Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2006 Proceedings

Americas Conference on Information Systems (AMCIS)

December 2006

On the Transition from Computation Independent to Platform Independent Models

Milan Karow
University of Muenster

Andreas Gehlert
Technische Universitaet Dresden

Jörg Becker
University of Muenster

Werner Esswein
Technische Universitaet Dresden

On the Transition from Computation Independent to Platform Independent Models

Milan Karow
European Research Center for Information Systems (ERCIS)
Department of Information Management, University of Muenster
Leonardo-Campus 3, D-48149 Muenster
milan.karow@ercis.de

Andreas Gehlert
Technische Universitaet Dresden
Department of Business Management and Economics
Chair of Information Systems, esp. Systems Development
D-01062 Dresden
gehlert@wise.wiwi.tu-dresden.de

Jörg Becker
European Research Center for Information Systems (ERCIS)
Department of Information Management, University of Muenster
Leonardo-Campus 3, D-48149 Muenster
becker@ercis.de

Werner Esswein
Technische Universitaet Dresden
Department of Business Management and Economics
Chair of Information Systems, esp. Systems Development
D-01062 Dresden
esswein@wise.wiwi.tu-dresden.de

ABSTRACT

The Model Driven Architecture (MDA) describes software development based on models on different levels of abstraction. The development process is outlined as a sequence of model transformations which add specific details to the software models with each subsequent step. The OMG MDA guide refers to the computation independent model (CIM) as the highest level of abstraction. It can therefore be considered as the designated means for analysis in MDA software development projects. However, this model type is disregarded by most MDA methodologies and tools, which start their transformation process on the level of platform independent software models (PIM). This paper investigates the role and nature of the CIM, especially focussing on development processes for management information systems. We argue that an automatic translation from CIM to PIM is theoretically not feasible. To provide evidence for our claim, we evaluate existing transformational approaches that address the transition from models as real world perceptions to software designs. Lastly, we support our arguments by extending the traditional software development view by decision theory aspects during the design phase.

Keywords

Model Driven Architecture, model transformation, conceptual modelling, analysis, design, Computation Independent Model, Platform Independent Model

INTRODUCTION

In the field of software engineering, modelling has become a major technique to deal with increasing complexity and risk of software development projects. Models are used to analyse, design and document software artefacts in different stages of the software life cycle. They represent an important instrument for the communication between different stakeholders within a development process (Génova, Valiente and Nubiola 2005).

The MDA approach has been established as a standard framework for model driven software development. Various methods and supporting CASE tools are existent and in use for large scale projects. The realisation of MDA, however, is addressed by software engineers and tool manufacturers in very dissimilar ways. While there are numerous approaches for model

transformation between platform independent and platform specific models (see Czarnecki, K. and Helsen 2003 for a classification), the computation independent model has been widely excluded from current research.

Given the rising importance of business process management and conceptual modelling (Weber 2003), it seems obvious that a transition between computation independent and platform independent models can be considered as being critical for the development of high-quality business software.

The main question investigated in this paper is how the information of such business models influences the design of the software system. This raises the question which information is expected to be modelled in computation independent models and whether this information can be processed in a formal fashion to generate an application system design. We will show that the translation from analysis to design is a highly creative task so that an automation is generally not feasible. Within the analysis of different model types used in software development processes we show that analysis and design models are fundamentally different. Additionally, we analyse whether existing transformation approaches provide an adequate formalisation for the transition of analysis to design models. We conclude this paper by applying decision theory elements to this problem.

MODEL DRIVEN SOFTWARE DEVELOPMENT

A common distinction of model types in software engineering is to categorise them depending on their utilisation in a certain phase of the software life cycle. This leads to stereotypes such as analysis models, design models and - regarding code as a textual form of a model - implementation models (Sommerville 2001).

The term *analysis*, however, is imprecise, since different kinds of models fall into the software development phase referred to as "analysis", but not all these model types have an *analytical* character. Genova et al. distinguish two fundamentally different types of analysis models depending on their *semiotic reference* (Génova et al. 2005, see also figure 1). From this viewpoint, we distinguish between two analysis model types and for each type between two subtypes:

1. **Organisational model**: The first type of model refers to a perception of the real world. The intention of these models is a better understanding of the business environment (*Universe of Discourse, UoD*) in which an application system will be embedded. To construct such a system in a systematic manner and to exploit its full potential, the analysis phase can be split into two sub-phases:
    a. **Actual state organisational model**: The actual state model describes the organisation as it is when the project is initiated. Consequently, it specifies and structures the problem, which is to be solved within the project.
    b. **Target state organisational model**: The target state model is developed during the business process (re-)engineering activities and specifies the solution to the problem represented in the actual state organisational model. The target state organisational model cannot fully be declared as being computational independent because it is constructed on the assumption that the application system in development already exists.
2. **Requirements model:** The second type of analysis models represents the requirements of the software system. These *requirement models* already refer to the formal software system and have, therefore, a *synthetic* nature. Most interestingly, requirements models are not regarded as core models in the Model Driven Architecture (Génova et al. 2005). In contrast to *design models*, which refer to the software system as well, requirements models represent a different kind of abstraction. While requirement models represent what the system should do (external view), design models depict how the system is operating (internal view).

    Two different kinds of requirement models can be distinguished:
    a. **Functional requirements model:** Functional requirements describe the functionality of the software system. As the target state organisational model was already constructed with the future software system in mind, it is the most valuable source to derive functional requirements. To do so the functions, which should be supported by the software system, should be selected from the organisational model. This set of functions is the starting point to evolve the functional requirements model.
    b. **Non-Functional requirements model**: Non-functional requirements describe additional constraints of the software system including performance, security, scalability, maintainability, etc. issues. This type of software requirements cannot be derived from any of the afore-mentioned models and must be elicited directly from the stakeholders.

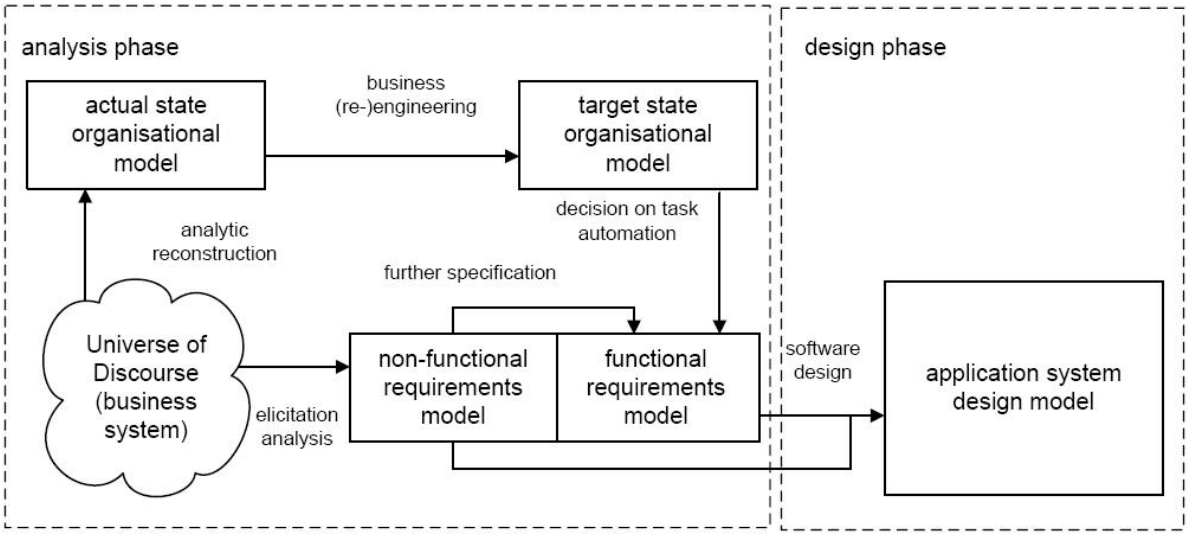

Figure 1. Model Types in Business Systems Engineering

The introduced model types strongly differ in the languages utilised for their construction. These languages can be classified into three different types (Fraser, Kumar and Vaishnavi 1994):

- **Formal languages:** Formal modelling languages have a strictly defined syntax. The restricted set of signs and rules to combine these signs is defined in the meta-model of the respective language. Formal languages have no material semantics, i.e. their symbols do not relate to real world concepts. The *formal semantics* of a language describe precise rules on how to transform sentences of this language between different formal systems, relying on mathematical principles. Examples are programming languages.
- **Natural languages:** Natural languages do neither have a defined syntax nor semantics. These languages evolved naturally in their specific community. Syntax and semantics of natural languages are determined by a common consensus and contain multitude redundancies and inaccuracies (Kamlah and Lorenzen 1984). The special language of a business domain (e. g. a particular organisation or a specific industry) is also regarded as being natural.
- **Semi-formal languages:** Semi-formal languages have a formal component, which precisely describes the set and combination rules of graphical elements (syntax) but also a material component (semantics). The semantics of semi-formal models are composed of the concepts of the language (language elements) as well as the natural language used to name instances of these elements. Thus semi-formal models can only be interpreted by model users that are capable to comprehend the special (natural) language of the business domain (Pfeiffer and Gehlert 2005).

Organisational models are typically constructed by utilising a semi-formal language such as event-driven process chains (ARIS-EPC), entity relationship models (ERM) and the like. Similar languages are used to describe the functional requirements. Non-functional requirements, however, are usually described in textual form using a natural language. Design models are either specified with a semi-formal language (e. g. the Unified Modeling Language UML) or by using a formal language like Petri Nets. Consequently, the transition from analysis to design is a translation from natural and semi-formal language artefacts to formal language artefacts; in other words, a transition from semantics to syntax.

MODEL TRANSFORMATION APPROACHES

This section will investigate some existing approaches for the transformation of organisational models into software design models. Therefore an evaluation framework is outlined, which guides the analysis on the several model transformation technologies. The properties taken into consideration are derived from basic quality requirements for the model types described above.

Transformation Evaluation Framework

A prominent quality criterion for organisational models is its *completeness* (Moody and Shanks 2003). This is especially relevant for the goal of automated transitions from analysis to design. Models that refer to real world systems, however, cannot be *proven* to be complete since real world phenomena are always perceived by individuals and perceptions cannot be considered as being objectively correct or false (Schütte 1999).

Models of a real world domain can only be evaluated and agreed on being *sufficient* and *useful*. This *consensus* is dependent on the purpose of the respective model. A modelling and transformation methodology ought to support the process of **consensus finding** (Re1). Consensus finding includes the utilisation of modelling languages comprehensible for *all* stakeholders during the respective development phase (Schütte 1999).

Furthermore, all models and modelling languages must strictly distinguish between concepts regarding a real world domain and concepts describing software system properties (Kaindl 1999). This criterion will be referred to as **semiotic integrity** (Re2).

Quality requirements for design models are strongly interrelated with general software quality requirements (Bass, Clements and Kazman 2003). A central argument for evaluating software designs is the **handling of non-functional requirements** (Re3) elicited in the analysis phase. These requirements determine the *architecture* of a software system. Functional requirements can be fulfilled by *almost any system architecture*, even by one monolithic function (Kleppe and Warmer 2003). Consequently, architectural and structural issues are not determined by the system design until non-functional requirements are to be met.

Turning back to the MDA perspective the model transformation process is important to the whole software life cycle. The **transformation applicability** (Re4) describe the set of factors including transformation control (parameters etc.), flexibility of the transformation (limitation on particular system types), traceability of generated artefacts, roundtrip engineering (protection of manual changes) and the coverage of the transformation.

In table 1 the criteria used to evaluate different transformation approaches are summarised. For our analysis, we used transformation approaches introduced in the first half of the last decade including the ontological state machine by Wand and Weber, the Object-Oriented Analysis OOA (Coad and Yourdon) and the Object Modeling Technique OMT (Rumbaugh et al.) approaches as well as the Recursive Design (Shlaer and Mellor). The reason for the evaluation of these rather "out-of-date" techniques is their explicit emphasis on model transformation from analysis to design, which has been abandoned by most of today's software development processes. Currently applied methodologies (such as the Unified Process) tend to accentuate the *creative nature* of the design process as a complex and experience-driven task (Jacobson, Booch and Rumbaugh 1999). With MDA on the rise however, the transformational view on software development regains significance and is, therefore, a rewarding object of research.

Criterion	Description
Re1: consensus finding	Evaluates how the transformation approach supports the communication between the different stakeholders of the project.
Re2: semiotic integrity	Evaluates whether the transformation approach distinguishes between real world concepts and concepts of the formal software domain.
Re3: handling of non-functional requirements	Evaluates whether the transformation approach includes techniques to analyse non-functional requirements.
Re4: transformation applicability	Evaluates the properties of the transformation approach itself including, control, flexibility, traceability, roundtrip engineering and coverage.

Table 1. Evaluation Criteria

The Ontological State Machine

Wand and Weber introduced the approach of the ontological state machine and utilised it for information system development (Wand 1989, Wand and Weber 1989). According to their theory, information systems can be regarded as being direct representations of real world systems. The approach describes a phase model covering the phases analysis, design and

implementation. The purpose of the *analysis phase* is to build a *formal model* of the real world system. This analysis model is expected to be a *faithful representation* of the UoD. The description of the real world is realised complying with the ontology of Bunge (Bunge 1977, Re1 not applicable because of a realist world view). In this view a system is seen as a triple of *state space*, *system law* and the *set of events*.

In the *design phase*, the state model of the real system is transformed *homomorphically* into a system design (Wand and Weber 1995). The part of reality modelled in the analysis phase represents the UoD. An information system and its respective environment are coupled by events so that events in the environment are reflected within the information system. Thus the information system's state is always synchronised with the state of the real world system (Weber 1997). For implementation purposes, the ontological elements are simply mapped to technological counterparts: events and states represent data, system laws are expressed as processes.

The approach, however, does not take any non-functional requirements into consideration (violation of Re3). Technological or economical restrictions of software development as well as user aspects are completely and consciously ignored. The information systems structure and architecture is not discussed at all. Consequently, the system's decomposition is solely determined by the formal requirements expressed in the good decomposition model (Wand and Weber 1995).

The analysis model in this approach can be classified as the target organisational model since it describes reality at information system's runtime. The model type referred to as the system design, however, has more in common with a *functional requirements specification*. It dominantly describes system behaviour from an outside perspective and does not address the technological solution. Furthermore, implementational restrictions are not considered at all, which can be regarded as being the most important function of the design phase (very limited support for Re4).

Another significant concern is the question, whether a business environment can be fully described using a formal language since it most likely hinders the consensus building process. The requirement Re2 is not applicable for the approach since the software system is seen as a true representation of the UoD. In this view there is no difference between real world and software system concepts.

Object Oriented Transformation Approaches

Object oriented modelling techniques emerged from advances in the field of programming languages. The *object* concept has been implemented firstly by programming languages like Smalltalk-80 (Capretz 2003). The notion of the term object in OOP was, however, not related to real world entities but to formal and abstract concepts of respective programming languages. With the introduction of object oriented analysis techniques, a relation between modelled objects and entities of reality has been adopted (Meyer 1988). The main advantage propagated by many authors was the utilisation of one and the same concept throughout the whole development process and thereby seamlessly integrating its different phases (Kaindl 1999).

OOA and OMT

Popular techniques were the Object-oriented Analysis (OOA) by Coad and Yourdon (Coad and Yourdon 1991a, Coad and Yourdon 1991b), as well as the Object Modelling Technique (OMT) by Rumbaugh et al. (Rumbaugh, Blaha, Premerlani, Eddy and Lorensen 1991). Both techniques adopted the object oriented paradigm for creating conceptual models in the analysis phase. Furthermore, both approaches focus on a structural view, using class and object diagrams to visualise association, aggregation and generalization relationships between objects and classes.

The transformation approach of the two methodologies directly transfers the resulting models of the analysis phase into the design. OOA models are mapped into the problem domain component, which represents a particular partition of the Object-oriented Design (OOD). The remaining three partitions deal with issues like data persistence and retrieval, user interface and task management. Although these additional partitions are critical for the attainment of non-functional requirements, they are not influenced by the results of the analysis phase.

The OMT does not explicitly distinguish between analysis and design models. The object models created during the analysis represent the starting point of the design phase. In the design phase these models are extended by implementational details. OMT design has two sub-phases: system design and object design. During the system design phase, the architecture of the software system is outlined; the object design phase covers the detailed design. Analysis results only reappear in the detailed design and, therefore, do not influence the architecture of the system.

Consequently, non-functional requirements are neither respected in the analysis nor do they have a defined influence on the design task. The compliance of the application system with respective user requirements is therefore strongly dependent on the designer's experience (no support for Re3).

Both approaches lack a consistent understanding of the term "object" as objects of the organisational models representing real world concepts are not distinguished from objects representing software artefacts in the design phase (low support for Re2). The *finding of a consensus* between software and domain experts is obstructed by the utilisation of IT-specific concepts during the conceptual analysis phase (Ortner and Schienmann 1996, low support for Re1). This threatens the semantic consistence of the analysis models and leads to precipitated design influences on the modelling task (Kaindl 1999, Parsons and Wand 1997).

Recursive Design

The *Recursive Design* (RD) of Shlaer and Mellor (Shlaer and Mellor 1992, Shlaer and Mellor 1997) is based on yet another object oriented modelling methodology (also called OOA) which resembles the techniques discussed above in many ways. A noticeable difference is the fact, that RD can be applied to the development of applications which eventually are not implemented in an object-oriented programming language.

The central concept of the approach is the *domain*, which represents a real or abstract UoD inhabited by objects with a specific behaviour. These domains can be analysed and modelled using the OOA methodology. Not only is the *application domain*, which is an abstraction of a part of the real world, modelled that way but also the systems *architecture*. The architecture defines rules and mechanisms that are applied to all elements of the software system. The term architecture, however, is used differently in RD. An architecture does not describe the general structure of complex system components but a *framework* of design elements thoroughly designed to meet a particular behaviour. This system of software elements can be constructed *independently* from a particular application by analysing the general characterization of the system and then designing the architecture. However, with its *archetypes* (code templates) related to the modelled concepts, the architectural model represents a *design language* and cannot be regarded as a particular design. The actual design is derived by type-mappings of the application domain model to architectural concepts (see Figure 2).

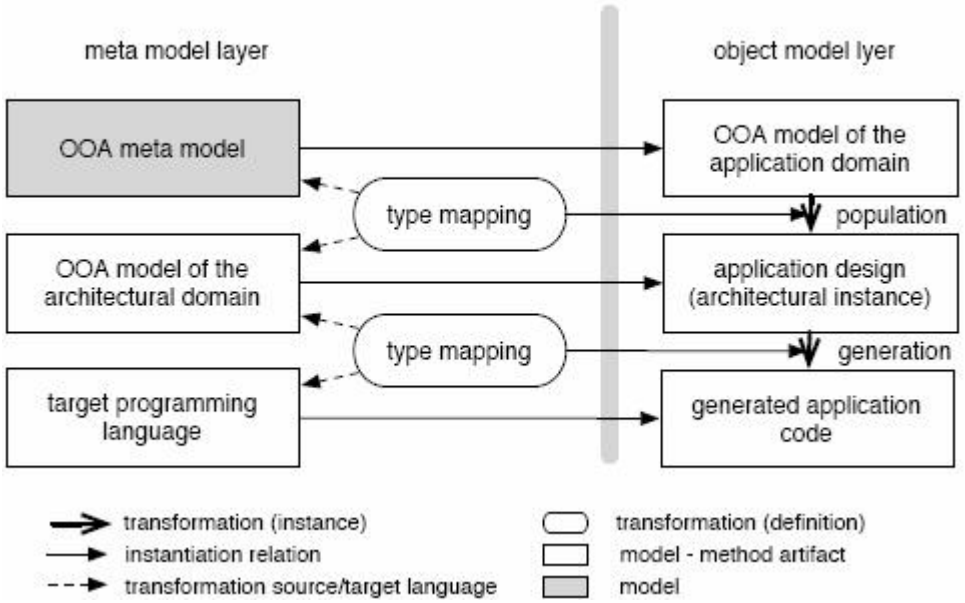

Figure 2. Method Artefact Relations in Recursive Design

As described above, RD lacks a consistent object definition (violation of Re2; weak support for Re1). The focus on behaviour throughout the method resembles it with the ontological state machine, yet RD includes tasks for the realisation of the system in software. The limitation on dynamics restricts its use to real-time applications, where RD has been applied successfully (Kozlowski, Carey, Maguire, Whitehouse, Witzig and Sorensen 1995), but makes it less useful for business applications within their function oriented environment (limited support for Re4). In contrast to the methods formerly discussed, non-functional requirements are explicitly taken into consideration when constructing the architectural model. However, since the architectural model represents a design language it is not specific for a certain development project (limited support for Re3).

Table 2 summarises our findings. All methodologies investigated above are not suitable for the transformation of organisational models into software design models. The major difficulty of all methodologies is the utilisation of concepts originally created to represent software components to model parts of reality. This way, implementational decisions take a precipitated influence on the way how the application domain is modelled. The contribution made by experts of the UoD is, additionally, impaired by the use of a language, which is difficult to understand for domain experts. The high influence of non-functional requirements on design decisions beginning from the early architectural outline of an application system is *virtually ignored by all approaches*. Only the recursive design provides supports non functional requirements on a very high level.

Requirement	Ontological state machine	OOA/D	OMT	RD
Re1	not applicable	weak	weak	weak
Re2	concept separation explicitly negated	weak separation, only by model types not by object definition	no concept separation	explicit domain separation, but no clear object definition
Re3	explicitly ignored	not considered	not considered	elicitation only in generic architectural design, not application specific
Re4	no applicable design as result	no explicit limitation, most suitable for real-time systems; no protection of manual changes	like OOA, but analysis model can no longer be separated during design	broader applicability by support of non-OO languages, most suitable for real-time applications

Table 2. Results of the Analysis

THE DESIGN TASK BEYOND TRANSFORMATION

As explained above, the transition from analysis to the design is usually described as a transformation. Transformations are systems that create an output depending on a respective input. To automate a transformation, its input has to be *complete*. This completeness, however, cannot be applied to models, which represent parts of the real world since persons perceive the world differently. Thus, it is impossible to guarantee that all information modelled in the analysis phase is necessary and sufficient to derive design models automatically. Consequently, this section is intended to broaden the view on software design by addressing decision theory aspects which guide the design process.

Decision Aspects

In software development projects, a system's designer has to choose from a theoretically infinite set of software solutions to meet the specified *functional* requirements. In real life projects, however, there will be a restricted set of possibilities to choose from, depending on the designer's experience or depending on available reuse-artefacts. To elect a particular solution, a *decision* is made considering the non-functional requirements. While functional requirements can be completely implemented, non-functional requirements can only be achieved *to a certain degree*, because non-functional requirements are often competitive (Mylopoulos, Chung and Yu 1999), so that a designer has to balance the influence of a particular design on their fulfilment.

Consequently, software design is not a mere transformation but a set of *decision tasks*. The question arising from this fact is, whether these decisions can be formalized and automated or remain strictly manual. To formalise a decision process, the variables influencing the decision have to be fully operational. Furthermore, it is necessary to fully describe the dependencies between all alternative solutions and their impact on the feasibility of decision goals (requirements) in a functional way.

Software decision problems, however, are poorly structured problems. The use of natural and semi-formal languages implies potential ambiguities in the problem statement. The influence of design alternatives on goal achievement is not fully computable – the decision is made under *uncertainty*.

The main conclusion that can be drawn by the decisive character of software design is, that development processes consist of several intellectual tasks that can only be carried out by a human task owner. The interpretation of ambiguity-burdened descriptions of real-world problems *denies the automation* of a design for instance by using transformational techniques as utilised by MDA methodologies. The theoretical feasibility of such transformations is tightly coupled with the rather philosophical question, whether computer systems can be taught to *understand* natural language in a way not conceivable with today's technology. It can be argued, that this is theoretically not feasible (Pfeiffer and Gehlert 2005).

CONCLUSION

The MDA has given rise to the question how far the abstraction from technical details can go. With the computation independent model the specification provides a means for organisational modelling. Therefore, it is evident to discuss its transformation to a platform independent model, so raising the level of abstraction to a business level.

This paper has shown that the computation independent model has not achieved much attention so far. Additionally, it has been revealed that there is no model type in MDA covering the *requirements* for the software system in development, which is the actual source model on the transition from analysis to design.

By evaluating different transformation oriented approaches on model driven software development, we were able to show the defects associated with the transformational view on the transition from the analysis to the design. All of the transformation approaches intended to map a model which is a perception of the real world (in MDA terms: a CIM) to a software design model (PIM or PSM). By resembling the structure of real world perceptions without regarding software quality requirements, the transformation results are potentially insufficient and inappropriate for the design process.

We also showed that software design is a poorly structured decision task which depends on *manual* decisions. The *computation independent model*, however, merely informs the decision makers about the system's context but does not influence design decisions in a functional describable way. Therefore an automation of the design starting from the CIM is not possible by utilising currently existent transformation approaches.

REFERENCES

1. Bass, L., Clements, P. and Kazman, R. (2003) Software architecture in practice, 2nd ed., Addison-Wesley, Boston.
2. Bunge, M. (1977) Ontology I: The Furniture of the World. D. Reidel Publishing Company.
3. Capretz, L. F. (2003) A brief history of the object-oriented approach, SIGSOFT Softw. Eng. Notes, 28, 2, 6.
4. Coad, P. and Yourdon, E. (1991a) Object-Oriented Analysis, 2. ed., 4. pr. Edition, Prentice Hall, Englewood Cliffs, NJ.
5. Coad, P. and Yourdon, E. (1991b) Object-Oriented Design Yourdon Press computing series, Yourdon Press / Prentice Hall, Englewood Cliffs, NJ.
6. Czarnecki, K. and Helsen (2003) Classification of Model Transformation Approaches. OOPSLA 2003, 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture.
7. Fraser, M. D., Kumar, K. and Vaishnavi, V. K. (1994) Strategies for Incorporating Formal Specifications in Software Development, Commun. ACM, 37, 10, 74-86.
8. Génova, G., Valiente, M. C. and Nubiola, J. (2005) A Semiotic Approach to UML Models, Proceedings of the 1st Workshop on Philosophical Foundations of Information Systems Engineering (PHISE'05).
9. Hammer, M. and Champy, J. (1994) Business Reengineering, 2nd Edition, Campus.
10. Jacobson, I., Booch, G. and Rumbaugh, J. (1999) The Unified Software Development Process, Addison-Wesley, Reading, Mass.
11. Kaindl, H. (1999) Difficulties in the Transition from OO Analysis to Design, IEEE Software, 16, 5, 94-102.
12. Kamlah, W., Lorenzen, P. (1984) Logical Propaedeutic: Pre-School of Reasonable Discourse. Rowman & Littlefield.
13. Kleppe, A., Warmer, J. (2003) Do MDA Transformations Preserve Meaning? An investigation into preserving semantics Evans, A., Sammut, P., Willans, J. (eds.): Metamodelling for MDA, First International Workshop, 13-22.
14. Kozlowski, T., Carey, T. A., Maguire, C. F., Whitehouse, D., Witzig, C. and Sorensen, S. (1995) Shlaer-Mellor object-oriented analysis and recursive design, an effective modern software development method for development of computing systems for a large physics detector Proceedings of the Proceedings of the International Conference on Computing in High Energy Physics.

15. Meyer, B.: Object Oriented Software Construction. Prentice Hall, 1988.
16. Moody, D., Shanks, G. (2003) Improving the quality of data models: empirical validation of a quality management framework, Information Systems, 28, 619-650.
17. Mylopoulos, J., Chung, L. and Yu, E. (1999) From object-oriented to goal-oriented requirements analysis, Commun. ACM, 42, 1, 31-37.
18. OMG (2003) MDA Guide Version 1.0.1.
19. Ortner, E., Schienmann, B. (1996) Normative Language Approach. A Framework for Understanding. Thalheim, B. (ed.): ER '96 Conference Proceedings, Springer, 261-276.
20. Parsons, J. and Wand, Y. (1997) Using objects for systems analysis, Communications of the ACM, 40, 12, 104-110.
21. Pfeiffer, D., Gehlert, A. (2005) A Framework for Comparing Conceptual Models, Desel, J., Frank, U. (eds.) Enterprise Modelling and Information Systems Architectures, 108-122.
22. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ.
23. Schütte, R. (1999) Architectures for Evaluating the Quality of Information Models - A Meta and an Object Level Comparison, Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E. (eds.) ER 1999 Proceedings, Springer, 490-505.
24. Shlaer, S. and Mellor, S. J. (1992) Object lifecycles: modeling the world in states, 7. print Edition Yourdon Press computing series, Yourdon Press, Englewood Cliffs, NJ.
25. Shlaer, S. and Mellor, S. J. (1997) Recursive Design of an Application-Independent Architecture, IEEE Software, 14, 1, 61-72.
26. Sommerville, I. (2001) Software Engineering, Pearson press, 6th edition.
27. Wand, Y. (1989) An Ontological Foundation for Information Systems Design Theory, in Barbara Pernici and Alex Verrijn-Stuart (Eds.) Proceedings of the Proceedings of the IFIP WG 8.4 Working Conference on Office Information Systems: The Design Process, 201 -- 221.
28. Wand, Y. and Weber, R. (1989) An Ontological Evaluation of Systems Analysis and Design Methods, in Eckhard D. Falkenberg and Paul Lindgreen (Eds.) Proceedings of the Proceedings of the IFIP TC 8 / WG 8.1 Working Conference on Information System Concepts: An in-depth Analysis, 79-107.
29. Wand, Y. and Weber, R. (1995) On the deep structure of information systems, Information Systems Journal, 5, 203-223.
30. Weber, R. (1997) Ontological Foundations of Information Systems. Coopers & Lybrand.
31. Weber, R. (2003) Still Desperately Seeking the IT Artifact, MIS Quarterly, 27, iii-xi.