Automatic Semantic Comparison of STEP Product Models
Application to IFC product models

G. Arthaud and J.C. Lombardo¹
Ecole Nationale des Ponts et Chaussées
¹ Centre Scientifique et Technique du Bâtiment

Keywords: Industry Foundation Classes (IFC), STEP models, EXPRESS language, Semantic comparison, Design process

Abstract: This paper introduces an original method to compare IFC models and more generally any STEP models. Unlike common “diff-like” tools which compare textual files by proceeding line against line, our approach compares actual graphs created from STEP-files. Therefore added, removed, and changed objects can be tracked between two versions of the model. Besides, this standalone tool does not need any heavy database to work so it is fully adapted to design methods of construction projects, where actors are free to modify a local version of their project without any dependence on the database. Moreover it is reusable for other industrial fields thanks to its compatibility with any STEP model. This tool is a part from a more global project which tends to improve accessibility and sustainability of IFC therefore it can be used as a support for VR based design tools.

1. INTRODUCTION

Our team carries out research on Industry Foundation Classes visualization and real-time simulations in virtual environments, in order to improve accessibility and sustainability of IFC. We investigate a major issue in this paper which delays the use of IFC in the AEC community: comparison and merging of IFC models. After a short reminder of IFC and design process in a construction project, we will address the main problem of our research: semantic comparison of product models.
1.1 Interoperable Project

In normal practice, information is exchanged between engineers, architects and clients, in the form of verbal and hardcopy programs, sketches, diagrams and drawings. This often requires interpreting, re-documenting and re-entering information into software and systems to make it usable for the project team. Great loss and corruption of data may occur during this translation.

Thanks to the Industry Foundation Classes data model specified by the International Alliance for Interoperability, software can exchange standardized product models during a civil engineering project. This standard avoids multiple specialized interfaces between stakeholders specific tools as shown in the figure 1.

Central IFC databases support such models (Cruz, Nicolle et al., 2002) (Vanlande, Cruz et al., 2003). They bring the project team a significant help for document management. But services provided by these databases are not fully adapted to usual design methods of a construction project. Therefore many AEC stakeholders are still reluctant to exploit IFC in their own project. Next subsection is dedicated to a study of the typical workflow during a construction project.

![Figure 1. Model of a shared project with IFC (Lebegue, 2004).](image)

1.2 Design Process

Common design processes usually exploit the repeated succession of two important phases (Hanser, 2003):
- Co-design
- Distributed design.
Co-design phase can be a meeting where objectives are defined and work is split. Communication between actors is synchronous here. Afterwards actors can work in parallel and communication between them is asynchronous: fax, e-mails... This is the distributed design phase. Following meetings could define new objectives to restart a new design cycle.

Central IFC databases of an IFC-based shared project only exploits distributed design phase. It makes the process too restrictive, according to many AEC project teams.

That is why our research team suggests a new system. Instead of working around a unique object-oriented model, design team develops the first model (an architectural model for example) and every group works on a local version of the original model. Actors asynchronously communicate these to each other. These product models are freely modifiable without any coherence checking. Then models are merged at the next meeting. This method gives the project team the required flexibility. Figure 2 illustrates that process.

![Figure 2. Suggested workflow system. It is fully adapted to the design process.](image)

1.3 Semantic Comparison

Concerning this suggested system, two issues should be considered:

1. Track the local models evolution in the distributed design phase.
2. Merge local models in the co-design phase.
Merging local models is not a trivial problem especially for coherence checking between architectural models and structural models (Chen, 2005). But if the first problem is brought a good solution, the second point will be far much easier to be addressed. In this paper we focus on the first point.

The goal is to track added, changed, removed, and moved objects between the original version of a construction object and the modified one as made with diff for ASCII files. Unlike diff, the comparison should be semantic here because the serialization of an IFC product model is not unique: Numbering of entities is not unique. Besides the size of IFC files can reach over 100 megabytes so their handling is a heavy task. Therefore semantic comparison should be optimized if the product model has little changed. Lastly design process could be definitely improved through an ergonomic graphical visualization of these differences.

Moreover we focus on light tools which do not need any heavy database to work. Input data are single files coming from standalone software in order to provide the project actors the required flexibility.

The rest of this paper is organized as follows. Section 2 describes previous works on version management and tree comparisons. Section 3 shows our approach for model comparison. Section 4 gives implementation tips. Section 5 suggests future works on this prototype.

2. PREVIOUS WORK

First of all no previous work brings us a direct solution or application to solve the main issue. Actually we have not found any light tool able to compare IFC models or more generally STEP models. STEP is a formalism to standardize data exchanges. IFC is a STEP-compliant data model. Data model is defined through a STEP-11 file written in EXPRESS language (Pierra, 2000). Product models coming from a specific data model are exchanged through STEP-21 files (named SPF files as well).

A research work (Broad, 2003) compares data models: It compares EXPRESS models but it does not compare product models. This research deals with other problems like ontology and taxonomy. A typical application of this work could track evolution of IFC data model.

Therefore we should look for low-level comparisons: ASCII files comparison, trees comparison, and version management.

2.1 The Status-Based Comparison with Diff

Concerning ASCII files, classical diff-like tools allow you to track differences between two textual files. Similar tools are intensively used by
cooperative software development in order to build versions of source code files: Concurrent Versions System.

*Diff* assigns a status to each line of both original and modified file. Four statuses are available: IDENTICAL, CHANGED, REMOVED and ADDED.

However ASCII files are basic structures. Connected semantic data could be described by a graph whose nodes numbering and position in file is arbitrary, according to STEP specifications.

We would like to compare elements from these product models and assign status as *diff* does. Therefore we have to compare two data graphs.

### 2.2 Trees Comparison

Comparing trees is the origin of a lot of famous problems and many papers deal with them. The most known one is “Tree Edit Distance” (Bille, 2003). It compares labelled trees based on simple local operations of deleting, inserting and re-labelling nodes. One common condition is required: nodes must be ordered. Otherwise the problem is in general NP-hard. Usually, an ordered labelled tree can be compared in $O(n^4)$ in time.

Our problem is quite different because the graph (or the tree after a pruning phase) is not a free graph but a constrained one because of its model structure, for example IFC model. Data model defines the skeleton of the product model (STEP-21 file), even if the graph of the product model could contain aggregation nodes: A building contains a list of floors whose size is not fixed by the data model. Nevertheless global structure of a product model is fixed and comparison could be much more effective than a generic tree comparison.

### 2.3 IFC Servers and Databases

Because of their size IFC files are often stored in robust databases and several researchers tried to improve data content (Ting, Yang et al., 2003) (Tanyer, Aouad, 2005), interfaces (SABLE project), and accessibility to data (Vanlande, Cruz et al., 2003), (Cruz, Vanlande et al., 2004).

More generally many research projects were carried out on objects comparison and versions. Complex Entity Versioning (Urtado, 1998) suggests a flexible approach to manage data evolution, compatible with object oriented data models like IFC. Every stored entity creates a version of entities with various dependences. These dependences imply a propagation mechanism of operations described by propagation rules and strategies. However this approach tends to create heavy version tools which cannot be used as a single diff tool between two models. Few applications separate the
comparison mechanism from the version process. We need a light standalone tool in order to obtain a flexible and fast system.

3. APPROACH

As was introduced in the previous section, IFC model comes from a STEP-11 model. This model is described in EXPRESS language. Figure 3 shows an extract from such an EXPRESS file. Product models are exchanged through STEP-21 files.

![Figure 3. Extract from the STEP-11 file of IFC 2.0.](image-url)

Many data models are defined using STEP: IFC, IFC-Bridge, AP203, AP210 (for electronics) and Step-TAS (for satellites) ... Besides a specific data model often evolves: current version of IFC is 2x2 Addentum1. Anyway developing a comparator for a fixed version of IFC is not relevant whereas a more general formalism was created. That is why we have chosen to design and implement a generic comparator of STEP product models. However our analysis has been influenced by IFC structure and specific properties. We validate implemented algorithms with construction projects.

3.1 Analysis of Data Structure

A product model is an oriented graph. Figure 4 shows the transformation from an extract of a STEP-21 file into a graph. A node contains its state and its elementary attributes. An oriented connection creates a link from an entity to another entity. In the figure 4, the graph has two more links (dotted
lines). They illustrate specific attributes: the inverse attributes. They are defined in the STEP-11 file and provide a bidirectional link between entities. But only explicit attributes are written the STEP-21 file. For instance, node 17 is an inverse attribute of node 15.

Figure 4. From a STEP-21 file to the associated data graph.

We may need these bidirectional links to traverse the graph, especially with IFC product models: Comparing two graphs starts with the root node comparison. Root node is not set by STEP specifications but the most relevant is IfcProject. Besides connections between important objects use relationship entities: IfcRelContains. It needs an IfcProject as a container and an IfcSite as a contained object. Figure 5 shows the corresponding diagram.

Figure 5. An IfcProject and an IfcSite connected together by an IfcRelContains.

Therefore a graph traversal from the IfcProject needs inverse links because direct links come from IfcRelContains and not from IfcProject. Let us introduce how we will handle the graph nodes.
3.2 Status

We assign a status to each node of the graph. There are four statuses:

- **IDENTICAL**: Two nodes match and have the same content.
- **CHANGED**: Two nodes match but their content is different.
- **MOVED**: Two nodes match and have the same content but their positions in the graph don’t match.
- **MOVED and CHANGED**: Two nodes match but their content and their position are different. For example, figure 6 only shows that the geometry of the cables has been modified but their position in the abstract graph has changed as well because of the removal of the first cable.

Without these “Moved” status and “Moved and Changed” statuses, we describe only the node state without any consideration about connections in the graph. However, in this model we do not assign status to connections because we only focus on objects. As was mentioned before, graph structure is constrained by model structure so we do not need to track differences between connections. Figure 6 shows a trivial example with a bridge.

Actually the real challenge is to find a way of matching nodes. The aim is to differentiate a node whose elementary attributes have changed between the original model and the modified one, from a node which has been replaced by another node in the modified file. So nodes should be identified. Fortunately specific STEP models like IFC create UniqueIds for many entities: most important entities inherit from IfcRoot entity. It has a globalId attribute. Unfortunately, every entity does not inherit from IfcRoot entity. As our comparator model has to be generic, we cannot use UniqueIds directly. In section 4, we explain how to work around this problem thanks to helpers. Anyway we suppose that several objects are identifiable and others are not. Next section describes how to manage non-identifiable nodes.
3.3 Graph Pruning

As described in first section, an IFC file can be larger than 100 Mb. The associated graph can have more than 1,000,000 nodes with cyclic links. So another challenge is the simplification of the graph. There are two categories of simplification:

- To take off elementary nodes
- To cut redundant connections

Actually elementary nodes are not assigned a status because they are absorbed by the parent entity. Therefore we differentiate the composition of a node from its state: The content of an elementary node is inserted in the state of its parent entity. If there is a change in this node, the parent becomes changed. If a node has several parents, every parent gets a copy of its child node. But what is an elementary node? How can we differentiate a composition from a state? The sharpest way is to assume elementary nodes are non-identifiable nodes. Therefore we solve the previous problem of assigning a status to non-identifiable node. Figure 7 is another extract of an IFC file and figure 8 shows the associated graph with inverse links.
In this graph, there are only four identifiable nodes: 20, 31, 32, 33 and 34, identifications are sequences of characters as first parameter of these nodes (cf. figure 8). Consequently, we obtain the following graph in figure 9.
The element 19 was referenced by 5 parents so this element is copied 5 times in these parents.

Lastly the semantics of a model are not impacted by this pruning since absorbed nodes from a product model are instances of unidentifiable objects, so they never can be compared between them.

### 3.4 Graph Traversal

Previous paragraphs gave some details concerning the traversal, especially the inverse links role. This section suggests an optimized method to traverse this kind of semantic graph.

Our approach is definitely different from classical trees comparisons and traversals because:

- The global structure of the graph is fixed by a data model.
- We focus on nodes traversal, not on links.

A first approach would compare nodes without caring for connections between them. A quadratic-time computation would compare lists of nodes from the original model to lists of nodes from the modified model. We reject this extreme approach because MOVED and MOVED AND CHANGED status cannot be assigned in this case.

This leads us to suggest a true graph traversal: we drive a simultaneous traversal in the original graph and the modified graph. This can be done if and only if root nodes from original and modified graphs are same type. During this process only four statuses are assignable: IDENTICAL, CHANGED, ADDED, and REMOVED. During a node comparison, we compare at first identifications. If there is no matching of IDs, then original entity is assigned a REMOVED status and the modified one is assigned an
ADDED status. Otherwise we compare the state of each node, and we assign IDENTICAL or CHANGED for both original entity and modified entity. Lastly, we compare children entities (like a pre-order tree traversal) and so on. As the structure is a graph and not a tree, we may meet an already visited node. We use tags to ignore already visited nodes.

When the traversal is finished, some ID correspondences between a REMOVED element and an ADDED one may occur. It means that some nodes have changed their position in the graph between the original model and the modified one. Therefore we have to compare in quadratic-time the REMOVED nodes list and the ADDED nodes list. If two nodes have the same ID, they will be assigned a MOVED status or a MOVED AND CHANGED status depending on the state comparison result.

Therefore complexity is quadratic in the worst case (all nodes are assigned ADDED and REMOVED during the traversal), but the graph traversal visits every node only one time. This is much more efficient if a model is hardly modified between the original version and the modified one.

4. IMPLEMENTATION

This section gives some details concerning our implementation of the STEP-21 Semantic Comparator. First we need to know how C++ IFC classes could be generated from the STEP-11 file.

4.1 Early Binding or Late Binding?

STEP models can be exploited in two ways (Loffredo, 2005):

- The package can work directly on any STEP objects: IFC, AP203, Step-TAS... without knowing the model structure before, this is Late Binding. In this case the STEP-11 model is used at runtime as a dictionary to interpret the STEP-21 file.
- The package code has been generated from a program which reads the model structure (STEP-11 file) and then can only work on STEP objects compatible with this model structure. This is Early Binding. In this case the STEP-11 model is hard coded in the package.

Late Binding is very useful when application is intended to work on multiple EXPRESS schemas, but Early Binding is easier and faster to process. So we would rather work on early binding implementation.

Several applications are able to parse an EXPRESS schema to generate C++ classes. We have chosen Expressik from MINT Group of the University
of Manchester. This is a Java package which contains an EXPRESS file parser and a C++ classes Generator (Withers, 2005).

Generated C++ classes contain entities and various get/set methods which give access to attributes. Our comparison model adds a new behaviour to these generated classes. Therefore we use the Visitor design pattern (Gamma, Helm, et al., 1995) whose implementation is prepared during the generation of classes.

4.2 Comparator Generator

Like C++ model classes, we have to choose a way to implement the semantic comparator: Early-binding or Late-binding. As we previously chose Early-binding for C++ model classes, we generate specific comparator classes and hard code the STEP model into these classes.

We used the EXPRESS parser library provided by Expressik package. We have created a Java application which needs three input data: the EXPRESS schema (STEP-11 file), C++ template files and helpers.

Figure 10 illustrates the global mechanism.

![Figure 10. Generating C++ classes and the semantic comparator.](image)

C++ template files are encapsulated into a XML file where specific tags are handled by the comparator generator. C++ template files set a specific syntax compatible with the Express2Cpp Generator. Therefore our comparator generator is reusable for any C++ model classes if specific C++ template files are created.

Lastly helpers are encapsulated into a XML file as well and give some guidelines concerning special properties of specific EXPRESS schema: the class name of the root node, the class name of identifiers, a function to
compare identifiers... So we have to create a helper for every EXPRESS schema. Concerning IFC, we choose IfcProject as root node, and IfcGloballyUniqueId as identification. The comparing identification function is a string comparison. Figure 11 shows a simple helper for IFC. Helpers could be improved to provide subtle behaviours for the comparator.

```xml
<helper>
  <Id name="IfcGloballyUniqueId">
    bool haveSameId(IfcGloballyUniqueId* obj1, IfcGloballyUniqueId* obj2) {
      return *(obj1)==*(obj2);
    }
  </Id>
  <Root name="IfcProject"/>
</helper>
```

Figure 11. A simple helper for IFC data model.

### 4.3 Generated Classes

This section describes the structure of the generated comparator. As was mentioned in the section 4.1, we use the Visitor design pattern. It lets you define a new behaviour without changing the classes of the elements on which it operates (Gamma, Helm et al., 1995).

Actually a comparator simultaneously visits two graphs. Besides, the comparison is divided into four main actions and caching comparison results optimizes the process. Four visitors have been implemented:

- **SemanticComparator**: This class drives the comparison of two entities. If IDs match, entities are visited by StateComparator. Then it calls ChildComparator.
- **StateComparator**: This class compares entities state. As soon as a difference is detected, the method returns a CHANGED tag. If no difference was found, it calls then NidComparator.
- **NidComparator**: This specific visitor prevents the system from copying multiple times non identifiable nodes. It visits these nodes by comparing elementary attributes and complex attributes recursively. It returns a CHANGED tag as soon as a difference is detected.
- **ChildComparator**: This comparator calls Semantic Comparator to visit the children nodes sequentially.

Figure 12 shows a simplified UML diagram of the comparator system. “m_old” is the original graph. There are as many “visit” methods as the number of model entities.
5. RESULTS

The current prototype is running on a PC (Pentium D 820 3.0 Ghz, 2 Gb RAM, Windows XP Pro). We evaluated the performance of two IFC files comparison. We have chosen to measure performance in the worst case: n² comparisons, where n is the number of identifiable nodes, no ID correspondence between nodes has been found. Results are reported in Table 1.

This evaluation simulates a comparison of very different models. Therefore performance should be usually better. These results show that the comparison is a fast operation comparing to file loading. It allows us to improve helpers in order to create a smarter semantic comparator.

<table>
<thead>
<tr>
<th>Table 1. Measure of IFC files comparisons.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Name</td>
</tr>
<tr>
<td>File Size (Mb)</td>
</tr>
<tr>
<td>Entities</td>
</tr>
<tr>
<td>Identifiable nodes</td>
</tr>
<tr>
<td>Loading time</td>
</tr>
<tr>
<td>Comparison time</td>
</tr>
</tbody>
</table>
6. CONCLUSION AND FUTURE WORKS

We have introduced a semantic comparison method and we have implemented it through a standalone tool. After the generation of C++ comparator classes, the system is able to compare two STEP files without any other database. Results are stored in lists sorted by node status.

This application could be exploited by visualization tools in order to focus on changed objects in the model. It could efficiently assist a designer and provide a better communication between actors. Our research team has developed visualization tools of IFC models which can be improved easily through modules. This semantic comparison application is being implemented as a module of our visualization system. Besides we will focus on improving helpers, in order to add sharp mechanisms to our semantic comparator. We are validating the relevance of our system with IFC files generated by commercial software and helpers are the fundamental keys for the achievement of the semantic comparator.

Other tests will validate the semantic comparator with other STEP models, like IFC-Bridge (IFC extension for engineering structure) and Step-TAS (STEP model for satellites).

Lastly this application tends to improve design process. We should take into consideration the other phase of the design process: co-design. Complementary tools and algorithms will be developed to merge local version of IFC files. During this merge cohesion problems could occur so we need a semi-automatic way to check cohesion between construction objects. Once again design process could be improved and AEC actors may be more motivated to exploit IFC in their own project in the near future.

7. REFERENCES


Gamma, E., R. Helm, R. Johnson, J. Vlissides, 1995, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley.


