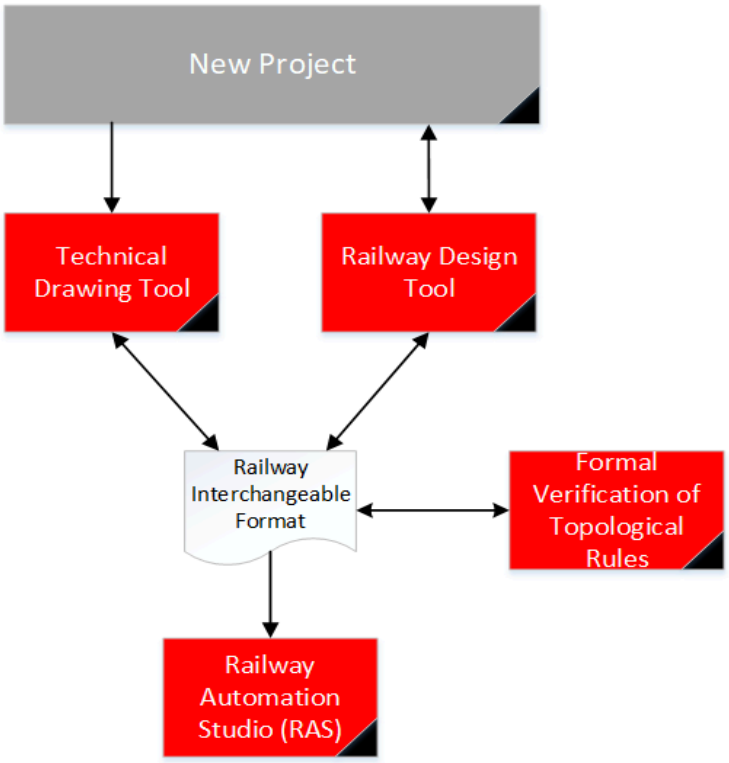
Technical Report

- **Project**: DigiLightRail
- **WP**: WWW
- **Deliverable**: T1.3
- **Producer**: HASLab / INESC TEC
- **Title**: Railway Network Modeling: Tools and Languages
- **Summary**: *This report presents a state-of-the-art review of tools and languages that have been considered in the design of the DigiLightRail approach to railway network modeling (.....)*

Introduction

Project DigiLightRail aims at implementing a tool for designing command and control systems for light surface trains (vulg "*metros*" in the French terminology) allowing for the configuration of *automatic train protection* (ATP) systems and the basic configuration of an entire *cyber-physical system of systems* (CPSoS) control and command system.



The block diagram aside gives an overview of the overall intended design and basic operation principles of one of the key deliverables of DigiLightRail, the EVEREST (Efacec Verification of Railway nEtworkS Tool) design automation tool. In essence, every new project in the railway domain has four main facets, or components. First of all, the complexity of railway networks calls for a *Railway Design Tool* (RDT), that is, an interactive tool helping the railway designer to plan the whole framework independently of the technical drawings needed for the actual civil engineering construction work. These are carried out in another interactive tool, named *technical*

drawing tool (TDT) in the block diagram. Thus two branches of engineering interact in railway design: civil engineers (TDT) and transportation engineers (RDT).

As railways are safety-critical transportation networks whose ill-function might incur into loss of human life and disastrous economic damage, one needs to ensure safety of any transportation operation that the networks allow for. At the very basis of such infra-structures one finds the *network topology* itself. This motivates the third component in the tool-chain, the *Design Verification Tool* (DVT) block of the diagram: a component that allows for the specification of *safety-critical rules* and the verification that the topology abides to such rules by construction. Such *safe-by-construction* quality seal can only be ensured by *correct-by-construction* software design methods, *vulg* formal methods. Indeed, the CENELEC EN 50128 standard[1] explicitly recommends formal verification tools in the design of safety critical software for the railway domain. When the DVT detects any violation to the specified rules, counter-examples depicting the issue will be generated, which must be integrated back into the tool chain in order to be properly interpreted and debugged by the designers.

Once the railway network model is designed and properly validated and verified, it is fed to the subsequent tools where the interlocking systems and signal controllers are specified and verified for safety guarantees. This is represented by the fourth component of the toolchain, the *Railway Automation Studio* (RAS). At this stage railway engineers are handling the dynamic aspects of the railway system, but the network model provides the base over which the controllers are defined. Moreover, the dynamic safety properties that are to be checked by the RAS must be instantiated to each particular network. Thus, information from the railway track is expected to be (unidirectionally) integrated in the RAS.

For these four loosely coupled components of the diagram to interoperate there is a need for a common data-interchange format sufficiently expressive to support all the functionalities of the presented components, RIF (*railway interchangeable format*) in the diagram. Altogether, the four blocks interacting via such a common format assemble into EVEREST.

In summary, the architecture of EVEREST is intentionally *loosely coupled*, meaning that its actual building blocks can vary from installation to installation, including the common interchange format. This will be achieved by design, in the usual way, by providing well-defined APIs specifying which actions need to be provided by each block for the whole software system to interoperate. In this respect, this tool will be quite different from other commercially available tools in the same domain, which offer integrated solutions bound to particular technologies. This *loosely coupled* architecture is convenient in a domain that, albeit moving fast to standardization, is historically diverse in technologies, conventions and rules, in particular country-wise.

[1] EN 50128:2011 “Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems”. CENELEC CLC/TC 9X standard, 2011-06.

The goal of this report is to review existing tools and languages for railway infrastructure modeling that can be considered when putting together railway network modelling components of EVEREST, namely the TDT, the RDT and the underlying RIF format.

Overview of formats and tools for network modelling

As written above, EVEREST is composed of several decoupled components that support the design and validation of railway infrastructures. Central to this architecture is a railway infrastructure model, containing both logical and geometric / geographic information. To this purpose, such a model records information both from railway topological models and from technical engineering drawings. These two perspectives are to be handled by two independent teams and tools, but information is expected to flow in both directions consistently. Besides proposing a selection of such design formats and tools, this report explores relevant state-of-the-art technologies to support the implementation of the following procedures in EVEREST:

1. The introduction, in an existing technical drawing, of visual entities that depict information extracted from the railway topological model;
2. The computation, from the technical drawing, of geometric attributes of the physical railway network infrastructure, such as distances between topological elements;
3. The enrichment of the railway topological model with geometric attributes computed in the previous step.

As a first step towards such goals, this section presents a brief overview of data exchange formats for the interoperability of railway applications, as well as tools for modelling railway network topologies. As for the TDT for technical drawings, EVEREST will adopt *AutoCAD*®, the *de facto* industry standard for engineering projects.

The background of languages and tools for interoperability is the chronic difficulty of connecting different railway IT applications, calling for standardization. Efforts towards such standardization started in Europe around 2002, with the goal of defining the XML based *railway markup language* - railML®[2]. Meanwhile, promoted by the European open-source initiative railML.org, railML® has evolved to a *de facto* standard for data exchange in railway networks.

In 2013, the ERIM (*European Rail Infrastructure Masterplan*) project of the UIC (*International Union of Railways*) launched a group to standardize railway topological models. The project aimed at unifying several data formats for infrastructural models in use at that point. Different formats were being used by the EU - the *Register of Infrastructure* (RINF), the transport layer of the *INSPIRE* Directive for environmental applications - and by national infrastructure managers - *ARIANE* from France's RFF, *InfraNet* from Belgium's Infrabel, *Benedata* from Norway's Jernbaneverket, *RINM* from UK's Network Rail and *PPROD*/*EADB*/*ADB* from Austria's ÖBB. railML, then at version 2, was already being used to exchange railway data but had been

[2] https://www.railml.org/

developed without a clear topological model. The initiative's outcome was the *RailTopoModel*® (RTM) standard (IRS 30100)[3], which proposes a multi-level topological model supporting multiple referencing systems. RTM was developed in close collaboration with the railML.org organization, which resulted in the release of the RTM-compliant railML version 3 in 2017, with a complete re-organization of the infrastructure layer. The French ARIANE format has also evolved to support RTM.

Besides RTM, there are a few other data formats for topological models[4]. *IDM*^{VU} is a standard[5] for infrastructure data management proposed by Germany's VDV. Its models are presented as UML diagrams, although an XML exchange format is also provided. The *OpenStreetMap* (OSM) file formats[6] could also be used to specify railway infrastructures, as in the *OpenRailwayMap* initiative. However, since OSM is developed independently of infrastructure managers and the railway industry, it is not seen as a reliable format but rather as an open data source (indeed, OSM to railML translators have been proposed). It is also worth noting that other formats exist to exchange planned and real-time operational data (including other layers of railML), but these fall outside this project's scope. Nonetheless, some of those formats still allow the specification of rich tracks on which the operational aspects are defined, such as UNISIG *Subset-112*, an exchange format for test scenario specifications that also includes the full specification of the railway infrastructure.

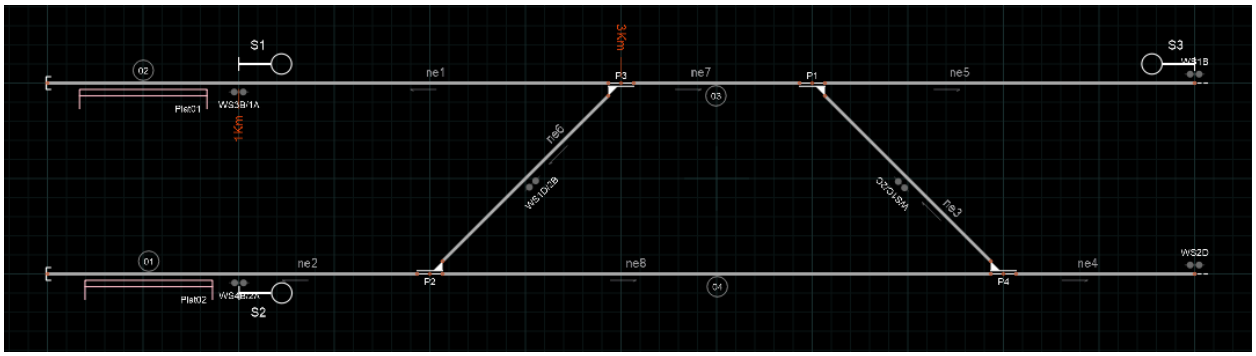
Another key criterion for the selection of the exchange data format is the tool support for the graphical creation of topological models, the candidates for the RDT component of EVEREST. Such tools are largely compliant with the railML format[7] nowadays (although not all with version 3 of the standard), meaning that they can be easily integrated with a railML-centered methodology. *RaIL-AiD* by NEAT[8] allows the user to draw and equip railway track plans, supporting the railML 3 topological layer. *RailComplete* by Railcomplete AS[9] is an AutoCAD plug-in for the design of railway infrastructures integrated into the BIM work process. *Novapoint*^{DCM} by Vianova[10] is a BIM solution for infrastructure projects, which integrates CAD editing through an AutoCAD plug-in. *Vis-All 3d* by Software-Service John[11] integrates CAD data in the 3D modelling of railways. *railOscope* by trafIT[12] is an online viewer for railway data with an integrated track editor. Other tools focus on railway operational simulation but also enable track edition and subsequent exportation into railML. Such is the case of the *ETCS Track Editor* by

[3] http://www.railtopomodel.org/
[4] http://www.capacity4rail.eu/IMG/pdf/c4r_-_d341_-_data_notation_and_modelling_-_public.pdf, 4. Story 1 Consistent Cross Industry Instructure Data
[5] http://www.idmvu.org
[6] https://wiki.openstreetmap.org/wiki/OSM_file_formats
[7] https://www.railml.org/en/introduction/software.html
[8] https://www.rail-aid.com/
[9] https://www.railcomplete.com/en/product/
[10] https://www.novapoint.com/products/novapoint/novapoint-railway
[11] https://john-software.de/produkte/vis-all-3d/
[12] https://railoscope.com/

ERSA[13], *RailSys* by RMCon[14], *OpenTrack* by OpenTrack Railway Technology[15] and *MTS* by Maprex[16]. These often also support track modelling in Subset-112. Lastly, some tools allow only the visualization of railML infrastructure data but not their edition. Such is the case of *RailVivid* by railML.org[17]. It is worth noting that, in many of these tools, track design is *tightly coupled* with other tools of their ecosystem - e.g., by being deployed as an AutoCAD plug-in, or having the editor integrated in a simulation tool - and are thus not well-suited for EVEREST. As already stated, interoperation in EVEREST is expected to be *loosely coupled* meaning that, in particular, the technical drawing in AutoCAD and the railway topological modelling are expected to be carried out independently.

With this state-of-the-art in mind, the high-level architecture of this tool has been designed as shown in the diagram above. In spite of the variety of solutions and of the intended “genericity” of the approach, the railML 3 standard will be adopted as the RIF between the various components of EVEREST, due to its expressive power and the wide consensus that is building up around this international standard in recent years. As already stated, the selection of AutoCAD as the TDT was inescapable. With the RIF selected, the following sections present the railML 3 standard, and survey and explore in more detail existing tools for railway design and editing specifically for railML, in order to support the selection of the RDT. Moreover, as the TDT component is also selected, the succeeding sections present the anatomy of an AutoCAD document, and explore in more detail mechanisms to integrate AutoCAD in EVEREST - both to enhance the technical drawing with information from the railML model, and to extract relevant geometric information from the CAD drawing.

For illustration purposes, throughout the document, we will use a simple example of a railway station whose topological model is depicted below. This station comprises two parallel tracks, four switches to enable trains to be guided from one track to the other, three signals and six train detection devices, which altogether define four train detection sections of interest for interlocking.



[13] https://ersa.clearsy.com/products/etcs-track-editor
[14] https://www.rmcon-int.de/railsys-en/
[15] http://www.opentrack.ch/opentrack/opentrack_e/opentrack_e.html
[16] http://www.maprex.co.kr/
[17] https://www.railml.org/en/user/railvivid.html

railML® in a nutshell

railML (Railway Markup Language) is an open, XML based, data exchange format for data interoperability of railway applications. As of version 3.1[18], railML includes four main subschemas:

- *Infrastructure*, for describing the railway network infrastructure;
- *Interlocking*, for describing railway signalling and interlocking systems;
- *Rollingstock*, for describing rail vehicles and formations;
- *Timetable and Rostering*, for describing railway timetables.

For the purpose of EVEREST, we are mainly interested in the Infrastructure and Interlocking subschemas. The identification and detailed description of the relevant XML elements of these subschemas will be addressed in a companion tech report T1.4. Below, we will briefly describe the main elements of the Infrastructure subschema. This encodes the RTM topological model, which is the only one relevant for the communication between the railway design tools. A more detailed presentation of this subschema can be found in the railML 3.1 Tutorial[19].

The model of a railway infrastructure, described in railML inside a single tag `<infrastructure>`, is divided into two main parts:

- The topology, inside tag `<topology>`, describes the structure of a railway track network. A railML model can describe the topology of several networks inside tag `<networks>`, at different levels of abstraction, ranging from a *microscopic* view that describes how track segments are connected at switches or crossings to a *mesoscopic* or *macroscopic* level, where the focus is on railway lines and their interconnection at railway stations or other operational points. (The DigiLightRail project is mainly concerned with the microscopic level.) In a railML topology model, track segments are defined using tag `<netElement>` and a connection between two track segments is defined by a `<netRelation>`. A snippet of the railML model of our example train station is listed below. As one can see in the depiction above, the track segment identified as `ne1` is connected to two other track segments, identified as `ne6` and `ne7`, at switch `P3`. As such, in the XML snippet, we can see that the `<netElement>` describing this track segment has two children `<relation>` elements that point to the `<netRelation>` elements that will describe the two connections. As an example, the XML snippet also includes one `<netRelation>` element that describes one of the connections, namely the connection between track segments `ne1` and `ne7`, identified inside children `<elementA>` and `<elementB>`, respectively. The `<netRelation>` element also indicates using attributes which endpoints of the two track segments are connected (with `0` denoting one endpoint and `1` the other). Having described all track segments and their connections, a network can be described inside tag `<network>` by just listing all the network resources that constitute it

[18] https://wiki3.railml.org
[19] Christian Rahming: *railML® 3.1 Tutorial - Simple Example Step-by-Step. Part 1: Infrastructure*. 2018.

(a resource being either a `<netElement>` or a `<netRelation>`) and specifying its level of abstraction inside tag `<level>`. The snippet includes a single network described at the microscopic level.

- The functional infrastructure, inside tag `<functionalInfrastructure>`, describes all functional infrastructure elements that belong to a railway network. Among these we have physical elements such as switches, signals, train detection elements, or buffer stops, but also "virtual" elements such as border points to other zones of the network not being modelled, or maximum speed zones. Most elements specify the track segment in which they are located using tag `<spotLocation>` that includes an attribute pointing to the respective `<netElement>`. For illustration purposes, the railML snippet includes a `<signalIS>` element that describes signal `S1`[20], the signal located on track segment `ne1`.

```
<railML version="3.1">
   <infrastructure>
       <topology>
           <netElements>
               <netElement id="ne1">
                   <relation ref="nr_ne1ne7_swi7"/>
                   <relation ref="nr_ne1ne6_swi7"/>
               </netElement>
               ...
           </netElements>
           <netRelations>
               <netRelation id="nr_ne1ne7_swi7" positionOnA="0" positionOnB="0">
                   <elementA ref="ne1"/>
                   <elementB ref="ne7"/>
               </netRelation>
               ...
           </netRelations>
           <networks>
               <network id="nw01">
                   <level id="lv0" descriptionLevel="Micro">
                       <networkResource ref="ne1"/>
                       <networkResource ref="nr_ne1ne7_swi7"/>
                       ...
                   </level>
               </network>
           </networks>
       </topology>
       <functionalInfrastructure>
            <signalsIS>
               <signalIS id="sig36">
                   <name name="S1" language="en"/>
```

[20] Notice that in this case `S1` is not the unique identifier of the `<signalIS>`, defined in the attribute `id`, but a human readable identifier defined in a child element `<name>`.

```
                <spotLocation id="sig36_loc-1" netElementRef="ne1"/>
            </signalIS>
            ...
        </signalsIS>
        ...
    </functionalInfrastructure>
  </infrastructure>
</railML>
```

Some railway design tools, such as RaIL-AiD[21], already export models of the network in the railML format. (The diagram of our train station example in the previous section was produced by this tool.) When sketching a design in such tools, many physical geometric attributes of the layout are omitted, for example, the concrete length of a track segment or the specific position of a signal along a track segment. These attributes can also be described in a railML model and will be the focus of the validation tool to be implemented in this project. For example, the `length` attribute of a `<netElement>` can be used to specify its length in meters, and the `pos` attribute of a `<spotLocation>` can specify the distance in meters to the endpoint `0` of the respective `<netElement>`. To compute these attributes with precision, one needs to resort to a civil engineering project that describes the physical layout of a network, typically developed with CAD tools such as AutoCAD®.

railML® Tools and Libraries

This section presents an overview of some railML tools and libraries[22]. Besides this overview, it also includes a brief description of XML functionalities that can support the processing of railML files.

Editors, validators, and translators

Among all tools supporting railML identified in the context of the project (briefly enumerated above), we highlight the following, which seem to be the most stable and widely used. For variety, this selection includes a validator, some editors, and a converter to another standard format.

- **railVIVID**[23] is a railML® validation and viewer tool that offers full support for railML files, allowing the visualization of infrastructure, timetable, and rolling stock elements. Moreover, it contains an integrated validator that checks the railML file regarding syntax correctness and semantic aspects.
- **RaIL-AiD** (Railway Infrastructure and Layout Aided Designer) is a framework developed by NEAT[24] to simplify the railway signaling system modeling process, including layout

[21] https://www.rail-aid.com
[22] https://www.raiml.org/en/introduction/software.html
[23] https://www.railml.org/en/user/railvivid.html
[24] https://www.neat.it/

definition. Its main features include creating error-free track layouts, the import/export of standard file formats, such as railML, and boosting the setup of the entire track plan.

- **OpenTrack**[25] is a railway planning software used by railway operators, the railway supply industry, consultants and by universities in several countries. Furthermore, it allows the modeling, simulation, and analysis of several railway systems, such as high-speed rails, metro systems and light rails. Moreover, OpenTrack offers interfaces to various data formats, including railML.
- **ETCS Track Editor**[26] is an ERSA tool that enables creating and modifying a track and its storage in a MySQL database. The produced track includes data about the topology, infrastructure, signalling information, switchable balizes and it can either be defined manually or imported through different standardized formats, including railML. Moreover, the produced track can be used with other ERSA tools since they belong to the same ecosystem.
- **RailCOMPLETE**[27] is an AutoCAD® plug-in for the design of railway systems with a high level of detail. It enables owners, designers, contractors, and testers to organize and edit railway data and produce tables, 2D drawings, and 3D visualizations for large multi-discipline railway infrastructure projects, while keeping this data synchronized and updated. Moreover, it offers support during design, ready-for-construction reviews, construction, acceptance testing and as-built documentation. Besides storing drawing details in the DWG format, it can store meta-data such as model names, article codes, relations between different objects, and construction stage information in a standard AutoCAD format. Lastly, it allows users to express rules such as valid locations of objects in a model. These rules can then notify design errors during the modelling process.
- **raiml4RINF**[28] is a tool capable of converting railML source data into the RINF XML format. This tool found its motivation in the fact that the RINF registry became mandatory on the European level.

General purpose XML libraries

Tools such as RaIL-AiD do not support fine-grained modifications of the railML model. However, since railML is an XML format, we can explore the standard and well-established techniques and libraries for querying and modifying XML files. This section provides an overview of two such XML querying mechanisms, XPath and XQuery, and a basic example of XML manipulation that relates to the requirements of EVEREST.

XPath[29]**,** which stands for XML Path Language, is a query language for selecting XML nodes using a path syntax to identify and navigate nodes in an XML document. Besides, it can compute values, such as strings, numbers, or Boolean values from an XML document's content.

[25] http://www.opentrack.ch/opentrack/opentrack_e/opentrack_e.html
[26] https://ersa.clearsy.com/products/etcs-track-editor/
[27] https://www.railcomplete.com/
[28] https://www.railml.org/en/introduction/applications/detail/railml4rinf.html
[29] https://www.w3schools.com/xml/xpath_intro.asp

XQuery[30]**,** standing for XML Query, is a query and functional programming language for finding and extracting elements and attributes from XML documents. XQuery is to XML what SQL is to relational databases.

By analysing the requirements of EVEREST, we have identified two features that must be supported when manipulating railML files: the extraction of information, for example, into another exchangeable format, and the enhancement of an existing railML model with externally calculated data, for example setting the values of the attributes with physical dimensions. In this context, we have specified two simple illustrative tasks that will be used to evaluate the candidate libraries for railML manipulation:

1. The extraction of the set of signals from a railML file into a CSV file, more specifically, extract a CSV file with all `<signalIS>` names and the identifiers of the `<netElement>`s where they are located.
2. Given a CSV file with `<netElement>` identifiers and the respective length, update a railML file adding (or replacing) the `length` attribute of each of the listed `<netElement>`s.

Any XML library should be capable of performing XML updates to a document, such as setting attributes and adding more elements to a document's structure. Thus, there is a vast set of libraries that can complete the tasks mentioned. One of such libraries is Python's `xml.etree.ElementTree` library. We will use this library as an example of the mentioned tasks' completion.
We assume a railML file called `hjallese.railml` containing a railML structure for the first task. We begin by creating a CSV file called `out.csv` and extracting all signal elements from the railML structure using an XPath query denoting the location path of the relevant XML elements. For each signal element, we extract its spot location that contains the reference for its net element. Lastly, we write a new line to the CSV file containing the signal's id and its net element reference.

```
import csv
import xml.etree.ElementTree as ET

# railML namespace
R = '{https://www.railml.org/schemas/3.1}'

# Opens a CSV file called 'out.csv' for writing
# If the file does not exist, then it creates a new one
with open('out.csv', mode='w') as csv_file:
  # Specifies csv file's delimiters
  writer = csv.writer(csv_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
  # Opens and parses railML file
  tree = ET.parse('hjallese.railml')
  # Gets the root element of the railML file
```

[30] https://www.w3schools.com/xml/xquery_intro.asp

```
  root = tree.getroot()
  # Gets all signal elements of the railML structure with XPath query
  path = f".//{R}infrastructure/{R}functionalInfrastructure/{R}signalsIS/{R}signalIS"
  signals = root.findall(path)
  for signal in signals:
    # Gets the signal's spotLocation element that contains the net element's id
    spot_location = signal.find(f".//{R}spotLocation")
    # Writes the signal's id and its net element to the CSV file
    writer.writerow([signal.attrib['id'], spot_location.attrib['netElementRef']])
```

For the second task, we assume the existence of two files: a railML file `hjallese.railml` that contains a railML model and a CSV file called `lengths.csv` that contains the lengths of all net elements of the railML structure. Each line of the CSV file has the format `netElementId, length`. The process starts by opening the CSV file for reading. For each line of the CSV file, we extract the corresponding net element of the railML document and update its length attribute. We can easily query the railML file's net elements by their `id` attributes using a predicate (between square brackets) in a XPath query.

```
import csv
import xml.etree.ElementTree as ET

# railML namespace
R = '{https://www.railml.org/schemas/3.1}'

# Opens file 'lengths.csv' for reading
with open('lengths.csv', 'r') as csv_file:
  # Specifies CSV delimiter
  csv_reader = csv.reader(csv_file, delimiter=',')
  # Opens and parses railML file
  tree = ET.parse('hjallese.railml')
  # Gets the root element of the railML file
  root = tree.getroot()
  for row in csv_reader:
    # The row[0] gives us the id of the current netElement to process
    path = f".//{R}infrastructure/{R}topology/{R}netElements/{R}netElement[@id='{row[0]}']"
    # Extracts the netElement with the id
    net_element = root.find(path)
    # Sets or updates the length attribute of the netElement
    net_element.attrib['length'] = row[1]
  # Saves the changes in the railML file
  tree.write('hjallese.railml')
```

AutoCAD® in a nutshell

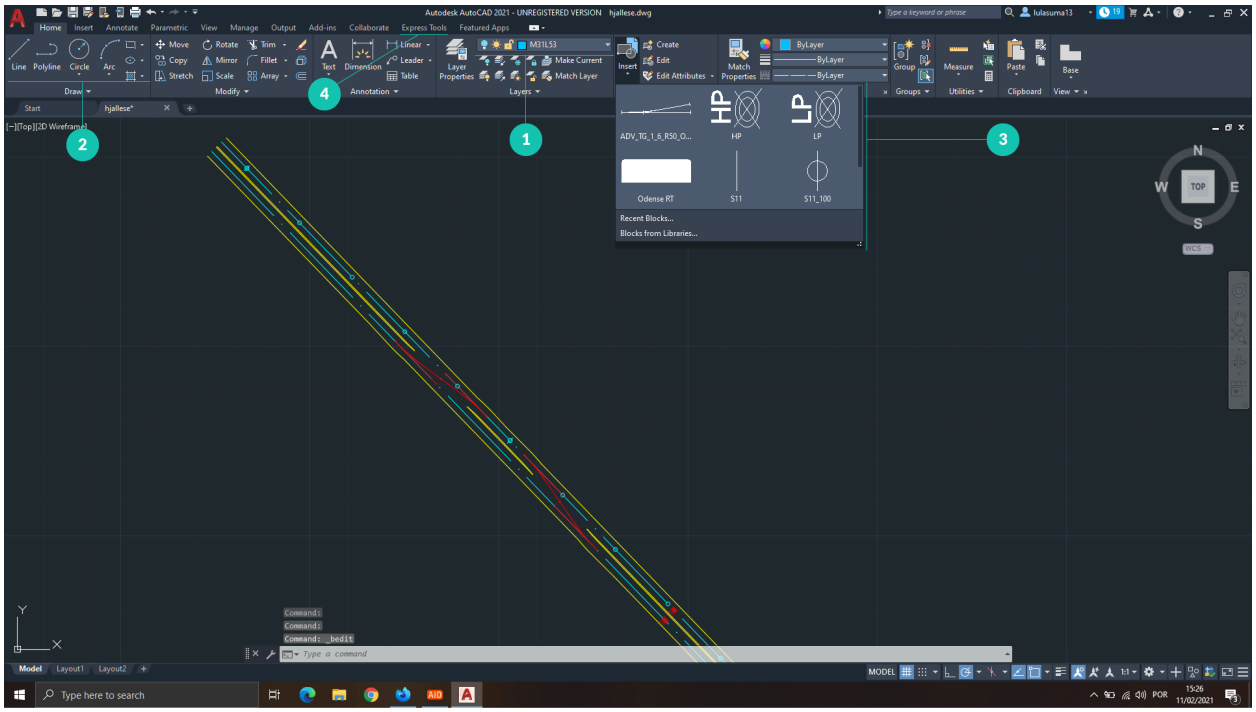
AutoCAD® is computer-aided design (CAD) software that architects, engineers, and construction professionals rely on to create precise 2D and 3D drawings[31]. In this section, we

[31] https://www.autodesk.com/products/autocad/overview

provide an overview of AutoCAD. We begin by describing the main components of an AutoCAD drawing relevant to this project. Then, we present a brief description of this tool's file formats, emphasizing the DWG and DXF formats. Lastly, we provide examples of AutoCAD's tools and libraries.

Anatomy of an AutoCAD® document

An AutoCAD document is a detailed 2D or 3D drawing displaying an engineering or architectural project's geometrical components. The following illustration is based on an AutoCAD document that represents our example railway station. In this section, this drawing will serve as a case study to describe the most relevant components of an AutoCAD document for this project.



1 Layers
2 AutoCAD Entities
3 Blocks
4 Express Tools (include Xdata)

An AutoCAD document contains a broad set of components[32]. However, we emphasize four types of components that we believe are most relevant for this project:

- **Layers:** This component allows users to organize AutoCAD drawings by grouping somehow related objects. It also allows the specification of attributes such as line type or colour for its elements and enables the user to display or hide of all related objects of a layer in a single operation[33]. In the provided example, we can notice several elements

[32] AUTODESK Help. The Hitchhiker's Guide to AutoCAD Basics. 2020.
[33] AUTODESK Help. The Hitchhiker's Guide to AutoCAD Basics. Layers Section, 2020.

with distinct colours. Each group of same-coloured elements belongs to a distinct layer and has different purposes in the drawing. For instance, red elements express rail switches while elements in blue express rail tracks. We expect information from the topological model to also be introduced in a distinct layer.

- **Entities:** An entity is either a simple geometric object, such as a line or an arc, or a group of shapes, such as a block reference that instantiates a block declared in the document's block library. Each entity contains standard information, such as its colour, layer, linestyle, geometry, and several attributes specific to its entity type[34]. In the example, every element presented in the drawing is an entity.
- **Blocks:** This component is a named group of objects that act as a single 2D or 3D object[35]. This feature offers users the possibility to create repeated content, such as drawing symbols, common components, and standard details. Blocks can also be assigned a set of custom attributes that are instantiated for each block reference. As depicted in the example, the "Home" tab's "Insert" option lists all block definitions of a drawing. Some of these blocks are domain-specific; for instance, we can observe rail switches illustrated in red that are instances of the first block presented in the mentioned list. We expect that other elements of the topological model, with the relevant attributes, will need to be added to the block library.
- **Xdata:** This feature allows the attachment of extended object data to any entity[36]. This allows the assignment of additional data to entities other than block references - for which custom attributes are available - such as geometric objects like circles and lines. Arbitrary Xdata properties - dubbed *applications* in this context - can be added to each entity. We expect Xdata to be useful to annotate elements of the topological model that cannot be represented by blocks due to not having a fixed shape, such as the track lines.

AutoCAD® formats

AutoCAD supports several file formats (DWG, DXF, DXB, DWF, DWFx…), not all containing the full information of a drawing. This section focuses on the native AutoCAD format, the proprietary DWG, and the semi-open textual DXF, which are interchangeable and directly converted to one another in AutoCAD. So far, we could not find any incompatibilities between these two file formats.

DWG (from *drawing*) is the closed, proprietary format of AutoCAD and other companion tools, containing all data contained in a CAD drawing. Several versions have been released over the years, the most recent one being DWG 2018. **DXF** (Drawing Interchange Format, or Drawing

https://knowledge.autodesk.com/support/autocad/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/AutoCAD-Core/files/GUID-FA005756-B8F5-4A78-988F-31335A68D77C-htm.html

[34] https://docs.safe.com/fme/2017.0/html/FME_Desktop_Documentation/FME_ReadersWriters/acad/Background.htm

[35] https://www.autodesk.com/solutions/cad-blocks

[36] https://knowledge.autodesk.com/support/autocad/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/AutoCAD-Core/files/GUID-F0299B36-232F-446E-9F81-98F300B36991-htm.html

Exchange Format) is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and other programs[37]. It uses plain text and its specification is publicly available, as opposed to DWG.
The DXF format is a tagged data representation of all the information contained in an AutoCAD drawing. Tagged data means that each data element in the file is preceded by an integer number called a *group code*. A group code's value indicates what type of data element follows. This value also indicates the meaning of a data element for a given object type[38].
The structure of DXF files encompasses the following sections:

- **HEADER:** Contains the drawing's general information, such as AutoCAD drawing database version and angle units.
- **CLASSES:** Holds the information for application-defined classes whose instances appear in the **BLOCKS**, **ENTITIES**, and **OBJECTS** sections of the database.
- **TABLES:** Includes definitions of named items, for instance, XData, block records, and layers.
- **BLOCKS:** This section describes the entities comprising each block definition in the drawing.
- **ENTITIES:** Contains each entity of the drawing, including any block references.
- **OBJECTS:** Holds the data that applies to non-graphical objects used by AutoLISP applications.
- **THUMBNAILIMAGE:** Includes the preview image of the DXF file.

AutoCAD® Tools and Libraries

As stated, in EVEREST information is expected to flow back and forth from the CAD drawings to railML models, with relevant geometric attributes to be calculated in the former, the physical representation of the track. In this section, we present an overall description of some DWG and DXF file formats' libraries to support these tasks, as well as a brief explanation of AutoCAD's scripting language AutoLISP.

DWG Libraries

To manipulate a DWG file programmatically, one can use an API provided by AutoCAD, as long as it is running in the background. AutoDesk also provides a (paid) standalone library, RealDWG[39], to edit DWG files.
Some attempts have been made to reverse engineer the DWG, the most successful by the *Open Design Alliance*. The *Drawings SDK*[40], previously known as *OpenDWG,* provides paid libraries (in C++, .Net or Java) to manipulate DWG files.
Due to the relevance of this format, open initiatives have been pursued regarding DWG. One of those is LibDWG, an open-source C library for manipulating DWG files, which has meanwhile

[37] https://en.wikipedia.org/wiki/AutoCAD_DXF
[38] https://images.autodesk.com/adsk/files/autocad_2012_pdf_dxf-reference_enu.pdf
[39] https://www.techsoft3d.com/products/realdwg/
[40] https://www.opendesign.com/products/drawings

been forked as LibreDWG, and is still actively maintained[41]. Unfortunately, we have attempted to manipulate our example DWG railway tracks using such a library but have failed to do so.

DXF Libraries

Unlike DWG, the DXF specification is open, and many libraries provide the means for reading and creating DXF files and drawing DXF entities. Since, when drawing a DWG file, AutoCAD allows the exportation and importation of DXF files (see next section), components of EVEREST decoupled from AutoCAD can manipulate DXF files while the CAD designers are still allowed to directly handle DWG files. The study covered by this report includes 4 DXF libraries: one written in Python, another written in Java, and two complementary libraries for the Node.js environment. These libraries are the following:

- **JDXF[42]:** This Java library offers support for the generation of DXF files for CAD programs using standard Java AWT Graphics commands. Its supported operations include drawing shapes, like lines and rectangles, text operations and affine transformations, such as scale and rotations. However, there is no built-in support for creating blocks and custom attributes or querying specific entities.
- **Dxf-parser[43] and Dxf-writer[44]:** Both these libraries are written in Javascript and executed in a Node.js environment. They complement each other since the first can read DXF files, while the second can create DXF files. The Dxf-parser can process a DXF file into a Javascript object. Its documentation states that this library supports the parsing of blocks, most CAD 2D entities, layers, and Xdata. However, a functional test could not be achieved with this library without errors, despite following its executing instructions. The Dxf-writer supports the creation of DXF documents and layers and the drawing of most 2D entities. However, it does not support the creation of blocks, custom attributes and Xdata.
- **Ezdxf[45]:** This library provides a Python interface to the DXF format, allowing developers to read and modify existing DXF drawings or create new DXF drawings. Its features include creating DXF entities, blocks and custom attributes, layers and the attachment of Xdata. Furthermore, it provides methods for querying DXF documents' entities by DXF attributes, for instance, name and layer.
- **NetDxf[46]:** netDxf is a .net library programmed in C# to read and write AutoCAD DXF files. It supports AutoCad2000, AutoCad2004, AutoCad2007, AutoCad2010, AutoCad2013, and AutoCad2018 DXF database versions, in both text and binary format. Moreover, it allows the creation of a wide set of DXF entities, blocks, custom attributes, layers and Xdata.

[41] https://www.gnu.org/software/libredwg/
[42] https://jsevy.com/wordpress/index.php/java-and-android/jdxf-java-dxf-library/
[43] https://github.com/gdsestimating/dxf-parser
[44] https://github.com/ognjen-petrovic/js-dxf
[45] https://pypi.org/project/ezdxf/
[46] https://github.com/haplokuon/netDxf

By analyzing the requirements of EVEREST, we have identified two features that should be supported when programming with AutoCAD files: (a) the introduction of new elements in a drawing, for instance, new block references; and (b) the extraction of information from an existing drawing, for instance, the attributes of all references for a particular block. In this context, we have specified two simple illustrative tasks that will be used to evaluate the candidate libraries for AutoCAD files manipulation:

1. Add a block reference with custom attributes to a layer of an existing drawing.
2. Export the attributes of all references to a specific block into a CSV file.

The library used as an example to fulfil these tasks is the Ezdxf library since it is one of the researched libraries that we could successfully use to process AutoCAD's files, supports block creation and offers the least verbose solution.
The instructions required for executing the first step of the evaluation task with this library consist of creating a new layer and a block with one custom attribute. Afterwards, it is possible to add references to the created block to the drawing and attach values to its custom attributes. The following code snippet depicts these steps:

```
import ezdxf

# Creates a new DXF document of version 2018
dxf_doc = ezdxf.new('R2018')
# Gets its model space to add entities
msp = dxf_doc.modelspace()
# Creates a new layer with name 'railml'
dxf_doc.layers.new('railml', dxfattribs={})
# Creates a new block with name 'signal'
signal_block = dxf_doc.blocks.new(name='signal')
# Adds a circle with radius equal to 25 to the block
signal_block.add_circle((0, 0), 25, dxfattribs={})
# Adds a custom attribute to the block with name ID
signal_block.add_attdef('ID', (25, 30), dxfattribs={'height': 15})
# Adds a signal block to the document at position (0, 0) and layer 'railml'
block_ref = msp.add_blockref('signal', (0, 0), dxfattribs={'layer': 'railml'})
# Adds an ID value equal to '1'
block_ref.add_auto_attribs({'ID': '1'})
# Saves the document
dxf_doc.saveas('example.dxf')
```

For the second task of the evaluation, we can easily extract all the references to custom blocks by performing a query by block name. The instructions are the following:

```
import ezdxf
import csv

# Opens file 'example.dxf'
dxf_doc = ezdxf.readfile('example.dxf')
```

```
# Gets its model space to process entities
msp = dxf_doc.modelspace()
# Creates a CSV file with name 'out.csv'
with open('out.csv', mode='w') as csv_file:
  # Specifies csv file's delimiters
  writer = csv.writer(csv_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
  # Searches for 'signal' blocks
  for block_ref in msp.query('INSERT[name=="signal"]'):
    # Prints the value of the block 'ID' attribute to the CSV file
    writer.writerow([block_ref.get_attrib('ID').dxf.text])
```

AutoLISP

AutoLISP is a dialect of the programming language Lisp explicitly built for use with AutoCAD and its derivatives[47]. Its primary purpose is to provide the means for users to create programs that will automatically generate drawings. User-defined customized commands can be added to AutoCAD using AutoLISP. These customized commands can then perform a set of automated activities[48].

Aside from the core Lisp language, most of the primitive functions are for geometry, accessing AutoCAD's internal DWG/DXF database, or manipulating graphical entities in AutoCAD. AutoLISP processes these entities' properties as association lists in which values are paired with AutoCAD group codes that indicate properties such as definitional points, colours, and layers. A user can interact with AutoLISP code through AutoCAD's graphical editor using primitive functions that allow the user to pick points, choose objects on the screen, input numbers and other data. It also provides commands to convert DWG and DXF files to one another. Lastly, users can import the AutoLISP code as `.lsp` files.

Given the requirements of EVEREST, we consider that the support of AutoCAD scripting functionalities is essential for supporting the following features: import elements of another drawing to the current document, for example depicting railML information as block attributes; calculate distances between objects of a drawing, and export specific block attributes to another file format. Consequently, we have specified three simple tasks to study the most relevant commands of AutoLISP:

1. Merge another document with an open DWG/DXF document.
2. Calculate a block's position along a polyline and store its value in an attribute.
3. Export some block attributes to a CSV file.

The AutoLISP dialect includes a command that allows the importing of an entire drawing to another document, therefore fulfilling the first evaluation task. Moreover, it contains a list of commands capable of calculating distances or getting points of a curve, which can be helpful for the second task. Lastly, the dialect also provides commands for selecting objects that match the criteria of a filter list, which are useful to perform both the second and the third task of the evaluation. The following table presents a description of the mentioned commands:

[47] https://en.wikipedia.org/wiki/AutoLISP
[48] http://www.caddsoftsolutions.com/AutoLISP.htm

(vla-Import dest source point scale)	Imports all elements of drawing `source` to drawing `dest`. The elements are inserted at the coordinates specified by `point` with a scale factor equal to `scale`[49].
(vlax-curve-getdistatpoint curve point)	Returns the length of the curve's segment between a curve's start point and a specified point[50]. The `curve` object is a `vla-object`[51], representing any drawing object depicted as a Visual LISP ActiveX® object.
(vlax-curve-getclosestpointto curve point [extend])	Returns the point on a curve that is nearest to a specified point. If the `extend` parameter is specified, the command extends the curve when searching for the nearest point[52].
(ssget [method] [point1] [point2] [point-list] [filter-list])	Creates a selection set from the selected object. Useful to select blocks by name, for instance[53]. `(ssget "_X" '((0 . "INSERT") (2 . "signal"))` The method (X) searches the entire drawing database for elements that match the filter list. The second argument of the example command is the filter list. This list has two criteria: search for `INSERT` entities (code 0) representing block references, with a name (code 2) equal to `"signal"`.

To perform the first evaluation task, we require a file pointer of the current document and the drawing's name to import. In this example, the imported drawing is inserted at position (0, 0, 0) with a scale factor of 1. The second task requires the selection of all target blocks. In this case, each block could, for example, represent a signal for which we intend to determine the distance to the beginning of a polyline representing a track. We can then calculate, for each block, the closest point to the polyline. With this point, we can calculate its position along the polyline with the command `vlax-curve-getdistatpoint`. Lastly, we can store the distance value as a string in an attribute "`position`" of the signal block. For the sake of simplicity, we merged both

[49] https://www.afralisp.net/archive/methods/lista/import_method.htm

[50] http://docs.autodesk.com/ACD/2013/ENU/index.html?url=files/GUID-90F585FE-D5C8-4C08-AFC7-A09A697EA52A.htm,topicNumber=d30e635363

[51] http://docs.autodesk.com/ACD/2013/ENU/index.html?url=files/GUID-90F585FE-D5C8-4C08-AFC7-A09A697EA52A.htm,topicNumber=d30e635363

[52] http://docs.autodesk.com/ACD/2013/ENU/index.html?url=files/GUID-90F585FE-D5C8-4C08-AFC7-A09A697EA52A.htm,topicNumber=d30e635363

[53] http://docs.autodesk.com/ACD/2011/ENU/filesALR/WS1a9193826455f5ff1a32d8d10ebc6b7ccc-693e.htm

the first and second steps of the evaluation task into one code snippet. Moreover, we have the following assumptions:

- The `dw-fp` variable stores the current document file pointer.
- The `source` variable holds the name of the drawing to import as a string.
- The `line-obj` holds the reference to the polyline.
- The command `LM:setattributevalue`[54] sets the value of a block's attribute. This is a utility command that was included in the script.

```
; dw-fp - File pointer to current drawing
; source - Name of the file to import
; Imports all elements of file source to dw-fp at coordinates (x,y,z)
; (0, 0, 0) and scale factor equal to 1
(vla-Import dw-fp source (vlax-3d-point '(0 0 0)) 1)
; Selects all signal blocks
(setq signals (ssget "_X" '((0 . "INSERT") (2 . "signal"))))
; For every signal block update distance to line
(repeat (setq i (sslength signals))
  ; Current block of the loop
  (setq block-ref (ssname signals (setq i (1- i))))
  ; Position (x,y,z) of the block
  (setq block-xyz (cdr (assoc 10 (entget block-ref))))
  ; Gets the closest point of the polyline line-obj to the point block-xyz
  (setq line-p (vlax-curve-getclosestpointto line-obj block-xyz))
  ; Gets the distance across the polyline line-ref of the point line-p
  (setq dist (vlax-curve-getdistatpoint line-ref line-p))
  ; Sets the attribute position of the current block to the distance across the polyline
  (LM:setattributevalue block-ref "position" (rtos dist))
```

For the third task we need to store all "`position`" attributes of signal blocks in a CSV file. We have the following assumptions for simplicity:

- The `csv-filename` contains the name of the CSV file to store the values.
- The command `LM:getattributevalue`[55] returns the value of a block's attribute. This is a utility command that was included in the script.

```
; Creates csv file
(setq fp (open csv-filename "w"))
; Selects all signal blocks
(setq signals (ssget "_X" '((0 . "INSERT") (2 . "signal"))))
; Loops through all signal blocks
(repeat (setq i (sslength signals))
  ; Current block of the loop
  (setq block-ref (ssname signals (setq i (1- i))))
  ; Gets the attribute position of the current block
```

[54] http://www.lee-mac.com/attributefunctions.html
[55] http://www.lee-mac.com/attributefunctions.html

```
  ; Writes the attribute's value to the output file
  (write-line (LM:getattributevalue block-ref "position") fp)
; Closes file pointer
(close fp)
```

Deliverable description

Consolidação do estado da arte das ferramentas e linguagens de modelação de redes ferroviárias [entregável: 1 relatório] A ferramenta a desenvolver tem como base diferentes linguagens standard. Para os desenhos técnicos, gerados por ferramentas de CAD, o formato a considerar será o DWG, e para a representação de informação relativa a sistemas e serviços de caminhos de ferro, será considerado o formato XML standard railML3.0. Esta tarefa tem por objetivo estudar estas linguagens e identificar possíveis limitações no contexto da ferramenta a desenvolver, assim como técnicas e ferramentas associadas que possam servir de apoio no desenvolvimento do projeto.

AutoCAD formats

Interface, blocks, etc.

Acho que é relativamente simples fazer uma script em autolisp/ActiveX que calcula o ponto mais próximo numa spline, mede a distância deste à origem da spline e faz o set desse valor num atributo de um bloco. Infelizmente acho que essas funções só funcionam em windows. Mas dados blocos (de sinais, por exemplo) em que um atributo (definido manualmente) é o nome da polyline correspondente à track, acho que o cálculo da respectiva posição na track (um dos atributos necessários no railML) é possível.

https://help.autodesk.com/view/OARX/2020/ENU/?guid=GUID-90F585FE-D5C8-4C08-AFC7-A09A697EA52A
https://help.autodesk.com/view/OARX/2020/ENU/?guid=GUID-627E64B0-B34C-4E7A-AC69-745EDAC1080D

Também não é difícil criar um dynamic block que faz snap tangencialmente a uma polyline, e que depois é estendido na perpendicular para definir o ponto onde está fisicamente o sinal, mas para preencher a distância continua a ser necessário o método acima, acho eu.

Parsers and other relevant tools and libraries

DWG:

https://www.opendesign.com/files/guestdownloads/OpenDesign_Specification_for_.dwg_files.pdf.

Há uma versão aberta do DWG que parece relativamente ativa (houve uma release este mês):

https://www.gnu.org/software/libredwg/
https://github.com/LibreDWG/libredwg

Alcino: experimentei esta biblioteca, mas dá bastantes erros a traduzir o ficheiro exemplo de dwg para dxf. Tentei editar o dxf e converter de volta, mas não conseguiu traduzir. Acho que não vai servir. A AutoDesk tem uma biblioteca para editar DWGs (RealDWG) em apps standalone, mas é muito cara. Também é possível utilizar gratuitamente a API do Autocad mas para correr os programas é necessário ter o Autocad a correr. A ODA também tem bibliotecas para isso mais baratas (para uso académico ou research são gratuitas) - https://www.opendesign.com/products/drawings (C++, .Net, Java). Também têm um conversor gratuito de DWG para DXF - https://www.opendesign.com/guestfiles/oda_file_converter - experimentei e consegui converter o DWG para DXF, editar qq coisa no DXF (o texto associado a um sinal) e converter de volta. No online viewer da AutoDesk parece tudo bem, mas não sei se se perde alguma informação...

DWFx:

Além do DWG há mais 2 formatos para os quais AutoCAD exporta, DWF e DWFx, este último é XPS (zips de XML)

- Que tipo de meta-dados são preservados?
- Aparentemente o processo é unidirecional, não podem ser importados de volta para o AutoCAD (podem ser adicionados como “underlays”)

DXF:

- Formato textual
- Referência do DXF: https://images.autodesk.com/adsk/files/autocad_2012_pdf_dxf-reference_enu.pdf
- Aparentemente tem a conceito de “Class”, são definidas por cada aplicação
- Também suporta os “extension dictionaries” usados em (Luteberget and Johansen, 2018) para introduzir railML como meta-data
- Este parece que dá para importar, deve ter os dados todos

DXF viewing:
https://chrome.google.com/webstore/detail/dxf-viewer/hbfpaeoimiicejdjhmnlhkkncllliibbm etc

Nuno: apesar de inicialmente ter sido proposto para conter a mesma info que os DWG há funcionalidades mais avançadas que já não são suportadas, mas acho que não são relevantes para nós (tipo dynamic blocks que tem a ver com associar ações de transformação geométrica a blocos).

descobri uma coisa gira que talvez nos vá safar de manipular os ficheiros completos. Os blocos podem têm **atributos**, basicamente propriedades adicionais (já confirmei e vão no DXF); mas há uma vantagem: estas propriedades podem ser exportadas para CSVs com o id do bloco.

adicionalmente os blocos podem ter também definidos **parâmetros**, que são propriedades dinâmicas calculadas a partir do diagrama (distância entre pontos, ou ângulos, etc) (já confirmei e vão no DXF); o interessante é que os atributos podem ser preenchidos dinamicamente com o valor dos parâmetros.

basicamente, com blocos com um conjunto certo de parâmetros e atributos, talvez consigamos automaticamente extrair toda a informação sem sequer processar os DWGs/DXFs completos, porque podemos exportar para CSV os atributos relevantes.

ainda por cima dá uma solução lightweight para bidirecionalidade: os atributos dos blocos podem ser importados de volta em CSV, actualizando os blocos.

Problema (?): algumas destas funcionalidades não dão para mac

- não tem export para CSV (mas tem um export para templates textuais que dão no mesmo)
- mais grave, não tem apontadores para o tipo de bloco, apenas para blocos individuais, o que implica apontar para cada parametro quando cada bloco é criado

Nuno: testei algumas bibliotecas open-source para Java e consegui fazer parsing:

- Mais antigo, consegui fazer parsing/printing e passar para svg, não tenho a certeza se permite transformações http://kabeja.sourceforge.net/
- Permite importar e manipular mas não descobri o parser https://jsevy.com/wordpress/index.php/java-and-android/jdxf-java-dxf-library/

RailML overview

Parsers and other relevant tools and libraries

railML.org: https://www.railml.org/.

Related work

Luteberget and Johansen, 2018

Como seria de esperar assumem que o CAD é concebido usando blocks para os sinais etc, mas penso que continuam a usar polylines para as ferrovias (será que o DWG permite definir novos tipos de polylines? Pelos vistos sim - ver https://forums.autodesk.com/t5/objectarx/custom-polyline-entity-with-an-additional-property/m-p/8275389, acho que usando o tal ObjectARX - AutoCAD runtime extension - dá para fazer tudo, mas não gostei do comentário “If you study ObjectARX every day, it will take from six months to a year until you can create your first Custom Entity. And it is very optimistic.”). Da Fig 2 infiro que a biblioteca deles de blocks (e polylines?) já tem o respectivo encoding em railML nos tais extension dictionaries, o que facilitará a posterior tradução para railML.

Também guardam outra informação relevante (tipo informação de interlocking) no global extension dictionary do DWG, ou seja, de facto o DWG já tem todo o modelo railML lá dentro e é o único ficheiro que é editado (eles chamam-lhe semantic CAD), sendo o modelo railML puro apenas uma view deste. Isto basicamente permite evitar toda a necessidade de bidireccionalidade, pois acho que esta view em railML puro nunca é editada no toolset deles.

Ou seja, eu acho que eles “obrigam” os engenheiros a usar um AutoCAD muito kitado por eles, e não aceitam DWGs desenvolvidos de outra forma. **Temos que discutir isto com a EFACEC.**.. Isto fica claro na secção 5, pois a própria verificação está integrada no AutoCAD. Concluindo, eu acho que não há “magia” nenhuma na conversão deles de DWG para railML - tudo que é necessário já está nos blocks / polylines e penso que o utilizador é responsável por editar informação que falte (por exemplo, desconfio que no block correspondente a um switch o utilizador tenha que dizer manualmente a que tracks - polylines - ele está ligado...).

https://www.mn.uio.no/ifi/english/research/projects/railcons/index.html

Se virem o vídeo fica claro que é suposto usar-se o plugin deles para o AutoCAD desde o início. **Se for para fazer uma coisa destas estamos tramados... (ver comentário acima sobre o ObjectARX).** Também tem outros papers que podem ser interessantes.

https://www.railcomplete.com/en/welcome-read-more/

Aqui referem algum suporte para fazer upgrade de DWGs legacy, mas não percebo muito bem quão automático é. Não encontro detalhes.

Standards (BIM) etc

IEC 81346 - referencing

AutoCAD norms (Australia):
https://www.dpti.sa.gov.au/__data/assets/pdf_file/0004/327298/Drafting_Standard_for_AutoCAD_drawings_AM4-DOC-000364.pdf

BIM na ferrovia, cf: http://www.ptbim.org/img/LivroAtas_ptBIM2018.pdf

BIM in Portugal (IST): http://www.ct197.pt/

Tools

Aparentemente há viewers DWG on line, cf *Can I open a DWG file without AutoCAD? DWG is a native AutoCAD file format but it can be opened with other software's and online tools as well. The easiest and most recommended method is using Autodesk Trueview, a free DWG viewer from Autodesk.* (https://viewer.autodesk.com/)

-