Change toolkit for digital building permit

Deliverable number	D2.4
Deliverable name	CHEK data validity-supporting tools
Work package number	WP2 Information requirements for the DBP use case
Deliverable leader	Delft University of Technology
Dissemination Level	Public

Status	Final
Version Number	V1.0
Due date	30/11/2024
Submission date	09/12/2024

Project no. 101058559
Start date of project: 1 October 2022
Duration: 36 months
File name: CHEK_101058559_D2.4_CHEK data validity-supporting tools_v1.0-Final

This project has received funding from the European Union under the Horizon Europe Research & Innovation Programme 2021-2027 (grant agreement no. 101058559).

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Funded by the European Union

Authors and contributors

Author	Organisation	E-mail
Jasper van der Vaart	TUD	j.a.j.vandervaart@tudelft.nl
Peter Bonsma	RDF	peter.bonsma@rdf.bg
Luiggi Alfaro	DIR	luiggi.alfaro@diroots.com
Alejandro Villar	OGC	avillar@ogc.org

Quality control

Author	Organisation	Role	Date
Claus Nagel	VCS	Reviewer	05/11/2024
Léon van Berlo	BSI	Reviewer	07/11/2024
Elisa Dutsch	VCS	Reviewer	07/11/2024
Silvia Mastrolembo Ventura	UBS	WP leader	26/11/2024
Jantien Stoter	TUD	Coordinator	09/12/2024

Document history

Release	Description	Date	Author
V0.1	First Draft	2/11/2024	Peter Bonsma, Luiggi Alfaro & Alenjadro Villar
V0.2	Second Draft	22/11/2024	Jasper van der Vaart, Peter Bonsma, Luiggi Alfaro & Alenjadro Villar
V1.0	Final	9/12/2024	Jasper van der Vaart, Peter Bonsma, Luiggi Alfaro & Alenjadro Villar

Contents

1. Executive Summary

The main aim of WP2 is to improve data and service interoperability by (1) the interpretation of building regulations to define information requirements for the digital building permit use case and associated dataset (D2.1) and (2) ensuring data interoperability within the defined GeoBIM environment through open Building Information Modelling (BIM) and 3D geoinformation standards (D2.2; D2.3). The final step of WP2, T2.5/D2.4 “CHEK data validity-supporting tools” is not a further refinement of the rules, regulations or files but the support of the validation and refinement of this data in order to guarantee the quality of BIM and 3D city models as well as their compliance to defined data schema. To fulfil this role three software applications have been developed: an IFC validator, IFC exporter and a CityGML/CityJSON validator.

- The IFC validation focuses on verifying compliance with the CHEK IFC schema (D2.2). It is split into four tools: EXPRESS schema validation, Information Delivery Specification (IDS) validation, Property Set Definition (PSD) validation and micro-services.
- The IFC Exporter enhances the integration of IDS requirements for the IFC exports within selected BIM authoring software: Autodesk Revit (version 2022 to 2025) and Graphisoft ArchiCAD (v27). At the core the exporter populates the data based on IDS requirements. It streamlines the IFC export workflow by allowing users to configure IDS settings and efficiently map necessary IFC data.
- The CityGML/CityJSON validator ensures the presence and proper characteristics of city objects in a set of CityGML and/or CityJSON files, as well as the correctness of the geometry primitives employed in them. The semantic data is validated in Resource Description Framework (RDF) while the geometry is validated with the help of val3dity which was earlier developed outside of the CHEK project.

2. Introduction

The main aim of WP2 is to improve data and service interoperability by (1) the interpretation of building regulations to define information requirements for the digital building permit use case and associated dataset (D2.1) and (2) ensuring data interoperability within the defined GeoBIM environment through open Building Information Modelling (BIM) and 3D geoinformation standards (D2.2; D2.3). These are the Industry Foundation Classes (IFC) and CityGML/CityJSON file formats respectively. The files with these additional and refined information requirements are named CHEK IFC and CHEK CityGML/CityJSON respectively. The final step of WP2, T2.5/D2.4 "CHEK data validity-supporting tools" is not a further refinement of the rules, regulations or files but the support of the validation and refinement of this data in order to guarantee the quality of BIM and 3D city models as well as their compliance to defined data schema.

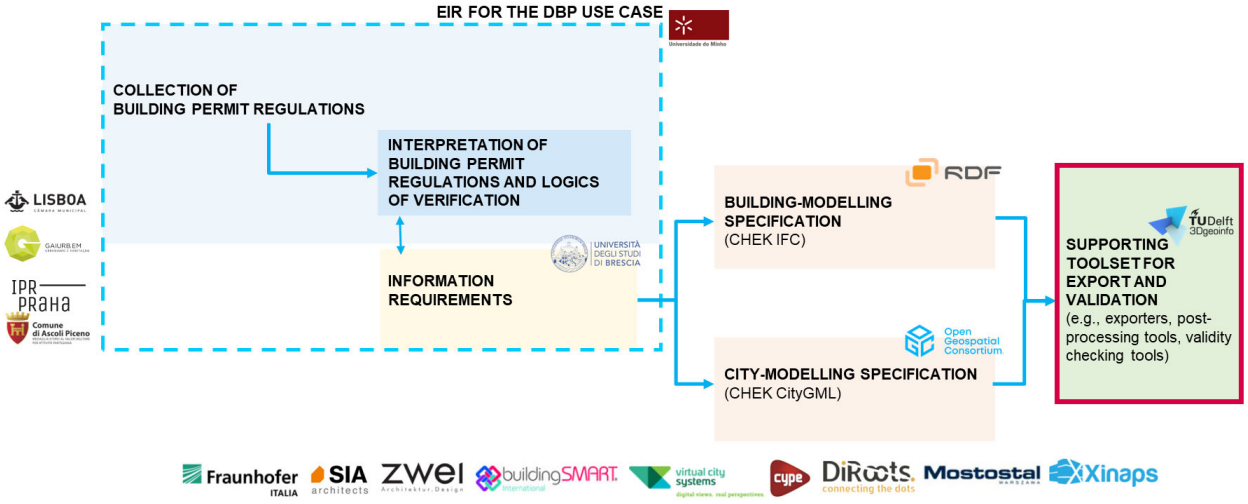

Figure 1 WP2 workflow with T2.5/D2.4 highlighted

It is crucial to validate both the geometrical and non-geometrical attributes embedded in the CHEK models to guarantee correct functioning of downstream applications. These applications require certain data to function and this will not be able to succeed if data are missing or invalid. Validating the data before further processing will increase the reliability of these downstream applications and their output. These applications are often able to validate the data themselves. However, centralising the validation process to a set of external tools has a couple of advantages:

All parties can validate the data they send and receive with the BIM and 3D city models validation tools. Issues can thus be found and addressed very closely from their origin, this will reduce the chance of certain errors propagating through the DBP process, because errors can be corrected early in the process.

A central validation tool will reduce the chances of different interpretations being utilized for the same rules. When every downstream application utilizes its own validation process that has been developed by different individuals or organisations a rule can be interpreted differently across these applications. This could result in a situation where a value or attribute can be validated as compliant according to one application while the same value or attribute can be validated as invalid for another.

A central validation tool will only require a single update when regulations are changed. When every downstream application utilizes its own validation process not all will be updated at the identical time but across a larger time window. This could result in a situation where during an updating period the same files are being handled differently.

This deliverable will address the creation and documentation of the exporting and post-processing toolkit. Two validators have been developed for checking the two defined data schemas and their contents: the CHEK IFC and the CHEK CityGML/CityJSON schema. Moreover, two IFC exporters have been developed for Autodesk Revit and Graphisoft ArchiCAD. Contents of this deliverable can be split into two different validation topics or scales, see Figure 1: The validation of the building modelling specification defined in D2.1 and D2.2, i.e. CHEK IFC

Chapter 3 will cover the IFC validator developed by RDF. This section will cover Express, IDS, and PSD validation expanded with micro-services.

Chapter 4 4 will cover the IFC exporters for ArchiCAD and Revit developed by DiRoots
The validation of the city modelling specification that was defined in D2.1 and D2.3, i.e. CHEK CityGML/CityJSON

Chapter 5 will cover the CityGML/CityJSON validator developed by the OGC

3. IFC Validator

3.1 General description

The IFC Validation is focused on the validation of the CHEK IFC schema (D2.2). The purpose of this set of validation tools is to make sure an IFC file is a valid CHEK IFC file. If a design is exported as a perfect CHEK IFC file, it will be possible for downstream applications to find the information required to execute the automated required automated processing. Therefore, it is important to understand the quality of the given input in order to value the quality of the output of downstream applications. CHEK IFC not only requires an IFC file according to a certain schema (IFC4 ADD2 TC1) and a certain MVD but also introduces very specific requirements concerning the content within the design. This means even a format compliant IFC file is not always a valid CHEK IFC file. All CHEK validation tools are developed in a way that allows their use in generic cases as well. This allows these tools, developed for CHEK, to be reused in many use cases outside the CHEK project. However, it also allows CHEK itself to adjust CHEK IFC definition and adapt to more recent versions of the IFC standard for example without losing the validation capabilities. CHEK IFC Validation consists of four individual groups of tooling:

EXPRESS (ISO 10303-11) Schema Validation
Each IFC schema (before IFC5) is defined in a computer interpretable language, i.e. EXPRESS. Although IFC itself as an ISO standard also includes more detail and written explanations in the documentation the subset defined in EXPRESS can be used as validation for the actual IFC files exported according to that schema. An IFC independent validation tool is created based on the EXPRESS language allowing validation of IFC files against the schema as delivered by buildingSMART International.

Information Delivery Specification (IDS)1.0 Validation
The IDS file defines the specific requirements regarding information that should be available in the IFC file. This includes the cardinalities of certain objects but also the structure of stored information. An IDS file defines extra rules on top of the IFC specification itself. The IDS validation tooling allows validation of an IFC file concerning the extra requirements defined in the IDS file.
Property Set Definition (PSD) Validation
Next to the EXPRESS schema and documentation the IFC ISO specification also defines Property Set and Property definitions. These definitions can be seen as a ‘soft’ extension of the EXPRESS file. This is not included in the EXPRESS file to prevent the file from becoming too large and cumbersome. The open standard used to define these Property Sets and Properties is Property Set Definition (PSD); This standard can also be used to define Property Sets and Properties that fall outside the ISO standard of IFC. This validation tool checks if PSD is followed correctly and can be seen as an extension of the EXPRESS Schema Validation tooling.

Micro-Services Validation Use
Micro-services can be used to extend, adjust and fix potential inconsistencies in IFC files. However, in a similar manner mirco-services can also be used to validate IFC files. Instead of adjusting IFC files a micro-service can check if already defined values in the IFC file are ‘in line’ with values that the application computed. For example, instead of adding a BuildingHeight property to an IFC file, the calculated building height by a micro service can also be used to check if the stored value for this property is available and the same (or the same within boundaries) as the calculated value.

All mentioned validations tools are developed as C++11 (ISO/IEC 14882) source code. The tools are all made available through a public GitHub page except for the EXPRESS validation. It is expected third party tooling will use all these sources as part of their own development.

To ease this process of integrating the delivered source code an API has been created for each of the first three validation sources (EXPRESS, IDS and PSD). These API's are also integrated in one example IFC Viewer application that is delivered together with source code access. This IFC Viewer will be used in this document to show the results of these developed validations.

When comparing the mentioned validators to the definition of CHEK IFC as defined in D2.2it can be noted that one validation tool is missing. This is the validation tool focusing on validation of the selected MVD (Model View Definition), i.e. mvdXML validation. Development of this validator was initially planned but since mvdXML is deprecated and being phased out it was decided to not implement it and trust certified software companies to export such data correctly. BuildingSMART International, as partner of the CHEK project, advised using the newer IDS standard for exchange requirements definitions.

3.2 EXPRESS Schema Validation

EXPRESS as an ISO standard (ISO 10303-11) is a language that allows the definition of an EXPRESS schema. This is in many ways like XSD (XML Schema Definition) if standard IFC SPFF file (STEP Physical File Format) would be compared to an XML (Extensible Markup Language) file. However EXPRESS does not only define a schema data model itself, but it also harbours a very simple version of a programming language. This programming language part of EXPRESS is used to apply validation on content and in case of IFC schemas is quite elaborated. The EXPRESS Schema Validator allows validation of the data model as well as validation of the WHERE rules and FUNCTIONS written according to the ISO specification. This was a time intensive task as it includes support for the embedded 'programming language' being part of the EXPRESS definition, and an important part was already available in the IFC Engine library as a late binding solution, see Figure 5.

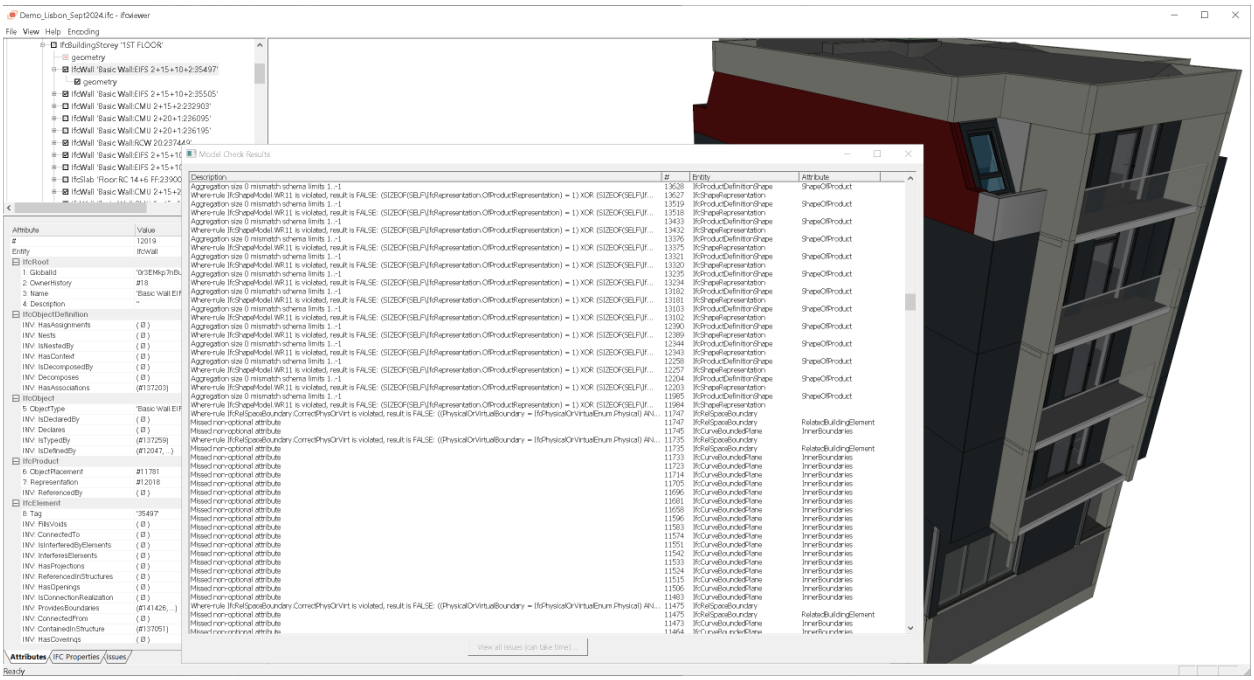

Figure 2 IFC Viewer screenshot Lisbon design in Autodesk Revit with 1352 identified issues.

In Table 1, Figure 2, , Figure 3 and Figure 4 the results of the EXPRESS validation on CHEK IFC models can be found. When analyzing these results, it was noted that all commercial BIM authoring platforms that were used to generate the CHEK project's IFC files are making mistakes (e.g., over 100 different issues in the IFC files developed within the CHEK project). These issues range from missing arguments to incorrect geometry. So even although the software is certified by bCert the generated IFC file has more than a few issues. These issues are mainly due to mistakes made by the BIM authoring software and very hard if not impossible to prevent from happening by the software's users. These findings have been shared with buildingSMART International and the software vendors; it is expected that such issues will be solved by the vendors once CAD systems will be certified for the most recent official IFC version as released earlier this year, IFC 4.3 ADD2 (ISO 16739-2024). Because adoption of new IFC versions has been a slow process in practice and partners within CHEK do not support this standard yet, CHEK IFC is not based on this new official standard.

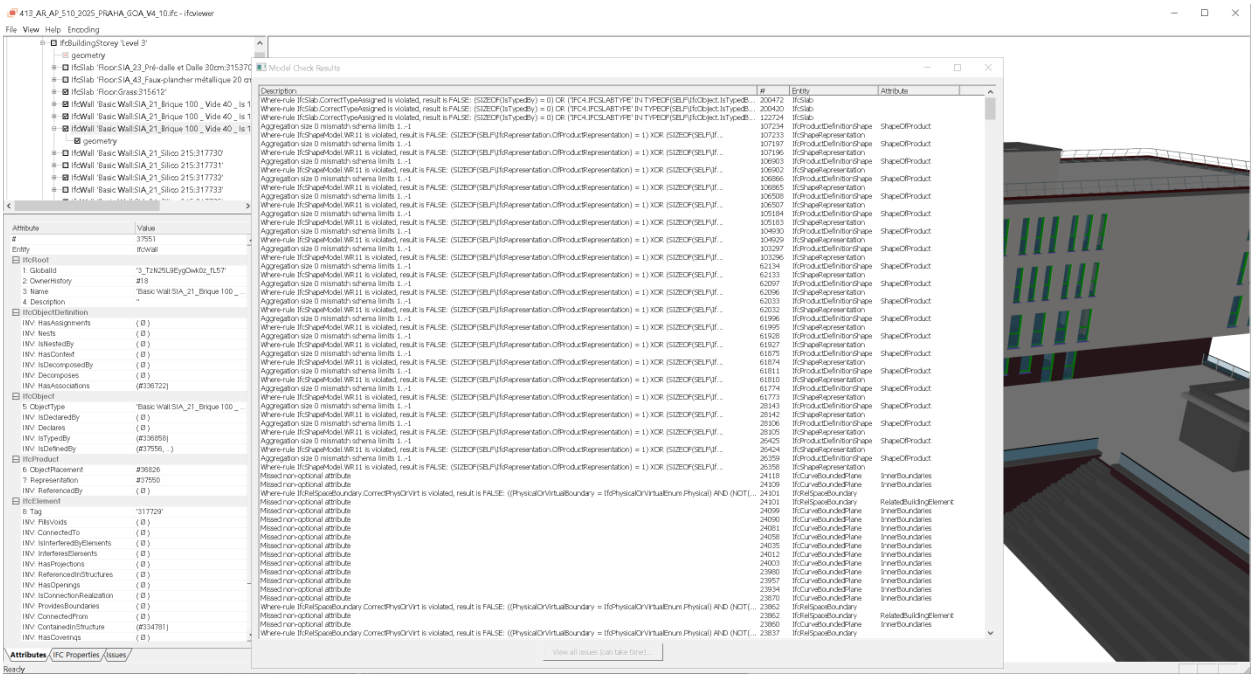

Figure 3 IFC Viewer screenshot Praha design in Autodesk Revit with 1518 identified issues.

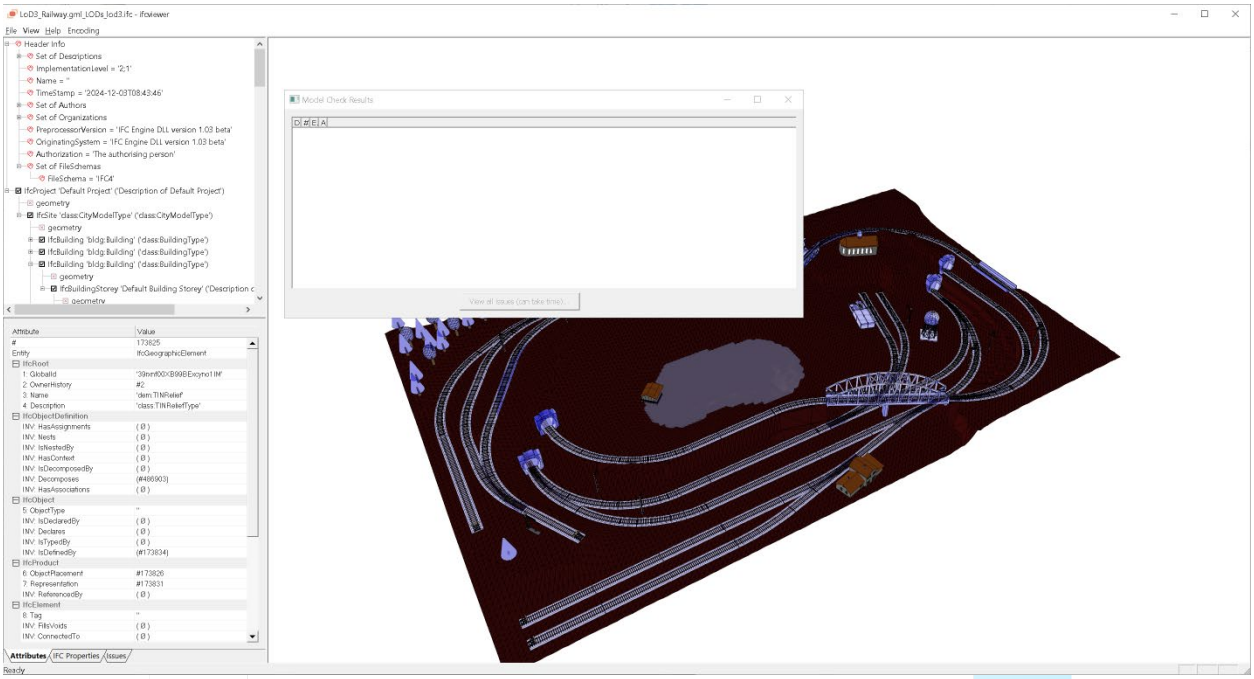

Figure 4 IFC Viewer screenshot converted CityGML model without any identified issue.

Table 1: results of the EXPRESS validation

Model name	Modelling software	Identified issues
Demo_Ascoli Piceno v1.ifc	Revit 24.2.0.63 (ENU) - IFC 24.2.0.49	107
413_AR_AP_510_2025_GAIA_MAB_V3_4.ifc	Autodesk Revit 25.1.0.44 (ENG) - IFC 25.1.0.44	753
Demo_Lisbon_Sept2024.ifc	Autodesk Revit 24.2.20.41 (ENU) - IFC 24.2.20.35	1352
413_AR_AP_510_2025_PRAHA_GOA_V4_10.ifc	Autodesk Revit 25.1.0.44 (ENG) - IFC 25.1.0.44	1518
LoD3_Railway.gml_LODs_lod3.ifc*	CHEK CityGML/CityJSON to IFC	0

The EXPRESS Validation tool is available in any of the recent IFC Engine libraries:
https://rdf.bg/downloads-all/ifc-engine-downloads/
The following example (with Microsoft Visual Studio solution and source code) can be found here
https://rdf.bg/ifcviewer/ifcviewerpackage.zip
The API calls (see Figure 5) are available from here:
https://rdf.bg/ifcdoc/CP64/validateModel.html

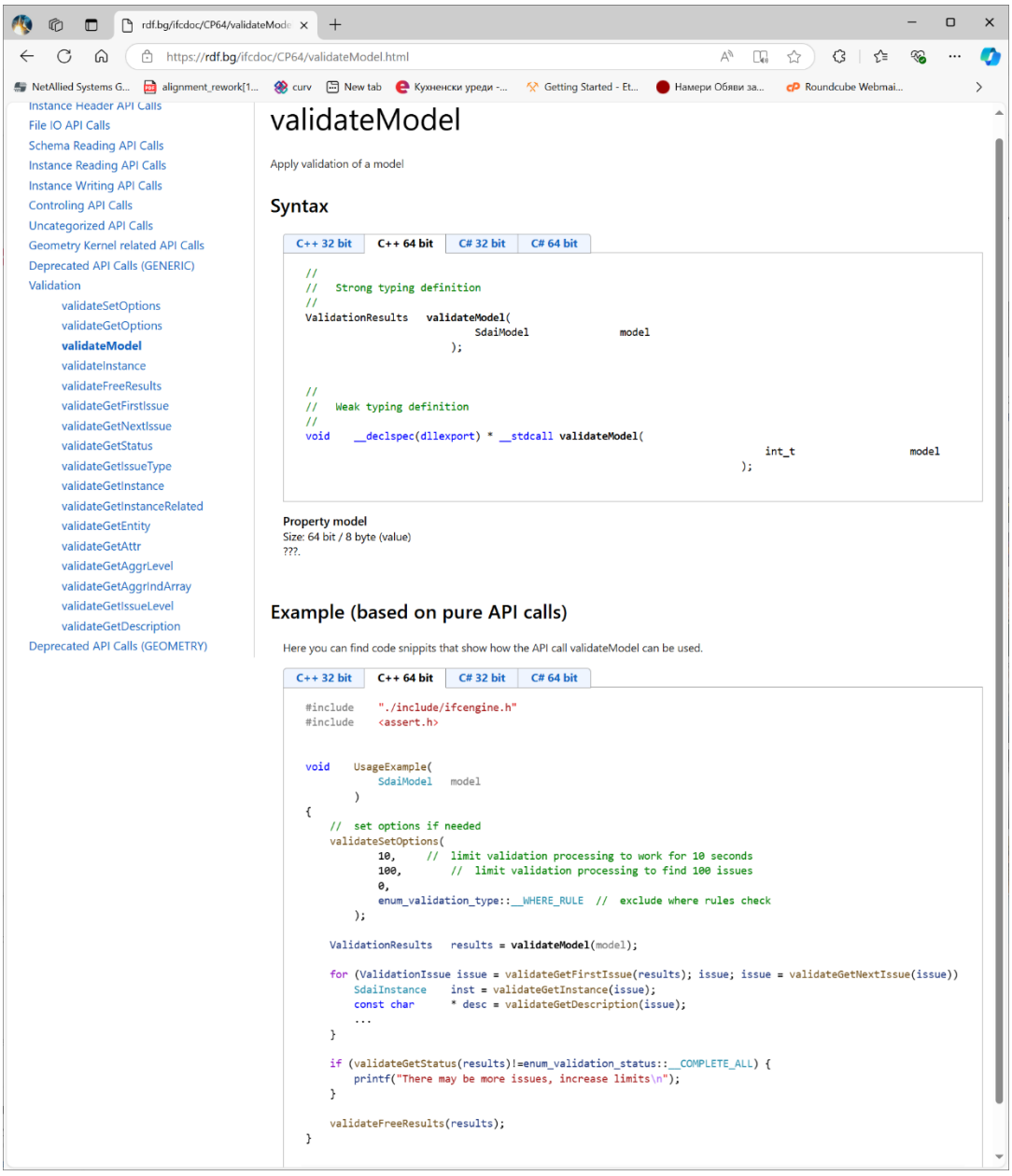

Figure 5 API of the IFC Engine library used for EXPRESS (ISO 10303-11) validation.

3.3 IDS 1.0 Validation

The selection of the Information Delivery Specification (IDS) 1.0 as the open standard to store machine interpretable requirements modelling had a long history. As mentioned before, originally mvdXML was foreseen as being the open standard for storing requirements modelling (D2.2), this was changed into using IDS as mvdXML was deprecated.

IDS is quite new, and the first official version was not released before the delivery deadline of the CHEK IFC format (D2.2). This was due to a longer and more rigid standardization process than expected. However, the official release fell before the deadline of Task 2.5. Finalization of the IDS validation tool is according to the IDS1.0 official release. Due to ongoing development in the way rules and regulations are defined, digitalized and checked, the CHEK IFC (D2.2) format itself is also still progressing. This allowed the IDS per municipality to be updated to the official IDS 1.0 schema as well.

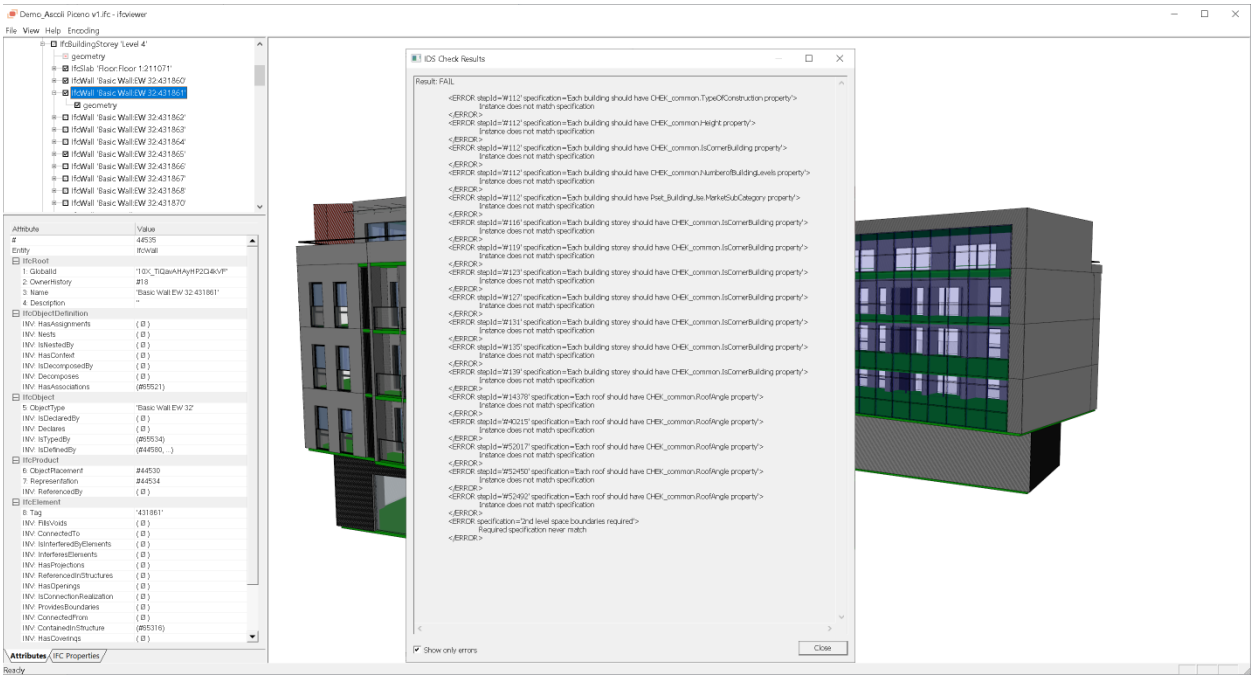

Figure 6 IFC Viewer screenshot IDS Validation in Ascoli Piceno model.

The IDS Validation tool has proven to be a multifunctional tool. It helped a lot during the development phase of both the IFC exporter (see section 4) and of the IDS itself. Outside of development support it will also play a crucial role when submitting an IFC model to a municipality. Here it has to be decided whether the information offered by the designer is complete enough to make the next step in the process. Finding the right place for the IDS validation tool in the to-be DBP process (D1.1.) is complicated. For IFC exporting tools like the one DiRoots is developing (adding on-the-fly relevant data to IFC given a certain IDS file on IFC export in Revit or ArchiCAD, see also Chapter 4) generates correct output. Even if the IFC file is incorrect or incomplete it is the question of whether the designer should be informed as he/she will most likely not be able to fix potential issues. These issues are often created outside of control of the file's author. Next to that, in many use cases users are aware that they generate IFC files that are invalid against CHEK IFC for the IDS part as these data are simply not yet available. Running the rule checking with incomplete data can still be relevant in generating valuable feedback. One place where it definitely can serve is the acceptance for municipalities of uploaded IFC files.

The results can be seen in Table 2 and Figure 6. The complete source code for the IDS validation itself (based on an SDAI API) can be downloaded from a public GitHub page:

https://github.com/I-Sokolov/RDFApps/tree/main/IDSChecker

Table 2: results of the IDS validation

Model name	IDS file	IDS 1.0 issues
Demo_Ascoli Piceno v1.ifc	CHEK_Ascoli_Piceno.ids	18
413_AR_AP_510_2025_GAIA_MAB_V3_4.ifc	CHEK_GAIA.ids	27
Demo_Lisbon_Sept2024.ifc	CHEK_Lisbon.ids	59
413_AR_AP_510_2025_PRAHA_GOA_V4_10.ifc	CHEK_Prague.ids	128

3.4 PSD Validation

The Property Set Definition (PSD) format is fairly unknown, this does not make it trivial. The CHEK IFC specification is not only defined by its schema (i.e., modelled as ISO 10303-11 as explained above), but it also exists of a large set of property-sets (PSet) and properties. They can be defined in PSD format. The CHEK IFC format defines custom PSets which can also be defined in the PSD format; this is what is done for CHEK IFC.

The results can be seen in Table 3 and Figure 7. The complete source code for the PSD validation itself (based on an SDAI API) can be downloaded from a public GitHub page:

https://github.com/I-Sokolov/RDFApps/tree/main/PSDChecker

knowledge available within PSD for the custom property-set can be defined in IDS 1.0 at the moment. Within each IFC file available issues have been identified; however these issues are relevant, there is not a lot the designers can do to prevent them. The current state of the IDS standard allows to cover all relevant information as can be stored in PSD to be embedded within the IDS file itself; for the example cases this is done. This means PSD validation finds additional issues within the developed IFC files that in principle could be identified as issues in the CAD solution used. The relevance of these issues should be identified by WP4 as these issues represent actual missing semantical knowledge that potentially could be expected when executing rule calculations.

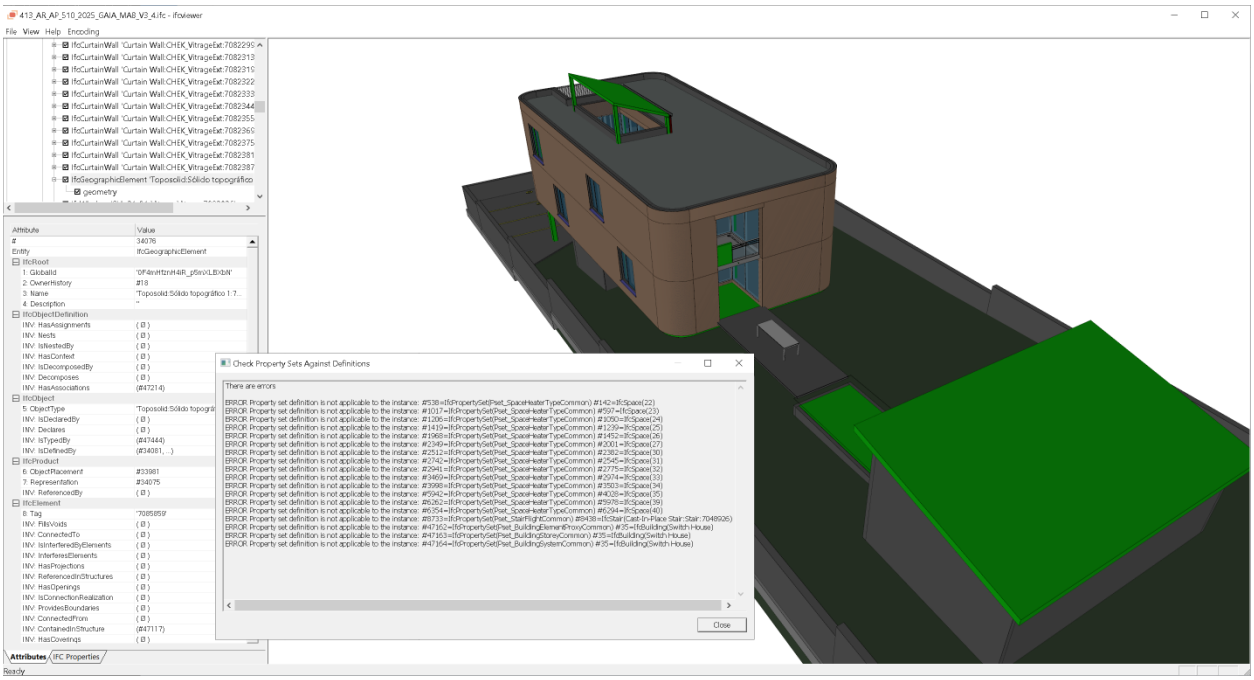

Figure 7 IFC Viewer screenshot PSD Validation in GAIA model.

Table 3: results of the PSD validation

Model name	Modelling software	PSD issues
Demo_Ascoli Piceno v1.ifc	Revit 24.2.0.63 (ENU) - IFC 24.2.0.49	803
413_AR_AP_510_2025_GAIA_MAB_V3_4.ifc	Autodesk Revit 25.1.0.44 (ENG) - IFC 25.1.0.44	18
Demo_Lisbon_Sept2024.ifc	Autodesk Revit 24.2.20.41 (ENU) - IFC 24.2.20.35	49
413_AR_AP_510_2025_PRAHA_GOA_V4_10.ifc	Autodesk Revit 25.1.0.44 (ENG) - IFC 25.1.0.44	134

3.5 Micro-Services Validation Use

There are several Micro-Services developed as part of the toolkit and each of them can be used to enrich an existing IFC model (potentially to make it from CHEK IFC invalid into CHEK IFC valid) or to validate numerical values defined by architects to be close to the truth given the geometry that can be interpreted.

Each Micro-Service can be seen as an implementation of a small algorithm and understanding of how its relevant data is translated in the IFC format (i.e., in our case IFC4 ADD2 TC1).

Each tool is offered as a CLA (Command Line Application) with under Windows a simple .exe and .bat to be executed.

There are no micro-services covering validation. The question is if these are needed at all in the current process, the typical information that could be validated is something that can be and is calculated within applications like Veerify3D and CYPE Urban anyway.

3.6 Future Steps

If one thing, the validation tooling for CHEK IFC shows that a complete and 100% correct IFC file cannot be expected from practice at the moment. This is expected to be solved once more tools become available that support the most recent version of the ISO IFC standard. An important future step for CHEK IFC will be to follow IFC4.3 ADD2 (as the most recent official ISO version). For the validation tooling this will not change much as the tools are independent of the selected IFC version and will also work out-of-the-box for newer versions of IFC (except for IFC5 as this will follow a different technology).

It is expected that IDS 1.0 validation will be adjusted. The standard is very new, and it is expected that the current software contains bugs that were not identified during the development and use till now.

It is also expected that PSD validation will become less relevant as definition of new Property Sets as can be defined in PSD can also be defined in IDS.

4. IFC Exporter

4.1 General description

The CHEK IFC Exporter tools that have been developed for Autodesk Revit (version 2022 to 2025) and Graphisoft ArchiCAD (v27) enhance the integration of IDS requirements for the IFC exports within BIM authoring software. The tools simplify populating data based on IDS requirements and streamline the IFC export workflow by allowing users to configure IDS settings and efficiently map necessary IFC data.

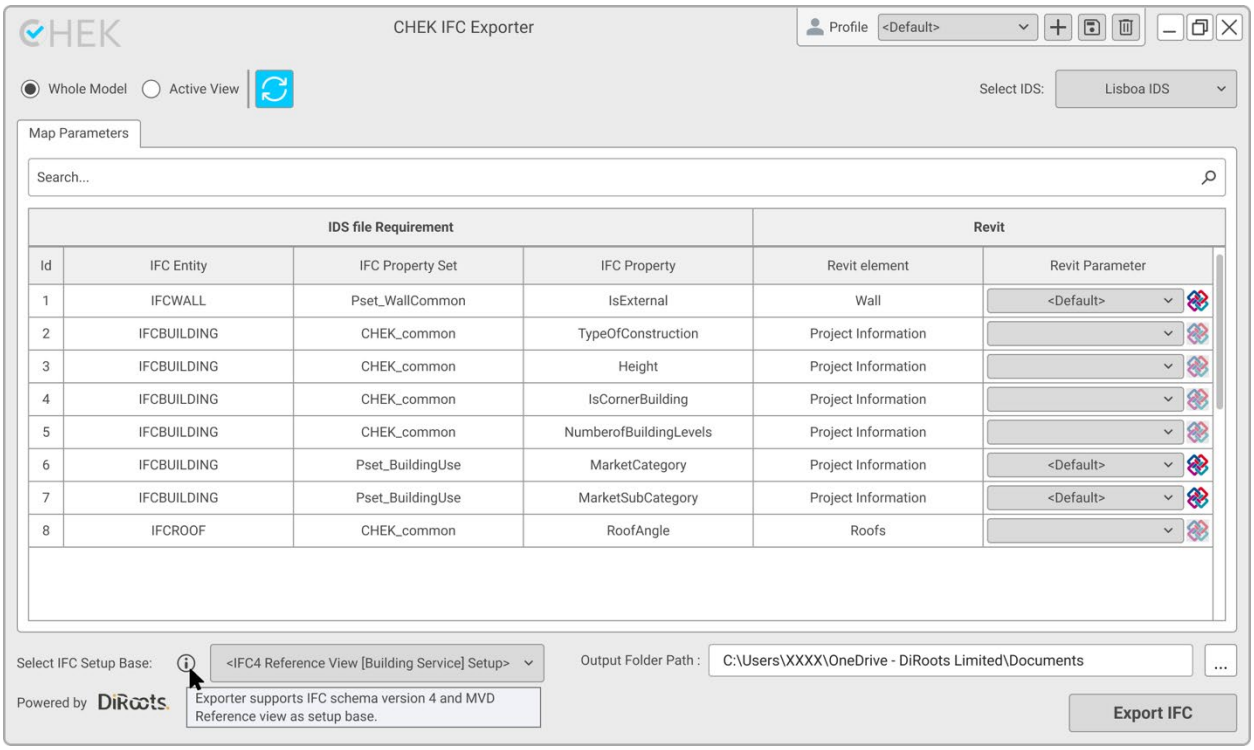

Figure 8 User Interface of Chek IFC exporter

4.2 Application Workflow

the following describes the workflow including features that the application handles.

Selecting the IDS File
Within the main interface of the Autodesk Revit plugin, located under the CHEK tab, users can select from predefined IDS files such as 'Ascoli Piceno', 'Prague', 'Lisbon', and 'Vila Nova de Gaia'. An IDS is available for each municipality which contains the information requirements for each category of checks within the CHEK scope (D2.1). The possibility to select one of the predefined IDS files ensures that appropriate IDS requirements are applied during the IFC export process.

Exporting to IFC (IDS-based Exporter)
The exporter assists users in generating IFC models that align with IDS requirements as defined by CHEK (D2.2). It supports IFC schema version 4 and utilizes the Model View Definition (MVD) Reference View as the base for export setups. While it enhances compliance with IDS specifications, full compliance is not guaranteed due to reliance on the core Autodesk Revit IFC exporter and Graphisoft ArchiCAD exporter. Full compliance before exporting is not guaranteed. The output IFC file will need to be validated by the IFC validator tools (described in section 3) to check if the user correctly mapped the information and if the authoring software exported the IFC file without mistakes.

Mapping the Properties & Custom Properties Mapping
Users can map user-defined properties from the IDS file to Revit parameters, ensuring custom data are accurately represented in the IFC export. In Figure 4, the selected IDS file shows some custom properties defined with the name of ‘CHEK_common’.

Predefined IFC Properties Mapping
The tool displays predefined IfcProperties (prefixed with 'Pset_') from the IFC schema. Users can map these to compatible Revit parameters, with options to set default mappings for efficiency.

Selecting Additional Configuration
Users can customize their IFC settings based on their requirements, using their configurations and applying changes on top of the existing IFC export setups. This provides flexibility and control over the export process.

Exportation Process
After configurations are set, users can choose to export either the active view or the entire model, accommodating different project needs and workflow preferences.

Refresh and Profile Features
For Revit the tool allows users to make changes and reload the plugin to update data seamlessly. The profile feature stores UI-related information, reducing repetitive data entry and enhancing efficiency.

4.3 Achievements

Enhanced compliance with IDS defined CHEK requirements
By facilitating the mapping of both custom and predefined properties, the tools improve the accuracy of data representation in IFC exports, ensuring better compliance with IDS defined CHEK requirements and enhancing interoperability in BIM based CHEK workflows.

Improved user experience
The intuitive interface and features like default mappings and profile storage simplify the workflow, making the tool more accessible and reducing the learning curve for new users.

Increased efficiency and flexibility
Customizable configurations and the ability to build upon existing export setups allow users to tailor the export process to their specific needs, saving time and reducing potential errors.

Seamless integration with authoring software
The tools integrate smoothly with Revit and Archicad, enabling users to maintain their existing workflows while benefiting from enhanced export capabilities.

The Chek IFC Exporter tools significantly improve the process of exporting IFC files with IDS defined CHEK requirements. They offer a more efficient, accurate, and flexible solution within BIM workflows, enhancing data integrity and supporting industry-standard practices.

5. CityGML / CityJSON data requirements and geometry validator

5.1 General description

The CityGML / CityJSON validator is a standalone application that can be used to validate the presence and characteristics of city objects in a set of CityGML and/or CityJSON files, as well as the correctness of the geometry primitives employed in them. The general operation workflow can be found in Figure 9. This workflow can be split in two different data aspects:

Data requirements (i.e., compliance / completeness checking), in which the input documents are first converted into semantic data in Resource Description Framework (RDF)[1] format after which they are validated according to a predefined requirements profile.
3D Geometry validation, using the val3dity[2] tool developed by TU Delft (see section 5.3).

Transforming the input data into semantic format provides several benefits that are essential to the workflow followed by the application:

Semantic models allow linking and merging of data from diverse sources by focusing on the underlying concepts rather than their physical representation. This capability is essential for workflows like digital building permits where there may be many heterogeneous sources of data, but commonality in the concepts and rules apply.
Different types of inputs can be supported by semantic models with minor changes in the application, since the only requirement is that the necessary elements are converted into an RDF graph using the concepts required for permit checking.
Because all inputs are merged into a single RDF graph, validations can be performed not only across objects, but also across different datasets. These datasets are not required to have the same source datatype, the validation will also function across different dataset types (e.g., CityJSON and INSPIRE).
Well-known ontologies are employed whenever possible, which means that other semantic consumers can also understand and leverage the data and metadata employed by the application (input datasets, profiles, etc.). A normative RDF representation of the CityGML data model in development will enhance this when available, the initial target RDF model for city data and the SHACL shapes used for validation could be easily adapted to it.
The rules used in the semantic models can be written using widely employed semantic web standards.

More information on how semantic technologies are leveraged can be found in section 5.4.

The validator accepts input data in both CityGML and CityJSON format, with the former being transformed into the latter using the citygml-tools3 library before performing the semantic conversion. Additionally, the validator accepts parameters that may have been defined in the requirements profile, e.g., building or location of interest. The profiles that are used by the validator will specify the input parameters necessary to target validation to the correct elements in the source data. This simplifies the problem of describing validation for all possible source data formats by focusing on specific elements. Once the validation has finished, a report is generated in JSON format. This is a standard format that can be easily used by client applications (through standard browser supported CSS style sheets) to display the validation errors that could have been encountered in the documents. For example, a 3D CityJSON viewer could run

[1] https://www.w3.org/RDF/
[2] https://github.com/tudelft3d/val3dity
[3] https://github.com/citygml4j/citygml-tools

a validation job on a given dataset and highlight any problematic objects visually, whether they relate to non-compliance with data requirements or to incorrect geometry primitives.

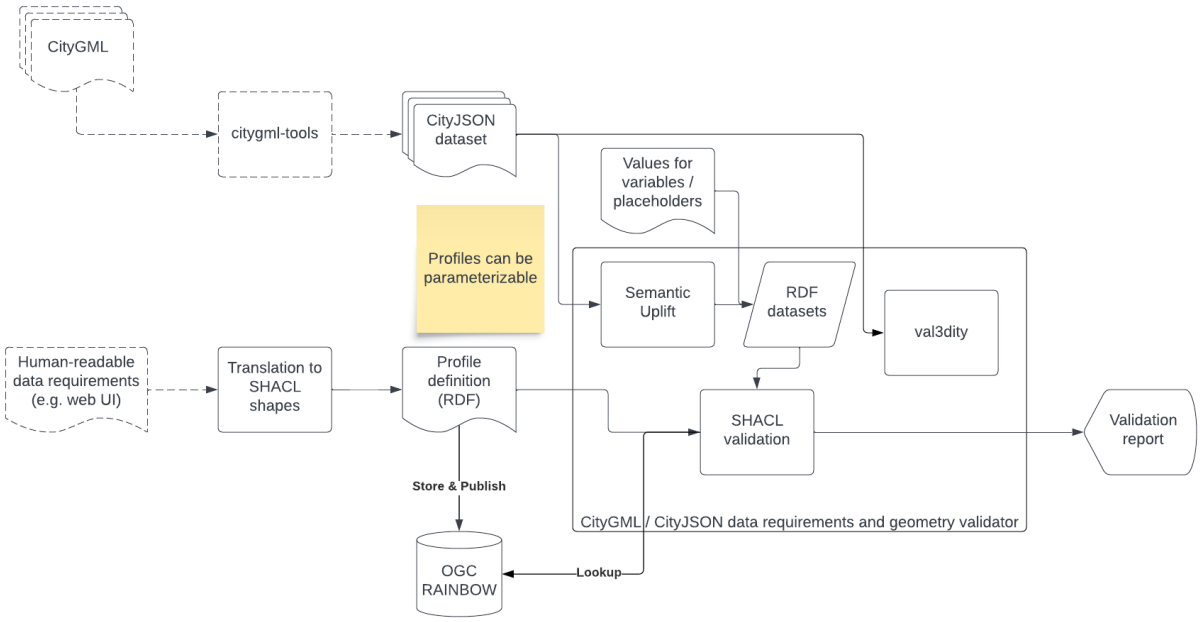

Figure 9 General operation workflow for the CityGML / CityJSON validator

The source code of the application is published on GitHub at the following URL, which also contains up-to-date documentation on the use of the service (including how to run it as a Docker container):

https://github.com/ogcincubator/chek-data-completeness.

While the input datasets and the rule collections bundled with the application are specific to CHEK, the overall methodology can be extended to support a wide array of use cases. For instance, profiles with additional checks, other than data requirements validation, can be created and used. Additionally, the semantic uplift of the input data could also be modified or enhanced to support other types of formats besides CityJSON, leveraging the constraint definition and checking capabilities offered by semantic technologies.

5.2 Architecture and interoperability

The validation tool is offered as a web application, compliant with the OGC API – Processes[4] standard developed by the Open Geospatial Consortium. It is also packaged as a Docker[5] image for convenience, making it easier for users, software vendors, or system administrators to integrate in their environments.

By offering a standard, JSON-based OGC API – Processes interface, the application can be easily integrated with other tools and libraries. A user-friendly HTML interface is also available, see Figure 10, allowing the tool to be used as a standalone application. The validation results being displayed at the bottom once the validation is finished.

This means that the application supports two different use patterns:

As a web application through which users can upload their datasets and validate them against a set of rule collections (profiles)
As a service that can be integrated in third-party tools. For example, a Revit plugin could execute validation jobs on a given dataset.

The flexibility offered by this integration architecture means that the validator can be used outside of the CHEK workflow (e.g., by municipalities that wish to check the completeness of their datasets), or embedded in it (e.g., invoked by another application in the pipeline).

A set of default, sample profiles is bundled with the application, but the profiles can also be fetched from other data sources, such as the OGC RAINBOW[6] instance where the CHEK Project profiles (which are currently under development) are to be hosted.

[4] https://ogcapi.ogc.org/processes/
[5] https://www.docker.com/
[6] https://defs.opengis.net/vocprez/

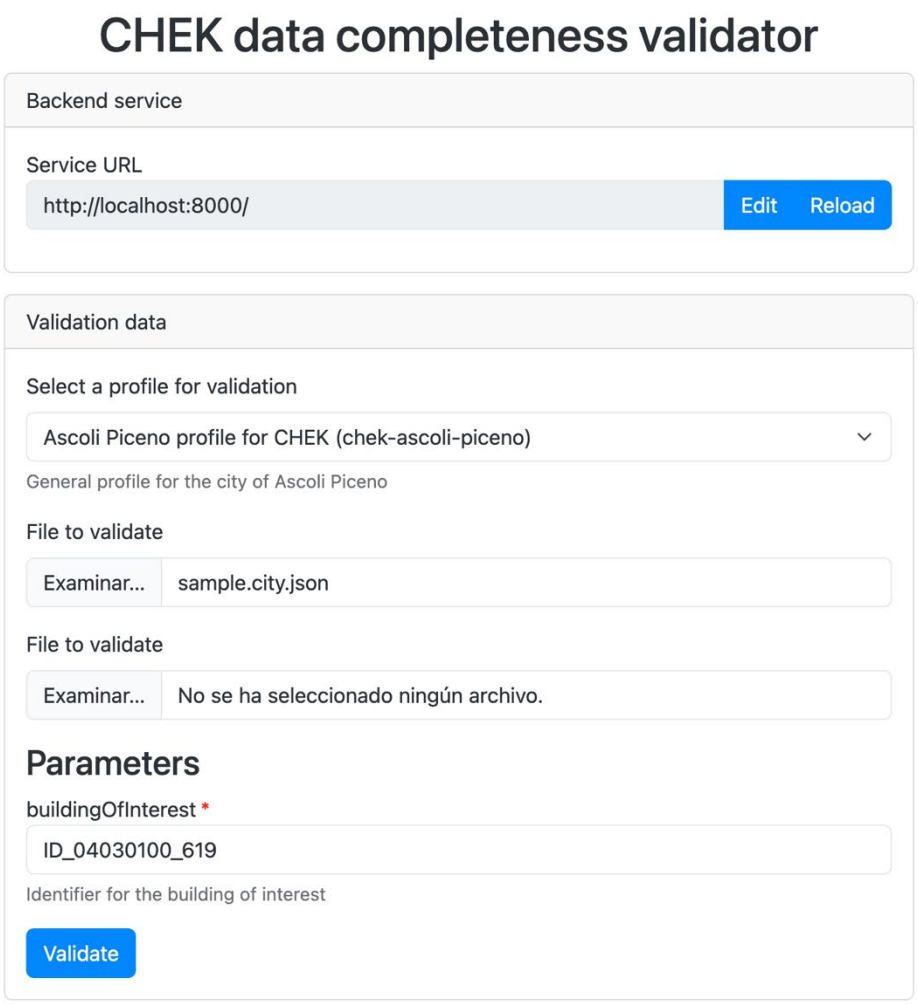

Figure 10 Look and feel of the HTML interface

5.3 Geometry validation

The CityJSON validator has the capability to check the geometry of the objects that are present in the input files. The validator utilizes the tool val3dity for this. This is an open-source geometry validator that has been developed by the TU Delft prior to the CHEK project. Its development has not been part of the CHEK project and due to this it does not take a central role within D2.4 nor the architecture of this validator. However, it demonstrates the extend of the solution, and since the geometry validation is done by val3dity, its functioning will be covered at a surface level.

Val3dity validates 3D primitives according to ISO19107. It supports the use of MultiSurface, CompositeSurface, Solid, MultiSolid and CompositeSolid geometry. The geometry validator supports inner rings and cavities. It does, however,

not support parametrically modelled primitives, such as curved edges and dome shaped surfaces. CityJSON supports this neither, so this is not seen as a significant issue.

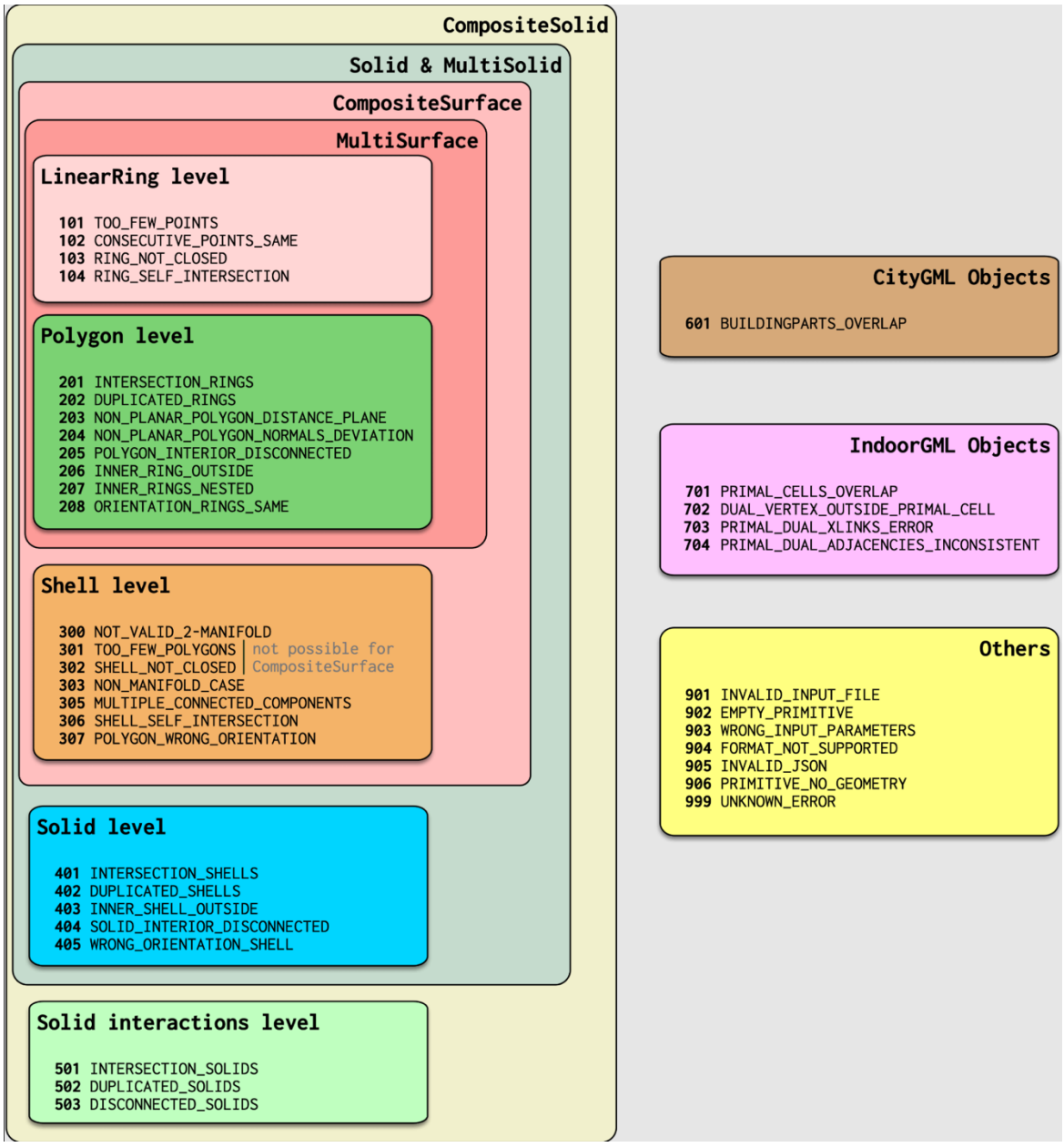

Figure 11 Issues that val3dity is able to detect

Issues that the geometry validator encounters are clearly structured and replied back to the user via a report. Figure 11 shows a clear overview of the issues that the geometry validator can encounter. The error code list in this figure does also give a good overview of the different sorts of validations that are executed. These error codes are not only

errors, but they also include some warnings. For example, 102 CONSECUTIVE_POINTS_SAME, does usually not cause terminal errors in downstream applications, but it is good practice to resolve the cause.

The tool does come with some limitations, the major one is the limited overlap/intersection validation between different objects. The tool primarily validates the geometric representations as single objects. If for example two different building representations intersect with each other it will not be noted by the geometry validator. The only exception for this is CityJSON objects of type "Building" that have child objects of type "BuildingPart". For the BuildingPart representations it is validated if they overlap/intersect.

This means that geometry validation only guarantees that the geometry itself is valid, but not that the city model is without geometric issues. E.g. buildings could still intersect with each other and/or the ground plane. However, the downstream application that are utilizing the CHEK CityJSON files can assume all the geometry to adhere to ISO19107 if no issues were encountered by the geometry validator.

5.4 Semantic data and semantic validation

Semantic and Linked Data technologies and standards that provide a flexible core for the requirements validator, the most relevant of which is the Resource Description Framework (RDF). RDF can be used for describing "resources", which can be anything ranging from physical things, to documents, to abstract concepts (e.g., "a chair", "The Adventures of Huckleberry Finn", "a trip to Las Vegas" or "technology"), by using a collection of simple subject-predicate-object statements (also called triples):

The subject: a resource or entity for which something is being described.
The predicate: the relationship between the subject and the object.
The object: the target or the value for the relationship.

For example, the sentence "Alejandro works for the OGC" can be represented in RDF with the entity "Alejandro" as the subject, the property "is employed by" as the predicate, and the "OGC" as the object. A collection of such triples can be thought of as a directed graph.

Input city data, in various formats, is first converted into a common, standardized semantic format using such assertions, and merged into a single graph. This is achieved through a process called "semantic uplift", which entails targeting relevant elements of the source data, converting to a simplified JSON format, and using JSON-LD to semantically "annotate" the resources to build the necessary relationships (triples). This is a multi-step process, but maximises both flexibility and use of standards, and is far easier to maintain, extend and re-use than more typical custom code approaches. This enables validation to be carried out not only within the context of a single CityGML / CityJSON file, but across a whole dataset composed of several documents. Thus, validation rules can be standalone (i.e., affecting only one single dataset or city object) or complex (i.e., affecting city objects contained in different datasets).

```
:BuildingOfInterestRequirements
  a sh:NodeShape ;
  sh:targetNode chek:document ;
  sh:sparql [
    sh:prefixes ex: ;
    sh:select """
      SELECT ?buildingOfInterest (?value as ?path) ?value ?req WHERE {
        ?buildingOfInterestParam a sd:Parameter ;
          dct:identifier "buildingOfInterest" ;
          sd:hasFixedValue ?buildingOfInterestValue ;
        .
        ?dataset city:hasObject ?buildingOfInterest .
        ?buildingOfInterest dct:identifier ?buildingOfInterestValue .
        {
          FILTER NOT EXISTS {
            ?buildingOfInterest city:hasGeometry ?geometry .
            ?geometry city:hasSurface/rdf:type city:RoofSurface ;
              city:lod ?lod .
            FILTER(REGEX(?lod, "^[23].*"))
          }
          BIND(city:RoofSurface as ?value)
          BIND("RoofSurface with LoD 2+" as ?req)
        } UNION {
          FILTER NOT EXISTS {
            ?buildingOfInterest city:hasGeometry ?geometry .
            ?geometry city:hasSurface/rdf:type city:WallSurface ;
              city:lod ?lod .
            FILTER(REGEX(?lod, "^[34].[1-4]"))
          }
          BIND(city:WallSurface as ?value)
          BIND("WallSurface with LoD 3.1+" as ?req)
        } UNION {
          FILTER NOT EXISTS {
            ?buildingOfInterest city:hasGeometry/city:hasSurface ?surface .
            ?surface a city:WallSurface ;
              attr:hasWindows ?hasWindows .
            FILTER(?hasWindows IN (1, "true", true))
          }
          BIND(city:WallSurface as ?value)
          BIND("WallSurface with attribute hasWindows = true or 1" as ?req)
        }
      }
    """ ;
  ] ;
  sh:message "Building of interest does not satisfy requirement: {?req}" ;
  sh:severity sh:Violation ;
.
```

Figure 12 Sample SHACL shape for a data requirements rule in RDF Turtle format

Once the diverse forms of (relevant) city data is available using standard-based models in RDF graphs they can be validated using available standards and tools, in particular the Shapes Constraint Language (SHACL)[7], developed by the World Wide Web Consortium. Each validation rule can be mapped to one or more SHACL shapes with varying

[7] https://www.w3.org/TR/shacl/

complexity. The shapes can even be defined by using SPARQL[8], a semantic query language for RDF graphs similar to the Structured Query Language (SQL) used in relational databases. An example of a shape that validates the presence and several additional aspects of a given building is shown on Figure 12

5.5 Data models

5.5.1 Profile definition

Data requirement profiles are defined using the RDF Profiles Vocabulary[9] to describe their metadata, and a collection of SHACL shapes containing the actual checks to be performed. Other vocabularies are used for metadata properties, such as the Dublin Core Metadata Initiative (DCMI) Metadata Terms[10] for commonly used properties, or the Software Description Ontology[11] and the Hydra Core Vocabulary[12] for the profile input parameters. A sample profile definition, in RDF Turtle format, and with the core metadata items supported by the application, a SHACL shapes artifact (i.e., document containing the actual shapes), and a single optional input parameter ("myParameter"), can be seen in Figure 13.

```
@prefix chekp: <urn:chek:profiles/> .
@prefix prof: <http://www.w3.org/ns/dx/prof/> .
@prefix dct:  <http://purl.org/dc/terms/> .
@prefix role: <http://www.w3.org/ns/dx/prof/role/> .
@prefix sd: <https://w3id.org/okn/o/sd#> .
@prefix hydra: <http://www.w3.org/ns/hydra/core#> .

chekp:sample a prof:Profile, chekp:Profile ;     # Only instances of checkp:Profile are processed, so this is required
  dct:title "Sample profile for CHEK" ;          # A title for the profile
  dct:hasVersion "0.1" ;                         # Profile version
  prof:isProfileOf chekp:chek ;                  # prof:isProfile of can be used for declaring inheritance
  prof:hasToken "chek-ascoli-piceno" ;           # A token is required and will be used to identify the profile
  prof:hasResource [                             # At least one resource must be described
    a prof:ResourceDescription ;
    prof:hasRole role:validation ;                        # The role must be role:validation
    dct:format <https://w3id.org/mediatype/text/turtle> ;  # Optional
    dct:conformsTo <https://www.w3.org/TR/shacl/> ;        # Conforming to https://www.w3.org/TR/shacl/ is *mandatory*
    prof:hasArtifact <./ap-shapes.shacl> ;                 # Path or URL to SHACL shapes file
  ] ;
  sd:hasParameter [                            # Zero or more parameters can also be declared
    dct:identifier "myParameter" ;               # Identifier that will be used when running validations
    dct:description "Sample argument" ;          # An optional description for the parameter
    sd:hasDataType "string" ;                    # Data type of the parameter
    hydra:required false ;                       # Whether the parameter is required (true) or optional (false)
  ] ;
.
```

Figure 13 Sample metadata definition for a data requirements profile

[8] https://www.w3.org/TR/sparql11-query/
[9] https://www.w3.org/TR/dx-prof/
[10] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/
[11] https://w3id.org/okn/o/sd
[12] https://www.hydra-cg.com/spec/latest/core/

5.5.2 City RDF model

Any SHACL shapes used in the validator must be tailored to the specific "City RDF" model employed by the application. Given that no standard RDF ontology exists for the CityGML (or CityJSON) conceptual / data models, a custom target RDF model is used. The following is a summary of the data conversion workflow, also depicted in Figure 14 follows:

If the input document is in CityGML format, convert to CityJSON using citygml-tools.
"@id" fields, which will be later used as RDF resource identifiers (URIs) are added to all city objects.
The semantics of the geometric primitives[13] are unrolled to shape them in a graph-compatible format.
All geometries are converted to the TopoFeature[14] representation format.
Vertices are also converted to TopoFeature, and their coordinates represented as individual properties.
JSON-LD15 context is added to the document to semantically enable it.
The resulting document is an RDF graph to which the validation SHACL shapes can be applied to.

The full semantic uplift definition document that implements this workflow can be found at:

https://github.com/ogcincubator/chek-data-completeness/blob/master/data/cityjson-uplift.yml.

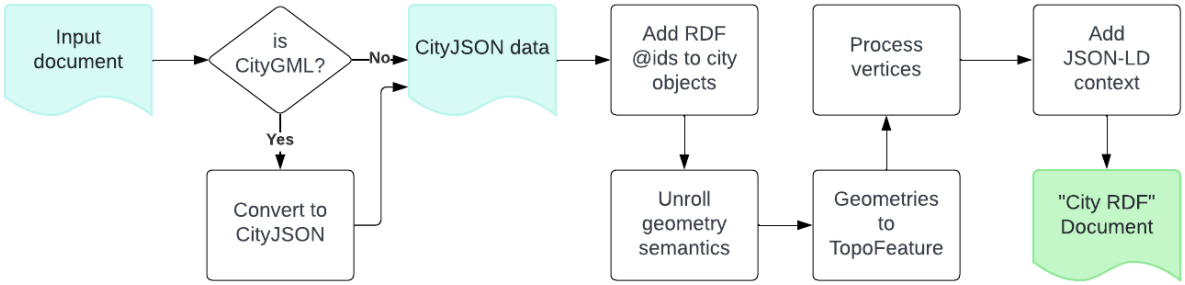

Figure 14 Semantic uplift workflow for CityGML / CityJSON data

In general terms, for a standard CityJSON document, the resulting RDF graph has the following characteristics

A Uniform Resource Names (URN) namespace is defined for the document, which will be used as the base Uniform Resource Identifier (URI) for all the RDF resources in it.
A resource of type city: City is created for the document.

A resource is created for each city object, setting its type to city:X, where X is the type of CityJSON object (Building, BuildingPart, Road, etc.). Each city object is assigned a unique URI generated from their "id"; the object "id" is kept, verbatim, in a dct:identifier predicate. A city:hasObject relationship is established between the city:City and each object resource.

A resource is created for each vertex, with an individual URI assign to it. The type of each of these vertices is set to geojson:Feature. Each vertex contains a city:geometry whose value is a blank node with type cityjson:Point and an

[13] https://www.cityjson.org/specs/2.0.1/#semantics-of-geometric-primitives
[14] https://ogcincubator.github.io/topo-feature/
[15] https://json-ld.org/

RDF list of the vertex coordinates using a geojson:coordinates predicate. A city:hasVertex relationship is established between the city:City and each vertex resource.

The “transform” object is converted into a blank node with city:scale and city:translate predicates for the scale and translate properties, respectively, each in turn containing city:x, city:y and city:z predicates corresponding to their values. This transform object is bound to the city:City by city:hasTransform.

Each city object can have a city:hasGeometry predicate with one or more geometry objects. A geometry object has a city:lod predicate for its Level of Detail (LoD), and a city:hasSurface to represent its surface.
Surfaces have a GML type from the GML ontology (e.g., http://www.opengis.net/ont/gml#MultiSurface), and a city:boundaries relationship to a TopoFeature resource describing the surface boundaries. To that end, the nested boundary arrays used in CityJSON are converted into a TopoFeature geometry hierarchy (e.g., MultiSurface → Polyhedron → MultiPolygon → …), in which each level is bound to the next through geojson:relatedFeatures. geojson:relatedFeatures is also employed to link each primitive to a list of its coordinates (vertex objects as described above).

Attributes are preserved with their names but using the attr: namespace (e.g., “hasWindows” → attr:hasWindows), bound to the objects that contains them.

“city:” refers to the “http://example.com/vocab/city/” URI namespace “geojson:” to the URI namespace for https://purl.org/geojson/vocab# “dct:” to “http://purl.org/dc/terms/” “attr:” to http://example.com/vocab/city/attr#

A (simplified) sample CityJSON document converted in RDF/Turtle can be found in Figure 15

```
@prefix c: <https://example.com/city-topo-feature#> .
@prefix city: <http://example.com/vocab/city/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix geojson: <https://purl.org/geojson/vocab#> .
@prefix gml: <http://www.opengis.net/ont/gml#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

c:city a city:City ;
    city:geographicalExtent [ city:max [ city:x 1.0 ;
                    city:y 1.0 ;
                    city:z 1.0 ] ;
            city:min [ city:x 0.0 ;
                    city:y 0.0 ;
                    city:z 0.0 ] ] ;
    city:hasExtension <https://cityjson.org/extensions/download/generic.ext.json> ;
    city:hasObject c:city-objects-id-1 ;
    city:hasTransform [ city:scale [ city:x 0.001 ;
                    city:y 0.001 ;
                    city:z 0.001 ] ;
            city:translate [ city:x 0.0 ;
                    city:y 0.0 ;
                    city:z 0.0 ] ] ;
    city:hasVertex c:vertices-0,c:vertices-1, c:vertices-2, c:vertices-3, c:vertices-4, c:vertices-5,c:vertices-6,c:vertices-7 ;
    city:version "1.1" .

<https://cityjson.org/extensions/download/generic.ext.json> dct:identifier "Generic" ;
    dct:version "1.0" .

c:city-objects-id-1 a <https://www.cityjson.org/extensions/download/generic.ext.json#GenericCityObject> ;
    city:hasFunction "something" ;
    city:hasGeometry [ city:hasSurface [ a gml:Solid ;
                    city:boundaries ( c:city-objects-id-1_geom_0 ) ] ;
            city:lod "3.3" ] ;
    dct:identifier "id-1" .

c:city-objects-id-1_geom_0 a geojson:Feature ;
    geojson:topology [ a geojson:Polyhedron ;
            geojson:relatedFeatures ( c:city-objects-id-1_geom_0_1 ) ] .

c:city-objects-id-1_geom_0_1 a geojson:Feature ;
    geojson:topology [ a geojson:MultiPolygon ;
            geojson:relatedFeatures ( c:city-objects-id-1_geom_0_1_1 c:city-objects-id-1_geom_0_1_2 c:city-objects-id-1_geom_0_1_3 c:city-
                objects-id-1_geom_0_1_4 c:city-objects-id-1_geom_0_1_5 c:city-objects-id-1_geom_0_1_6 ) ] .

c:city-objects-id-1_geom_0_1_1 a geojson:Feature ;
    geojson:topology [ a geojson:Polygon ;
            geojson:relatedFeatures ( c:city-objects-id-1_geom_0_1_1_1 ) ] .

...omitted...

c:vertices-0 a <https://example.com/Feature> ;
    city:hasGeometry [ a geojson:Point ;
            geojson:coordinates 0,
                1000 ] ;
    city:x 0.0 ;
    city:y 0.0 ;
    city:z 1000.0 .

c:vertices-1 a <https://example.com/Feature> ;
    city:hasGeometry [ a geojson:Point ;
            geojson:coordinates 0,
                1000 ] ;
    city:x 1000.0 ;
    city:y 0.0 ;
    city:z 1000.0 .

...omitted...
```

Figure 15 Sample RDF City dataset

5.5.3 Validation report

The report generated by the application is returned as a JSON object that adheres to the following structure:

"`valid`": Boolean value (true or false) indicating whether validation passed.
"`val3dityResult`": Boolean value (true or false) indicating whether val3dity validation passed.
"`shaclResult`": Boolean value (true or false) indicating whether profile (i.e., semantic rule collection) validation passed.
"`shaclReport`": Full SHACL report[16] containing the result of the profile validation.
"`fileValidation`": List (JSON array) of values, one per input file, with individual val3dity validation results.

The value for "`shaclReport`" is a JSON object obtained from parsing the SHACL report (generated in RDF format by the SHACL module) and formatting it as JSON-LD using a mechanism called JSON-LD framing[17], which makes it possible to deterministically define the layout (property names, object nesting, etc.) of the resulting JSON-LD document. In our case, the SHACL namespace is removed from the URIs (resulting in much more traditional-looking JSON field names), and the report is laid out using the global SHACL ValidationReport object as the root node. The specific JSON-LD frame used in our validator can be found at the following URL: https://github.com/ogcincubator/chek-data-completeness/blob/059d592b31ae87bff8a799e36c0d9486522fcfab/app/jobs.py#L22

The entries inside "`fileValidation`" use the following format:

"`fileIndex`": the 0-based index denoting the order of the file in the input dataset.
"`name`": the file name as provided by the user / client.
"`valid`": whether this individual file passed val3dity validation.
"`val3dityReport`": the verbatim output provided by val3dity. val3dity's documentation[18] contains more information about the specific format of this report[19] and the types of errors that can be detected[20].

An example of a simple, successful validation report using a dummy profile and the CityJSON cube example can be seen in Figure 16. Figure 17 shows an excerpt of the "`fileValidation`" entry for a roads dataset with geometry validation errors. Finally, a sample SHACL validation report with unmet constraints is shown in Figure 18..

[16] https://www.w3.org/TR/shacl/#validation-report
[17] https://www.w3.org/TR/json-ld11-framing/
[18] https://val3dity.readthedocs.io/2.5.1/index.html
[19] https://val3dity.readthedocs.io/2.5.1/usage.html#how-to-interpret-the-report
[20] https://val3dity.readthedocs.io/2.5.1/errors.html

```
{
    "valid": true,
    "val3dityResult": true,
    "shaclResult": true,
    "shaclReport": {
        "@context": {
            "shacl": "http://www.w3.org/ns/shacl#",
            "@vocab": "http://www.w3.org/ns/shacl#",
            "result": {
                "@container": "@set"
            },
            "focusNode": {
                "@type": "@id"
            },
            "resultPath": {
                "@type": "@id",
                "@container": "@set"
            },
            "resultSeverity": {
                "@type": "@id"
            }
        },
        "@type": "ValidationReport",
        "conforms": true
    },
    "fileValidation": [{
            "fileIndex": 0,
            "name": "cube.city.json",
            "valid": true,
            "val3dityReport": {
                "all_errors": [],
                "dataset_errors": [],
                "features": [{
                        "errors": [],
                        "id": "id-1",
                        "primitives": [
                            {
                                "errors": [],
                                "id": "0",
                                "numberfaces": 6,
                                "numbershells": 1,
                                "numbervertices": 8,
                                "type": "Solid",
                                "validity": true
                            }],
                        "type": "+GenericCityObject",
                        "validity": true
                    }],
                "features_overview": [{
                        "total": 1,
                        "type": "+GenericCityObject",
                        "valid": 1
                    }],
                "input_file": "tmp/d4/d4b34447-d223-4603-a41c-fade856dbe46/input_city.0.json",
                "input_file_type": "CityJSON",
                "parameters": {
                    "overlap_tol": -1,
                    "planarity_d2p_tol": 0.01,
                    "planarity_n_tol": 20,
                    "snap_tol": 0.001
                },
                "primitives_overview": [ {
                        "total": 1,
                        "type": "Solid",
                        "valid": 1
                    }],
                "time": "Thu Nov 21 11:26:34 2024 UTC",
                "type": "val3dity_report",
                "val3dity_version": "2.4.0",
                "validity": true
            }
        }]
}
```


Figure 16 Successful validation of CityJSON cube example

```
{
    "valid": false,
    "val3dityResult": false,

  ...omitted...

  "fileValidation": [
        {
            "fileIndex": 0,
            "name": "roads-lod2__v3.json",
            "valid": false,
            "val3dityReport": {
                "all_errors": [
                    102,
                    104,
                    906
                ],
                "dataset_errors": [],
                "features": [
                    {
                        "errors": [
                            {
                                "code": 906,
                                "description": "PRIMITIVE_NO_GEOMETRY",
                                "id": "Feature has no geometry defined (or val3dity doesn't handle this type).",
                                "info": "",
                                "type": "Error"
                            }
                        ],
                        "id": "ID_003e6833-b969-3457-bc06-923d6a11eef9",
                        "primitives": null,
                        "type": "Road",
                        "validity": false
                    },
                    {
                        "errors": [
                            {
                                "code": 906,
                                "description": "PRIMITIVE_NO_GEOMETRY",
                                "id": "Feature has no geometry defined (or val3dity doesn't handle this type).",
                                "info": "",
                                "type": "Error"
                            }
                        ],
                        "id": "ID_09043a5e-94ae-3dde-ab21-d60b24309489",
                        "primitives": null,
                        "type": "Road",
                        "validity": false
                    }
                ]
            }
        }
    ]
}
```


Figure 17 Excerpt of a val3dity report with incorrect geometry primitives

```
{
    "valid": false,
    "val3dityResult": true,
    "shaclResult": false,
    "shaclReport": {
        "@context": {
            "shacl": "http://www.w3.org/ns/shacl#",
            "@vocab": "http://www.w3.org/ns/shacl#",
            "result": {
                "@container": "@set"
            },
            "focusNode": {
                "@type": "@id"
            },
            "resultPath": {
                "@type": "@id",
                "@container": "@set"
            },
            "resultSeverity": {
                "@type": "@id"
            }
        },
        "@type": "ValidationReport",
        "conforms": false,
        "result": [{
                "@type": "ValidationResult",
                "focusNode": "urn:chek:vocab/document",
                "resultMessage": "Building of interest not found in dataset",
                "resultPath": ["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
                "resultSeverity": "shacl:Violation",
                "sourceConstraint": {
                    "prefixes": {
                        "@id": "http://example.org/"
                    },
                    "select": "\n      SELECT $this (rdf:type as ?path) (?buildingOfInterestValue as ?value) WHERE {
                    \n        ?buildingOfInterestParam a sd:Parameter ;\n          dct:identifier \"buildingOfInterest\" ;
                    \n          sd:hasFixedValue ?buildingOfInterestValue ;\n        .\n        FILTER NOT EXISTS {
                    \n          ?dataset city:hasObject/dct:identifier ?buildingOfInterestValue\n        }
                    \n      }\n    "
                },
                "sourceConstraintComponent": {
                    "@id": "shacl:SPARQLConstraintComponent"
                },
                "sourceShape": {
                    "@id": "urn:chek:profiles/ascoli-piceno#BuildingOfInterestPresent"
                },
                "value": "12345"
            },
            {
                "@type": "ValidationResult",
                "focusNode": "urn:chek:vocab/document",
                "resultMessage": "Dataset contains no Road objects",
                "resultSeverity": "shacl:Violation",
                "sourceConstraintComponent": {
                    "@id": "shacl:NotConstraintComponent"
                },
                "sourceShape": {
                    "@id": "urn:chek:profiles/roads-present#RoadsPresent"
                },
                "value": {
                    "@id": "urn:chek:vocab/document"
                }
            }]
    },

  ...omitted...
}
```


Figure 18 Sample SHACL validation report with errors

5.6 Results, and next steps

The methodology described for the mode of operation of the validator shows how semantic technologies can be leveraged to codify data validation rules (in this case, data requirements, but other types of constraints can also be defined) that perform checks not only across different objects or entities, but across several datasets as well. The rules can be defined in a declarative manner (i.e., avoiding general-purpose computer code that may introduce side effects), following well-established standards, and with a set of metadata that makes them easily identifiable and helps interpret the results of their application.

While the methodology and implementation described here have been proven effective, certain limitations must be considered:

The use of a common (standard) City RDF specification would be desirable. However, interoperability, both on the data and rule-checking levels, can be achieved even with our ad-hoc conceptual model.
The conversion from CityGML to CityJSON is not guaranteed to be lossless. This process could be further refined by transforming CityGML into RDF directly.
While the GeoSPARQL[21] standard defines a set of classes, properties and functions for representing and working with geometries in RDF, support for 3D geometry is missing. This means that (currently) any checks involving geometries are impossible to implement using the RDF-based validator. Therefore, they are checked with Val3dity that has been developed outside of CHEK for this purpose.

The flexibility and expressivity that can be achieved using semantic technologies are in a trade-off relationship with the ease and user-friendliness of writing the actual rules. However, while intimate knowledge of both the ad-hoc city data models and semantic technologies (RDF, SHACL, SPARQL) is currently required for such a task, interfaces can be developed to support a range of common use cases, leaving only the most complex ones to be written by hand.

Displaying report results in a more user-friendly manner is also something that will be revisited in the near future. Right now, the HTML interface of the tool, including error reporting, is mostly geared towards demonstrating how the methodology can be applied to the task at hand. However, due to the ease and standards-compliance of the integration mechanisms available, other interfaces can also be developed for the same service. For example, as a Revit plugin that connects to the service and interprets the validation results for the end user.

[21] https://www.ogc.org/es/publications/standard/geosparql/

List of Figures

List of Tables

List of used abbreviations

BIM	-	Building information modelling
CHEK	-	Change toolkit for digital building permit
DBP	-	Digital Building Permit
DCMI	-	Dublin Core Metadata Initiative
DIR	-	DiRoots
EC	-	European Commission
IDS	-	Information Delivery Specification
IFC	-	Industry Foundation Classes
JSON	-	JavaScript Object Notation
JSON-LD	-	JavaScript Object Notation for Linked Data
LoD	-	Level of Detail
OGC	-	Open Geospatial Consortium
PSD	-	Property Set Definition
PSet	-	Property Set
RDF	-	Resource Description Framework
SHACL	-	Shapes Constraint Language
SPARQL	-	SPARQL Protocol and RDF Query Language
SQL	-	Structured Query Language
TUD	-	Technische Universiteit Delft (Delft University of Technology)
URI	-	Uniform Resource Identifier
URN	-	Uniform Resource Names
WP	-	Work Package
XML	-	Extensible Markup Language
XSD	-	XML Schema Definition