METRICS FOR THE ANALYSIS OF PRODUCT MODEL COMPLEXITY

Ulrich Hartmann, Research Assistant , Ulrich.Hartmann@kit.edu
Petra von Both, Professor and Department Head, Petra.vonBoth@kit.edu
Department of Building Lifecycle Management, Karlsruhe Institute of Technology, Germany

ABSTRACT

Today we see several product model standards getting more and more corpulent by absorbing concepts of neighboring domains, but behind the good intentions of getting 'complete' the perils of complexity are lurking. Raising computer power gives us the means of handling large digital models, but the overall situation resembles the scenery of the mid-1970s, where the software industry ran into the so-called software crisis. Edsger Dijkstra (Dutch computer scientist 1930- 2002, Turing Award, Dijkstra's Algorithm, Structured Programming) put it quite bluntly: "as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem" (The Humble Programmer, Edsger W. Dijkstra, 1972). Pursuing traditional concepts with growing tool power may lead to structural deficiencies not anticipated before.

It is in the nature of complexity to have no single 'magic' number, representing the complexity of a general system, at hand. The comparison of system complexity at a universal level is therefore next to impossible by definition. Models -and in our case product models- are an abstraction of the system they represent, reducing concepts of the real world to the necessary minimum. Complexity analysis on this reduced set of conceptual model elements can therefore be conducted down to a numerical level. Metrics for the assessment of software complexity and design quality have been proven in practice within the software industry.

The article gives a brief overview on complexity metrics, how to apply them to product models and possible strategies for keeping model complexity at a reasonable level. Different model standards will be analyzed, distinguishing between logical complexity inherent to the problem domain and formal complexity imposed by the model notation. Due to the metrics presented, views on complexity can be structural, behavioral, quantitative and even cognitive. As a conclusion, a line can be drawn between different aspects of model complexity and potential model acceptance.

Keywords: product model, complexity, metrics, IFC, CityGML

1. INTRODUCTION

A model or a system is said to be complex, if its entire behavior cannot be predicted even if complete information about its parts or components and the interference between them exists (Härtl 2008). The Institute of Electrical and Electronics Engineers IEEE gives the following pragmatic complexity definition in its Standard Computer Dictionary, IEEE Standard 610.12 (1990): "The degree to which a system or component has a design or implementation that is difficult to understand or verify". In contrast, the theory of complexity of theoretic information science is looking for general algorithmic solutions of problems and their classification with respect to the necessary efforts. This publication does not refer to that approach.

In systems theory, complex systems are being characterized by a number of attributes. The complexity of a system is rising with its number of variables, the number of references between them, as well as the functional characteristics of the references (e.g. non-linearity) (Milling 1981). Milicev (2009) distinguishes between logical

complexity of the problem domain the model is based on, and the formal complexity caused by the means used to describe the model (notation, meta levels, etc.).

Compared to the systems they represent, models have a reduced complexity. By applying the modeling mechanisms of abstraction and reduction, only a limited set of attributes of the original system will be reflected by the model. Often a quantification of system complexity becomes possible only by this. Metrics provide an objective measure, independent of the complexity of the problem domain.

2. PROBLEM CONTEXT

Formal models for the representation of man-made or natural objects are commonplace since decades. Existing product models have been worked out further; new domains have introduced their own models. In parallel, information sciences proceeded with the development of modeling techniques. Today, object-oriented modeling can be regarded as commonly accepted and practiced in the area of product data modeling. Formal notation languages like XML and UML are mature, a plethora of tools is at hand to integrate them into a model-based software development process. Professional software production makes increasing use of software engineering methods for the effort prediction of implementations tasks as well as for spotting out areas with higher potential of failure. While information sciences have supplied the theoretic background knowledge for the analysis of software systems, many tools for the practical computer-aided analysis of software systems have been developed. In this article two product model standards are inspected with those tools, complexity and efforts of using CityGML 1.0 and the Industry Foundation Classes version 2x3 are compared. The interpretation of results is based on well-known design principles and their practical application within common system architectures.

The IFC comprises interdisciplinary building information as used throughout its lifecycle and has been further developed throughout its almost thirty year old history. On the other hand, CityGML –being accepted as a standard by the OGC in 2009- is quite juvenile. It is a common information model for the representation of 3D urban objects. It defines the classes and relations for the most relevant topographic objects in cities and regional models with respect to their geometrical, topological, semantical and appearance properties. Included are generalization hierarchies between thematic classes, aggregations, relations between objects, and spatial properties. The underlying problem domains of CityGML and IFC share many concepts (e.g. for geometrical representation, part-of relations, infrastructural elements, and more.). The conceptual overlap makes them ideal candidates to be compared in terms of complexity, although any affinity to the underlying problem domains is not necessary for the assessment of complexity, since metrics provide an objective measure of the formal aspects only.

3. ANALYTICAL METHOD

At first sight, product models do not seem to be a piece of software; with appropriate tools, however, product model schemas can be converted from an XML-Schema-Notation into C# or Java class definitions, holding the identical information in just another notation. Figure 1 shows the equivalence of different notations. In an XML-to-C# transformation XML elements are being transformed into C# classes, XML attributes are transformed into class variables, the inheritance hierarchy as well as any other structural or type related characteristic is maintained. For the product models discussed here, the transformation is lossless. It yields a pure OO data model without any methods. The transformation of an XML schema into the schema of a conventional object-oriented programming language enables the use of software analysis tools, in order to examine model schemas against different metrics. Due to this analytic viewpoint an objective numerical comparison between different product models can be conducted. Problematic areas and improvement potentials can then be

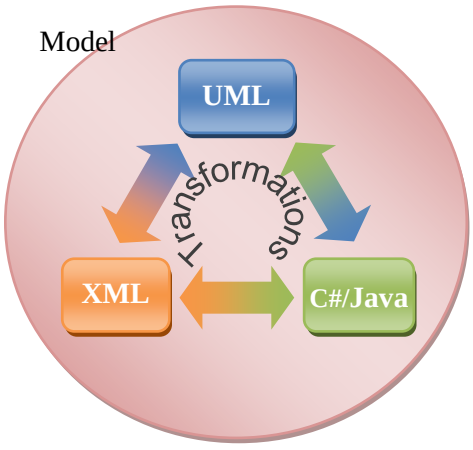

Figure 1: Model in equivalent notations

located on the basis of the statistics. In the context of this paper IFC2x3 and CityGML schema files have been transformed into C# class libraries. The determination of the values of different metrics have been made with commercial analyzers (e.g. Microsoft Visual Studio Ultimate, NDepends) for object-oriented structures. By using the same transformers and analyzers, comparability is ensured.

4. METRICS

A short overview is intended to clarify the approaches of different metrics; for a closer look into the matter the reader is kindly redirected to the bibliography in the appendix, e.g. Halstead (1976), Henry and Kafura (1981) and Shao and Wang (2003). Some commonly used metrics for OO structures (see e.g. Chidamber and Kemerer 1994) yield to values of zero when analyzing pure data models because they take the number of methods of a class or the number of operators and operands (Halstead) into account; that is why they have been omitted in this article.

Not surprisingly, metrics can take into account only those model definition parts that are expressed explicitly. With some product models, parts of the model definition or of common application samples are presented either in implicit or in not computer-readable form. Examples of these are implementers' agreements, data delivery manuals, best practices, generally accepted example data, verbal arrangements and the like. The resulting increase of model complexity cannot to be captured analytically; however, it affects training and application expenditures substantially.

Model Size: Lines of Code (LOC)

The simplest measured variable for the complexity of software is its number of code lines (usually without comment and blank lines). As previously mentioned, formally different, however content-wise equivalent, representational forms for models exist. They can be transferred by means of transformers into one another. For example, the schema of the Industry Foundation Classes (IFC) can be expressed in the formats XML, STEP EXPRESS physical file, as well as class definitions of a suitable programming language (C#, Java). It is worth mentioning that the number of lines, by which the different formats express same model concepts, may differ significantly so that a comparability of the complexity of different models is possible only by using same formal notation, to avoid side effects caused by the eloquence of the underlying format (Kan 2002). The LOC metric is simple, can be measured easily and is quite intuitive. However, it leaves intelligence and the structure of the code unconsidered. For the class libraries of the examined product models the following values have been calculated.

CityGML: 57.814 Lines of Code
IFC: 12.664 Lines of Code

Control Flow Complexity CFC

The cyclomatic complexity (also known as cyclomatic number, program complexity or McCabe complexity) was already introduced 1976 by Thomas McCabe (1976). It is a widely used static software metric independent of the programming language.

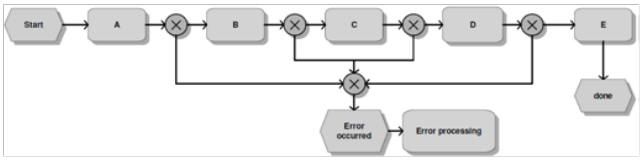

Figure 2: „almost linear" model with CFC=8 (Gruhn 2006)

The idea behind the software metric of McCabe is that starting from a certain degree of complexity the regarded (partial-) system becomes hardly understandable for humans. McCabes cyclomatic number v(G), represents the complexity of the control flow (function, procedure or code section). v(G) is the number of conditional branches of the flow chart. Simply put, the complexity measure of McCabe equals the number of binary branches in the code logic plus 1. It gives the maximum number of linear independent paths through the program (McCabe et al. 1989).

For a program, containing merely sequential instructions, v(G) amounts to 1. The higher the cyclomatic number, all the more paths through the function are possible and the more difficult it is to be understood and tested. In the control flow graph, where the instructions are represented as nodes and the control flow between the instructions is represented as edges, the McCabe number of M is defined as

$$M = e - n + p$$

e: amount of edges in the graph
n: amount of nodes in the graph
p: amount of single control flow graphs (one graph per function / procedure)

More details on the computation, as well as examples see (Cullman et al. 2007). Within pure data models, only class variables contribute to the value of e, but we will see in the sample application below that CFC is valuable for the assessment of structural design decisions and can show the significant side effects on applications.

Weaknesses of CFC

A disadvantage of this metric is that bare counting of the number of possible decisions in a model shows only small information about its structure. For example the models in Figure 2 and Figure 3 contain the same number of decisions (CFC = 8), however, the almost linear model in Figure 2 is obviously easier to understand. Furthermore, CFC does not detect the impact on complexity caused by references to abstract types, replaced by concrete derived type instances at runtime. This problem is further discussed in the sample below.

Advantages of CFC

- Suitable for the estimation of maintenance efforts
- Applicable as quality metric for the assessment of different software designs
- Usable in early design stages
- Measures minimum effort and most effective testing areas
- Can be consulted for the guidance of test processes during the development phase
- Easy to use

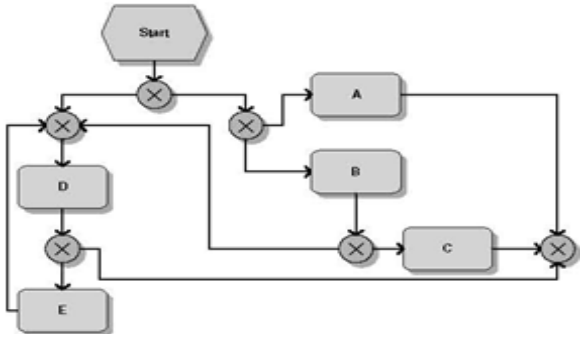

Figure 3: „poorly structured“ model with CFC=8 (Gruhn 2006)

For each examined *complete*[1] product model the following CFC values have been calculated.

CityGML: v(G) = 34.757
IFC: v(G) = 8.741

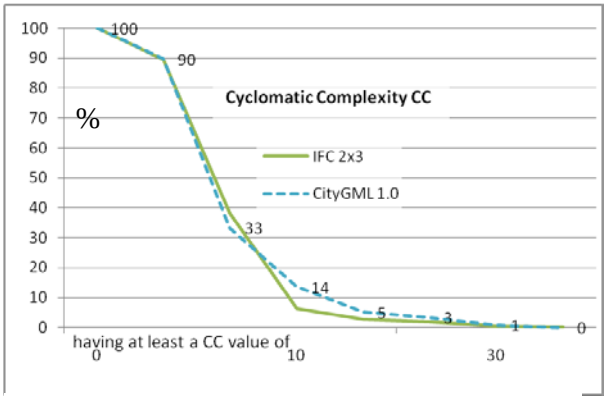

Figure 4: Distribution of the cyclomatic complexity of IFC and CityGML

Differentiation by individual elements results into the diagram in Figure 4. It shows the percentage of elements of the respective product model that exhibit a cyclomatic complexity of the value of the x axis or more. Since it is a matter of pure data models, here the control flow is primarily caused by complex entities and references between them. For the purpose

[1] Without respect to namespaces

of comparability all percentage values in the following diagrams have been normalized to the number of element types of the respective model. It becomes evident that the distribution of the cyclomatic complexity does not exhibit considerable differences between IFC and CityGML, i.e. the proportional portion of elements with low, middle and high McCabe complexity is approximately alike.

This picture changes fundamentally, when looking at the impact of logically same but formally different modeling variants. It will be shown that the complexity of applications is strongly affected.

Influence of Model Structure Design on the Control Flow Complexity of Applications

The metrics introduced so far evaluate program and data structures; they do not measure the effects of different modeling variants on runtime behavior and implementation expenditure within applications. Therefore complexity effects, resulting from topology specifications of the model schema, are not been taken into account so far. Thus, the complexity difference between IFC and CityGML does not appear significant by some of the metrics mentioned. Only an object-instance-based investigation shows the impacts and subsequent effects on application complexity. Therefore, we will have to have a closer look on two alternative modeling designs of the assignment of properties to elements. In alternative 1 (Figure 5) the element owns a reference to its properties in a part-of relationship (the object 'knows' its attributes), a quite usual object-oriented design. In alternative 2 (Figure 6) things are reversed. Here, the properties know the object(s) they belong to. This modeling approach is quite common in the relational paradigm, sometimes also referred to as 'loose coupling'.

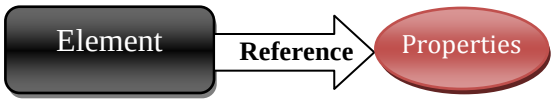

Figure 5: Assigning properties to objects (object-oriented model)

If a part-of relationship is designed as a loosely coupled reference (Figure 6), further operations for the resolution of the reference, including the examination of the existence of the reference target, are necessary, because the attributed object is unaware of its own characteristics. It is assigned using a relation between the object and its properties. The attributes are being assigned by 'backward' reference in a relational database fashion. In an object-oriented aggregation design no further operations are necessary (Figure 5), causing a lower complexity value for the same structural solution.

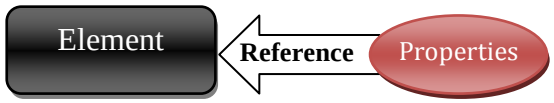

Figure 6: assigning properties to object using backward references (relational model)

A comparative computation of the cyclomatic number of the relational alternative according to Figure 5 and the object-oriented design alternative according to Figure 5 is conducted to clarify this.

A metric that takes into account the complexity issues caused by backward references and abstract target classes would be desirable. Unfortunately, even metrics assumed to be adequate on first sight, like the CBO metric (Coupling Between Object Classes), do not detect this. The following sample should help to clarify the problem.

Sample Application

Sample use case: The rooms belonging to a storey are to be determined.

In case of the object-oriented model (Figure 5) the list of rooms is embedded into the floor model and can therefore be directly accessed (→ CityGML). Also, a deletion of the owning parent object (e.g. floor) leads to a deletion of the related rooms, showing the appropriate behavior of a consistently modeled „part-of"- relation. In case of the relational model the associated rooms have to be determined in subsequent operations shown in Figure 7. In this case, a deletion of a floor object does not necessarily lead to the deletion of the associated rooms and can be considered modeled incorrectly.

Maybe it is because of IFC's long history that there is an affinity to relational data base systems. In fact, it is an indication of poor compliance with recent object-oriented design principles in favor of relational implementation variants. Object-oriented applications working with pseudo object-oriented concepts (but relational by its nature) suffer from the burden of increased complexity.
In the IFC, the floor object does not 'know' its room objects, because the reference is pointing from the relation object to the floor object. In addition, floors and rooms can be linked by an arbitrary number of relation objects. Besides references to objects of type floor (IfcBuildingStorey) also references to objects of type IfcSite, IfcBuilding and IfcSpace (room) are formally valid, because they share the same (abstract) base type in the type hierarchy. In the child direction, references can point to any IfcProduct-derived types (where IfcSpace is one of). Partly caused by this possible type spectrum, an application that wants to display the rooms of a storey has to implement the following control flow while using IFC.

- Select all relation objects (IfcRelContainedInSpatialStructure) that point to objects of type storey in the parent direction and to type room in child direction.
- Select relations that point to the individual storey requested, ignore all others
- Select children of type room, ignore all others
- Based on identifiers found, get the information of the referenced object

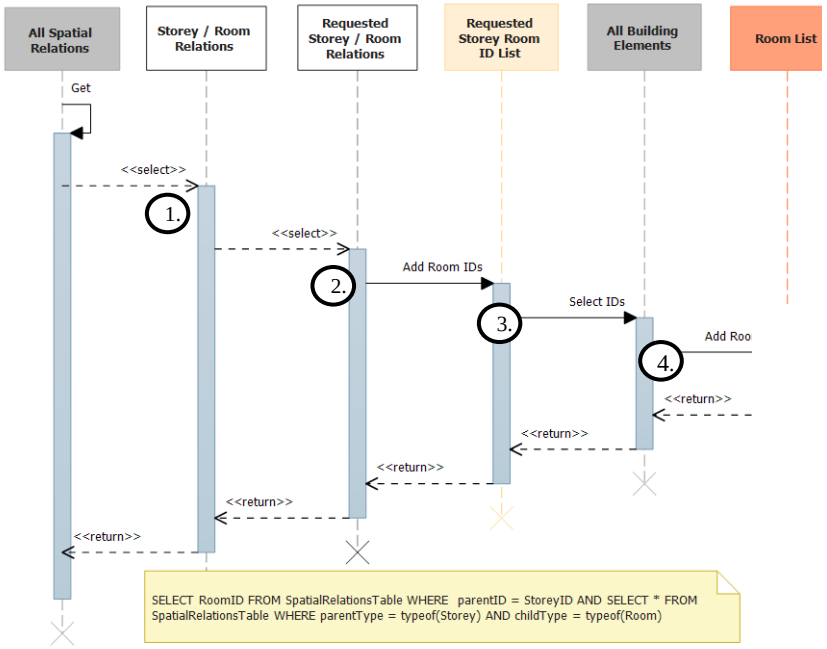

Figure 7: Stepwise determination of a storey room list from a relationally structured element set

The UML sequence diagram in Figure 7 tries to clarify this. In real world implementations additional efforts may be potentially hidden behind declarative database query expressions[2] (OCL, SQL); however, they have to be carried out in any case and increase complexity in the example by the factor 3, which has to be evaluated in a tiny sample application, implemented for both scenarios. The calculated complexity numbers are

Object-oriented (CityGML): v(g) = 4
Relational (IFC): v(g) = 12

Consequences

Since the presented modeling designs are widely used in the respective product models, the complexity resulting from this is of major significance. Furthermore, the composition relation cannot be modeled appropriately with IFC elements. That is why whole/part-relations cannot be represented correctly. In a relational data base system an unresolved reference may occur after a deletion of the floor data record, but the absence of the reference target becomes immediately obvious because the underlying relational database management system takes care of referential integrity. With an object-oriented system the loose coupling of parent and child objects is not suitable for the modeling of relations, where the lifetime of the objects depends directly on each other. Otherwise it does not implicitly lead to the

[2] most likely not analyzed by metric tools

deletion of child objects after the parent object has been deleted, therefore additional business logic has to be implemented to accomplish this, making the code more complex and error-prone. The frequently stated argument that an automation concerning the deletion of dependent elements would not be desirable because of a possible re-use of those objects in other contexts, is not sufficient. This form of lifetime management of objects requires an explicit modeling and should not be left to be handled by application logic.

Depth of Inheritance and Class Coupling

Both models in Figure 2 and Figure 3 have a CFC metric value of 8, although the model in Figure 2 appears to be less complex than the one in Figure 3. The first shows an almost linear control flow whereas the second shows some nested XOR branches or joins respectively.

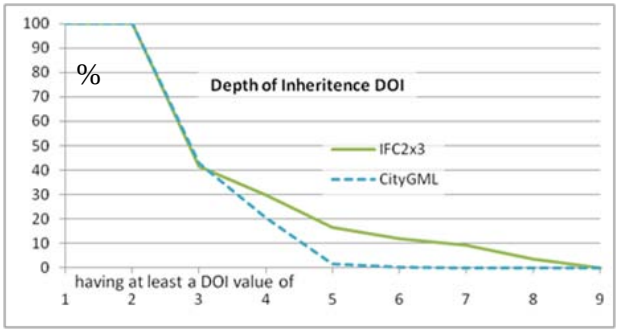

Figure 8: Accumulated distribution of depth of inheritance

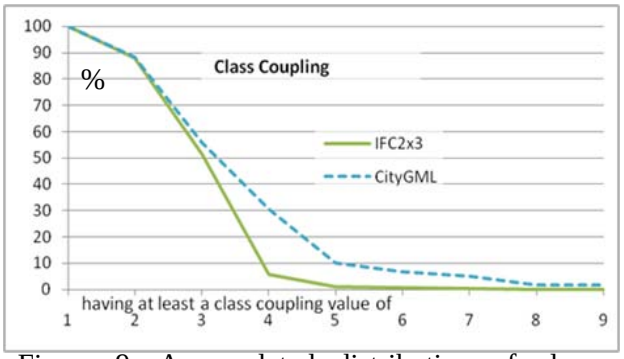

Figure 9: Accumulated distribution of class coupling

Research results show that the metrics “maximum nesting depth” and “average nesting depth” are useful factors of influence for the determination of the total complexity of a model. A larger nesting depth implies higher complexity. According to Schroeder (1984) both metrics have strong influence on other structure-related complexity metrics.

The depth of inheritance metric can be applied to the type hierarchies of product model element types, as well as to aggregate structures: Both, inheritance levels and aggregation relations lead to an increase of the nesting depth.

While IFC is characterized by a steep type hierarchy, spanning up to nine levels (16% of all types have five or more inheritance levels), hierarchy level remains basically below five in CityGML (Figure 8). In contrast, the CityGML modeling concept facilitates aggregation in favor of derivation, which leads to higher values of the class coupling metric (Figure 9). As a consequence of the predominant use of the relational model in the IFC schema, (Figure 5) the class coupling is clearly smaller within IFC than within CityGML.

With the same logical (=conceptual) complexity a deeper derivative hierarchy (=formal complexity) leads to a higher total complexity. The optimal balance between logical and formal complexity, where a formal structuring supports the logical structuring, has to be the definite goal of any design consideration.

Cognitive Complexity Metrics

In section ‘Control Flow Complexity’, two examples have been presented to illustrate weaknesses of the CFC metric. The metric 'Depth of Inheritance and Class Coupling' can be useful in cases similar to Figure 2 and Figure 3 and may be applied as supplement to the CFC metric.

As an indicator for the testing effort of structures similar to Figure 3, where parallel paths are potentially being processed concurrently, the CFC metric is ideally suited. However, this metric is less suitable as a measure for the difficulty to understand a model (Gruhn 2006). The number of control flow paths between an OR branch and its related conjunction does not have large influence on the effort to understand this control structure.

Regardless of the number of paths between bypass and combination, a human reader will always realize the model as one single structure.

Shao and Wang (2003) define the cognitive weight as a measure for the effort to understand a software sequence. Based on empirical studies, they define cognitive weights for fundamental control structures. The cognitive weight of a software component is defined as the sum of all cognitive weights of its control structures. The idea is to see the underlying control structures as patterns that can be understood as a whole by the reader.

A similar idea has also been published by Gustafsson (2000). Here, the attempt has been undertaken to find well-known (software-) architecture patterns automatically. It implies the idea that a high maturity of those well-documented patterns contributes to the improvement of code quality, brevity and ease of maintenance. This assessment should be adopted only with care, however. Architecture patterns are useful only if applied correctly by experienced engineers. An excessive use of architecture patterns is by no means an indication of good code quality. This can be transferred to modeling processes as well. Similarly, for the expression of model elements often different alternatives with varying expressive power and quality are at hand, but best practices for distinct model types have been carved out (e.g. loose coupling vs. embedding, CFG vs. B-Rep, parametric vs. fixed geometry, etc.). While using Gustafsson's approach, a deep knowledge about the patterns used and their correct employment is necessary. In Gustafsson (2000) however, not only the employment of "good" architecture patterns is discussed, but also the recognition of so-called anti-patterns. Those are quite often selected problem solutions, infamously known for their frequently negative consequences. The occurrence of such anti-patterns can be rated as an indication of bad programming style. It would be quite appealing to detect such indicators of bad modeling style in model schemas and instances as well. This could support interoperability between applications significantly and reduce errors and losses alongside the entire model lifecycle.

Gruhn presents an adaptation of the original cognitive weights mentioned above to the requirements of business process modeling in Gruhn (2006). In the same sense, cognitive weights for common structures found in product models should be found. Eventually, some of the control structures mentioned by Shao and Wang (2003) turn out to be of minor relevance for product models, while for the evaluation of product model complexity additional ones have to be taken into consideration.

Assumptions for Cognitive Weights of Control Structures in Product Models	**W'_i**
Inheritance Level (analogous sequence)	1
class coupling (analogous function call)	2
Inheritance (per hierarchy level)	2
Association relation (already covered by class coupling)	0
Relational reference	3

Table 1: Assumptions for Cognitive Weights

Empirical determinations of cognitive weights of model structures would exceed the limits of this contribution and have to be left for future research, up to the experimental validation of cognitive weights.

For a first rough calculation the assumptions W'_i of Table 1 have been made. They lead to values of approximately 2000 for each of the CityGML namespaces and about 6000 for the (one and only) IFC namespace. The significantly higher cognitive complexity of IFC in contrast to CityGML is clearly visible. Also here a factor of about three is to be observed. Complexity can be adjusted to the problem more easily with CityGML. You need to include only those namespaces that correspond to the problem, thus being faced to objects of those namespaces only, whereas applications using IFC have no formal means of narrowing the scope, because any IFC object may occur e.g. when loading an IFC file.

Modularity of the Model: The Fan-in / Fan-out-Metric

The partitioning of models into modular sub-models does not contribute to the easier understanding of the entire model, it leads however - with meaningful implementation of the modularity- to smaller, re-usable models. As an example, applications can selectively implement two or three CityGML namespaces and leave others unconsidered, thus, having a means to scale complexity.

Henry and Kafura (1981) developed a metric for the evaluation of the structure of modularized software systems. It calculates the fan-in (comparable to entry load in electrical engineering) and fan-out (output load) for each module. The fan-in value corresponds to the amount of all modules calling a given module and fan-out to the

number of modules called by a given module. Small sub-modules for simple tasks (utilities, catalogs, default properties, etc.) used by many other modules usually have high fan-in values. Modules within a higher abstraction level in the system architecture usually have a higher fan-out value. A high fan-in and a high fan-out value of a module may indicate that a re-design would be reasonable. Henry and Kafura propose the metric

$$((\text{fan-in}) \cdot (\text{fan-out}))^2$$

as a measure for this type of structural complexity (Sommerville (1992)).

Advantages of the metric of Henry und Kafura

- considers data-driven applications
- can be used for development preparation in the design phase

Disadvantages of the metric of Henry und Kafura

- complexity values of zero may occur (modules without external interaction)

The transferability of these metric on product models is not obvious at first sight, since models rarely offer modularity mechanisms. As a first approach, a high fan-in and fan-out value can be expected depending on the model size (types, not instances) - with the implications specified above. This is in no way a surprise, because such a monolithical structure is one of the manifestations of the so-called software crisis. With the generated class libraries of IFC and CityGML, we pursue the following approach. Each public type of the schema is a potential entry point for other modules to call in. Therefore, the number of public types corresponds to the fan-in of the respective product model. The fan-out accounts to the fact that a schema is not contained in itself, but at least refers to an external base schema. In this calculation, the fan-out value is therefore set to 1. So, the formula of Henry and Kafura simplifies to $(\text{fan-in})^2$. For this comparison the square is also avoided. The IFC namespace yields a value of about 8700 whereas the 11 CityGML namespaces have a value of about 3200 each. The modularization into namespaces in CityGML turns out to be an efficient instrument for the reduction of complexity.

5. CONCLUSION

The origins of the IFC XML schema from the relational database world is clearly noticeable. Instead of the aggregation mechanisms commonly used in the object-oriented world, relationship objects of the relational modeling paradigm like cross tables are frequently used. This makes object-oriented processing of topologies (object characteristics, building structures, etc.) on the application side more difficult. In contrast to CityGML, composition relations cannot be modeled, here IFC apparently relies on the referential integrity mechanisms of relational database engines. Despite the clearly higher amount of types in CityGML the user is confronted with a clearly lower complexity than within IFC. This is to be owed to the consistent use of namespaces for schema structuring in CityGML which encapsulates complexity in a scalable fashion. An equivalent conversion for structures like the IFC Views (coordination view, FM view, etc.) defined for software certification purposes could create a remedy here. Today, those views are used only informally. An improved format could promote a widespread beneficial use in applications.

At present the use of CityGML requires less effort compared to IFC while exposing comparable total complexity. Due to the consistent modularity of the CityGML schema the complexity can be scaled according to the requirements of the sub-domain. The training period into the schema is less time-consuming due to flatter type hierarchies. While elements resembling structures of the relational database world can be found in IFC, CityGML has a pure object-oriented model structure.

It has been shown, that a two to three times higher complexity value than with CityGML can be expected with IFC while using comparable constructs. This leads to a significantly higher effort for design, testing and

maintenance. Looking at the long history of IFC, a revision of the XML IFC schema appears advisable. This would open up this valuable standard to a broader application field, which would otherwise probably be introduced into other standards.

REFERENCES

Chidamber, S.; Kemerer, C.: A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol.20, No.6,1994

Cullmann, X-N.; Lambertz, K. 2007: Komplexität und Qualität von Software, *MSCoder*. www.mscoder.org/de.

Grady, R.B. 1994. Successfully applying software metrics. Computer 27: 18–25.

Gruhn, Volker; Laue, R. 2006. Complexity Metrics for Business Process Models, Applied Telematics / e-Business Computer Science Faculty, University of Leipzig, in: 9th International Conference on Business Information Systems, BIS 2006, Klagenfurt, Austria.

Gustafsson, J. 2000. Metrics calculation in MAISA.

Härtl, H. 2008. Implizite Informationen: Sprachliche Ökonomie und interpretative Komplexität bei Verben (studia grammatica 68). Berlin: Akademie-Verlag, ISBN 3-05-004502-7.

Halstead, Maurice H. 1976. Elements of Software Science.

Henry, S.; Kafura, K. 1981. Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5): 510–518.

IEEE Standard 610.12 (1990) IEEE Standard 610.12-1990. Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.

Kan, S.H. 2002. Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

McCabe, T.J. 1976. A Complexity Measure. in: IEEE Transactions on Software Engineering, Vol. SE-2: 308-320.

McCabe, Thomas J; Butler, Charles W. 1989. Design complexity measurement and testing, Communications of the ACM, Volume 32, Issue 12 (December 1989, Pages: 1415 – 1425, ISSN: 0001-0782.

Milling, P. 1981. Systemtheoretische Grundlagen zur Planung der Unternehmenspolitik. Berlin: Duncker & Humblot, ISBN 3-428-04931-4.

Milicev, D. 2009. Model-Driven Development with Executable UML, Wiley (Wrox), ISBN 978-0-470-48163-9

Schroeder, A. 1984. Integrated program measurement and documentation tools. In: ICSE ’84: Proceedings of the 7th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press: 304–313

Shao, J.; Wang, Y. 2003. A new measure of software complexity based on cognitive weights. IEEE Canadian Journal of Electrical and Computer Engineering: 69–74.

Sommerville, I. 1992. Software Engineering Addison Wesley Publishing Company, Workingham, England.