ of city object. Note the value *Replace* in *type* of the transaction.
A scenario stores the sequence of versions ($versionid$) and version transitions ($versiontransitionid$) defined by the user as shown in code block 3.

```
{
  "identifier": "Scenario23",
  "type": "consensus",
  "title":
    "History of adminsitrative building",
  "description": "Proposed hypothesis
       of changes to Admin building",
  "tag": ["Administration"],
  "storetransactionstarttime":
       "2018-05-10 12:55:46",
  "storetransactionendtime":
       "2018-05-10 12:55:50",
  "versionid" : ["V1", "V2", "V3",
       "V4", "V5"],
  "versiontransitionid" : ["VT1", "VT2",
       "VT3", "VT4"],
}
```

Code Block 3. Example Scenario in JSON format

CityGML 2.0 has two attributes of dateOfCreation and dateOfDemolition, however it is limited to buildings and cannot represent the intermediate states like the example shown above. CityGML 3.0 (Kutzner and Kolbe, 2018) will be released by the end of the year 2018 which will introduce the necessary timestamps for representing the versionable features, versions and version transitions presented in (Chaturvedi et al., 2017). Two timestamps *validFrom* and *validTo* will be able to represent the lifespan of any city object and therefore make the model shareable and interoperable. Our proposed approach can be used in both these versions.

6. CONCLUSION

CityGML 3.0 will be released by the end of this year and the use of timestamps proposed in our proof of concept will soon become part of the versioning module. We demonstrated in our work, how these changes can be further enhanced to represent concurrent points of view of urban evolution. By developing our proof-of-work on GIT, it is also easy to share the evolution related changes. Our next course of actions is to understand and deal with scalability and performance issues for very large scale CityGML files.

ACKNOWLEDGEMENTS

This work was performed within the framework of the LABEX IMU (ANR-10-LABX-0088) of Université de Lyon, within the program Investissements d'Avenir (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

REFERENCES

Chaturvedi, K. and Kolbe, T. H., 2015. Dynamizers - modeling and implementing dynamic properties for semantic 3d city models. In: F. Biljecki and V. Tourre (eds), *Eurographics Workshop on Urban Data Modelling and Visualisation, UDMV 2015, Delft, The Netherlands, November 23, 2015.*, Eurographics Association, pp. 43–48.

Chaturvedi, K., Smyth, C. S., Gesquière, G., Kutzner, T. and Kolbe, T. H., 2017. Managing versions and history within semantically enriched 3d city models. *Advances in 3D Geoinformation, Lecture Notes in Cartography and Geoinformation, Springer*.

Craglia, M. and Annoni, A., 2007. Inspire: An innovative approach to the development of spatial data infrastructures in europe. *Research and theory in advancing spatial data infrastructure concepts* pp. 93–105.

De Roo, B., Bourgeois, J. and Maeyer, P. D., 2013. On the way to a 4d archaeological gis: state of the art, future directions and need for standardization. *Proceedings of the 2013 Digital Heritage International Congress. Vol. 2.*

Dell'Unto, N., Leander, A. M., Dellepiane, M., Callieri, M., Ferdani, D. and Lindgren, S., 2013. Digital reconstruction and visualization in archaeology: Case-study drawn from the work of the swedish pompeii project. In: *2013 Digital Heritage International Congress, Marseille, France, October 28 - November 1, 2013, Volume I*, IEEE, pp. 621–628.

Finat, J., Delgado, F., Martnez, R. and Hurtado, A., 2010. Girapim: A 3d information system for surveying cultural heritage environments. *ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 38(4), pp. W15.

Gröger, G., Kolbe, T. H., C., N. and K. H., H., 2012. OGC city geography markup language (CityGML) encoding standard v2.0. *OGC Doc*.

Haklay, M. M. and Weber, P., 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing* 7(4), pp. 12–18.

Kaplan, F., 2015. The venice time machine. In: C. Vanoirbeek and P. Genevès (eds), *Proceedings of the 2015 ACM Symposium on Document Engineering, DocEng 2015, Lausanne, Switzerland, September 8-11, 2015*, ACM, p. 73.

Kim, Y., Kang, H. and Lee, J., 2014. *Developing CityGML Indoor ADE to Manage Indoor Facilities*. Springer International Publishing, Cham, pp. 243–265.

Kutzner, T. and Kolbe, T., 2018. Citygml 3.0: Sneak preview. In: *PFGK18 - Photogrammetrie - Fernerkundung - Geoinformatik - Kartographie*.

Loelinger, J. and MacCullogh, M., 2012. *Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development: Covers GitHub, Second Edition*. O'Reilly.

Milewski, B., 1997. Distributed source control system. In: R. Conradi (ed.), *Software Configuration Management*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 98–107.

Mimram, S. and Giusto, C. D., 2013. A categorical theory of patches. *Electr. Notes Theor. Comput. Sci.* 298, pp. 283–307.

Pfeiffer, M., Carré, C., Delfosse, V., Hallot, P. and Billen, R., 2013. Virtual leodium: from an historical 3d city scale model to an archeological information system. *ISRP Annals of Photogrammetry, 2-5/W1*.

Priestnall, G., Gardiner, J., Durrant, J. and Goulding, J., 2012. Projection augmented relief models (PARM): tangible displays for geographic information. In: S. Dunn, J. P. Bowen and K. Ng (eds), *Electronic Visualisation and the Arts, EVA 2012, London, UK, 10-12 July 2012*, Workshops in Computing, BCS.

Rizvic, S., Okanovic, V. and Sadzak, A., 2015. Visualization and multimedia presentation of cultural heritage. In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pp. 348–351.

Samuel, J., Périnaud, C., Servigne, S., Gay, G. and Gesquière, G., 2016. Representation and visualization of urban fabric through historical documents. In: *14th EUROGRAPHICS Workshop on Graphics and Cultural Heritage*.

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

Yano, K., Nakaya, T., Isoda, Y., Takase, Y., Kawasumi, T., Matsuoka, K., Seto, T., Kawahara, D., Tsukamoto, A., Inoue, M. and Kirimura, T., 2008. Virtual kyoto: 4d gis comprising spatial and temporal dimensions. *Journal of geography, 117* pp. 464–478.