Models as Programs: The Envisioned and Principal Key to True Fifth Generation Programming

Bernhard THALHEIM [a,1] and Hannu JAAKKOLA [b,2]
[a] *Christian-Albrechts-University Kiel, Computer Science Institute, 24098 Kiel, Germany*
[b] *Tampere University, P.O.Box 300, FI-28101 Pori, Finland*

Abstract. Programming became more and more comfortable with development of third and fourth generation programming languages. Although the fifth generation project did not achieve its goals, the necessity for more comfortability is still challenging. This paper delineates the path towards *true fifth generation programming*. based on *literate modelling* with model suites that generalises model-driven development and conceptual-model programming. A *model suite* consists of a coherent collection of explicitly associated models. A model in the model suite is used for different purposes such as communication, documentation, conceptualisation, construction, analysis, design, explanation, and modernisation. The model suite can be used *as a program of next generation* and will be mapped to programs in host languages of fourth or third generation. So, we claim that *models will become programs of true fifth generation programming*.

Keywords. model-based programming, models are programs

1. Introduction

Programming has become a common cultural technique, esp. for *non-computer specialists*, *engineers*, and *laymen* where the latter already start with simple tools like MIT Scratch programming or LEGO. Programs and computers have become an essential part of modern infrastructure. Programming is nowadays a socio-material practice in most disciplines of science and engineering. Despite the detailed research knowledge gained so far, the *quality of programs* decreases, for instance, due to the wide variety of program applications, due to the large variety of program libraries and their constant evolution, due to the numerous languages and toolboxes, due to integration and impedance problems among already existing programs, due to application of different programming cultures, and due to missing provenance and documentation support. Programming is not the most natural kind of communication for many programmers. They reason in a different language and at different abstraction levels. They often have difficulties in understanding their own programs later on or programs developed by others. At the same time,

[1] thalheim@is.informatik.uni-kiel.de
[2] hannu.jaakkola@tuni.fi

systems become more complex and thus less comprehensible. We are thus approaching a *software crisis 2.0* [Ama16,Far16,VM11,WVH17]. Programs are still developed on the basis of the third and fourth generation although the underlying mental concepts are not expressible in these languages. Moreover, critical software components for everyday life systems, for infrastructure, for management and control are developed by *non-computer-specialists* who are not familiar with matters of maintainability, risk avoidance, error tolerance, precision, completeness, integration, migration, and component coherence.

However, *programmers have already initially and intrinsically an idea and models how to solve their problems and how to solve them*. This idea is the rationale which underlies programming, i.e. it is a mental model of the solution that is going to be developed. As long as the models behind are only intrinsic and hidden, the solution background and the program ideas become tacit secrets of programmers. It seems far better if an appropriate support for modelling, gradual improvement, and refinement of the models is provided. If this support becomes sophisticated and code can be generated from models, the need for program development is reduced to the real problematic cases which are resolved by professionals. In this case, some models in a model suite [Tha10] become programs at a higher level of programming. They are compiled to classical programming languages. *Therefore, models become programs at a higher and more comprehensible level. They are more efficiently and correctly developed.*

1.1. The Path Towards True Fifth Generation Programming

Our approach fundamentally revises, combines, and corrects two already existing approaches: (1) Model-driven development approaches (MDD) (or engineering or architecture) are the latest developments (e.g. [SV05]). Users start with requirements and continue with platform-independent models which can be specialised and refined to platform-specific ones. The platform-specific models are translated to code. Yet, the mental model behind the requirements is not explicitly considered. The approach does also not consider the intrinsic details of all the models. (2) Conceptual-model programming [ELP11] asserts that programming activities can be carried out via models. Models are complete and holistic, are conceptual but precise, and are executable. These models can be refined at any level of abstraction. However, the underlying foundations remain incomplete thus hindering full realisation. Both approaches have so far failed to fully generate deployable systems. The path towards model-based programming has however already been tested for web information systems. A third approach, which is mathematically precise, is based on abstract state machines [BR18] that offers a semantically well-defined, pseudo-code language for specification at various abstraction levels. These models provide an accurate high-level description, support quality assessment, and can be mapped to third generation programs. By combining the first two approaches with the mathematically precise description, ***Modelling-as-Programming*** (MaP) will be a *springboard for next generation programming*. Next generation programming will be the first step towards true fifth generation programming.

1.2. The Storyline of This Paper

The paper develops a programme for true fifth generation programming that starts with models and uses models as a program specification. It is similar to second and third gen-

eration programming where programmers are writing programs in a high-level language and rely on a compiler that translates these programs to machine code. We propose to use models instead of programs and envision that models can be translated to machine code in a similar way. This paper presents the first starting vision to such novel kind of programming. The completion and full establishment of this programme may take a decade. Information system modelling is, however, already a positive proof of this kind of programming by models. Models delivered include informative and representation models as well as the compilation of the model suite to programs in host languages. Models will thus become executable while being as precise and accurate as appropriate for the given problem case, explainable and understandable to developers and users within their tasks and focus, *changeable* and *adaptable* at different layers, *validatable* and *verifiable*, and *maintainable*. Therefore, we start first with a discussion what models-as-programs means. Next we discuss literate modelling as high quality modelling with model suites. Section 4 describes our envisioned realisation path.

2. Modelling - The Next Generation Programming

Model research has a long, more than 2000 years old history in sciences, engineering, and daily life (e.g. [Mül16,TN15]). Computer science and engineering uses models as the main vehicle for system construction, description of problems and systems, explanation, and system quality investigation. Computer science has developed a very large number of model notions. As investigated in [TN15], these notions mainly differ according to the model purpose, the attention of the community, the background, and especially the language setting.

2.1. Modelling is Often Only Normal Modelling

The main difference to classical programming, model-driven development, and conceptual-model programming is the explicit orientation on the extrinsic surface model called *normal model* (yellow color in Figure 1). By contrast, the *deep model* (green color in Figure 1) consists of the background, the context, the intentions behind the model, the commonly accepted practice in the community of practice (CoP), and the setup of the model. The deep model and the normal model should however be considered as a whole.

We use the model notion from [TN15,Tha18][3] that is depicted in Figure 1. An essential result of the interdisciplinary brainstorming seminars of the modelling commu-

[3] A model is a *well-formed*, *adequate*, and *dependable instrument* that *represents origins* and that *functions in utilisation scenarios*. Its criteria of well-formedness, adequacy, and dependability must be commonly accepted by its CoP within some context and correspond to the functions that a model fulfils in utilisation scenarios. The model should be well-formed according to some well-formedness criterion. As an instrument or more specifically an artefact a model comes with its background that is often given only in an implicit and hidden form and not explicitly explained. The *background* consists of an undisputable *grounding* from one side (paradigms, postulates, restrictions, theories, culture, foundations, conventions, authorities) and of a disputable and adjustable *basis* from other side (assumptions, concepts, practices, language as carrier, thought community and thought style, methodology, pattern, routines, common sense). A well-formed instrument is *adequate* for a collection of origins if it is *analogous* to the origins to be represented according to some analogy criterion, it is more *focused* (e.g. simpler, truncated, more abstract or reduced) than the origins being modelled, and it sufficiently satisfies its *purpose*. Well-formedness enables an instrument to be *justified* by an empirical corroboration according to its objectives, by rational coherence and conformity explicitly stated through conformity

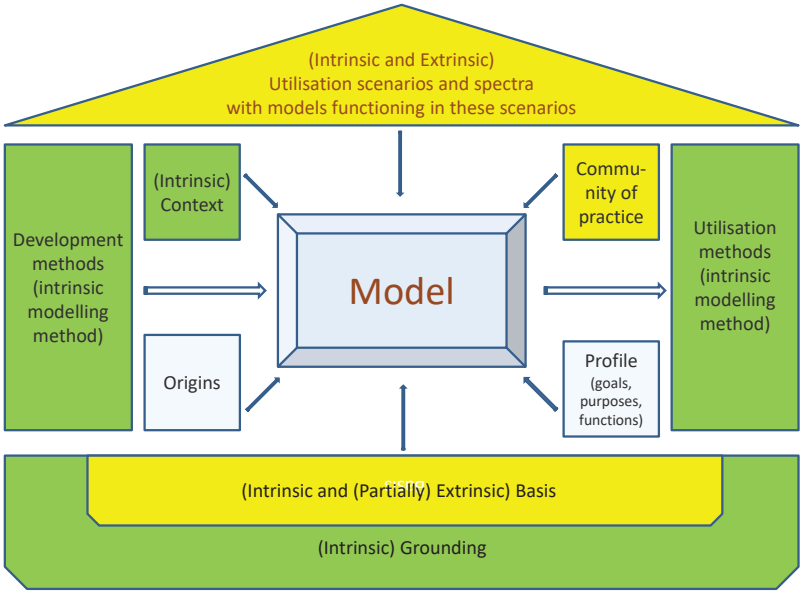

Figure 1. The conception and notion of a model with extrinsic elements of the normal model (yellow color) and intrinsic elements of the deep model (green color)

nity at Kiel University since 2009 has been the explication of the intrinsic enthymeme-like deep model within all models used in science and technology [TN15]. Modelling is currently mainly modelling of the surface-like normal model without explicit description of the background. The normal model is bound to its deep model. It is thus not entirely understandable to anybody, e.g. outside its context (e.g. discipline) and its CoP. The concentration on normal modelling is one of the main reasons why model-driven development and conceptual-model program have not succeeded as expected. If the deep model is not known and not understood then translation or mapping to platform-specific models becomes infeasible. This situation is similar to specifying LaTeX text without the corresponding strategic setup, e.g. by .cls, .clo, .def, .bst, .sty etc. files and libraries. The separation into the intrinsic and extrinsic parts of models is also depicted in Fig. 1 where the light blue part the normal model, the yellow part represents the mixed extrinsic and intrinsic part, the green part the part that is mainly build from the deep model. The explicit description of a deep model reveals the secrets within models.

The deep model has not been considered for model-driven development (MDE, MDA, MDD) and conceptual-model programming. This non-consideration is the main source for impedance mismatches between source and host languages, crucial translation problems, and the failure of these approaches. The *explicit treatment of deep models* and of high-quality source models (e.g. standardised generic and reference models) is

formulas or statements, by falsifiability or validation, and by stability and plasticity within a collection of origins. The instrument is *sufficient* by its quality characterisation for internal quality, external quality and quality in use or through quality characteristics such as correctness, generality, usefulness, comprehensibility, parsimony, robustness, novelty etc. Sufficiency is typically combined with some assurance evaluation (tolerance, modality, confidence, and restrictions). A well-formed instrument is called dependable if it is sufficient and it is justified for some of the justification properties and for some of the sufficiency characteristics.

THE *essential* difference of our approach. Complete knowledge about the model is THE *guarantee for modelling as programming*.

2.2. Models as Model Suites

Researchers and engineers often collaborate in interdisciplinary and interacting communities. A model suite [Tha10] can also incorporate viewpoints and sub-models that support interaction and exchange with collaborating partners on the basis of sub-models. I envision that model-backed collaboration is far more effectively support collaborative work and problem solving in communities.

A model may combine several facets at the same time and may thus have its structure where some facets support specific purposes and functions. A model suite is a coherent collection of well-associated models at a variety of abstraction levels, foci, and scopes. The associations are explicitly stated, enhanced to explicit maintenance schemata, and supported by tracers for the establishment of coherence. Coherence describes a fixed relationship between the models in a model suite.

Model suites support holistic and consistent description of models at numerous levels of detail, precision, completeness, foci, and scopes depending on context, function of the model, community of practice, and origins that are really of interest. They close thus the gap among ideas and intentions, requirements, conceptualisations, and realisations. Models in a model suite support various functions such as communication support, mediator for system construction, basis for problem solving, facilitator for contracting and negotiation, documentation, analysis and quality assessment, support for integration, and warrant for migration and modernisations. Representation and informative models are typical models in a model suite. The latter can be generated. Models in a model suite can also be generated from others, e.g. in order to represent viewpoints [Tha10].

Model suite development is an intellectually challenging task if we aim at a complete model suite. For this reason, MaP also incorporates toolbox support.

2.3. Models are New Generation Programs

Models are currently used as a prescription or blueprint for programs of the third or fourth generation. We envision that models themselves can be considered to be the source code, i.e. models and model suites are essentially the program source. The independence concepts (hardware, operation system, physical, and logical independence) will be extended by programming language, platform, environment, and system independence since models can be transformed to different kinds of programming languages. The translation requires sophisticated compilers including optimisation facilities. Models in a model suite can be translated to code while other models in the model suite serve as communication and collaboration means in the CoP.

MaP proposes now new programming paradigms, develops novel solutions to problem solving, integrates model-based and model-backed work into current approaches, and intents to *incubat*e true fifth generation programming.

3. Literate Modelling as Literate Programming

3.1. Towards Literate Modelling

A holistic approach entirely based on models will thus provide a better support. The model support must include multi-model treatment with coherent model ensembles at different levels of abstraction, with explicit and maintainable associations among these models, with supported intellectual management of the complexity, with explicit knowledge of details of the models, and with sophisticated quality management. Models have also their anti-profile [Tha17] that limits applicability of model-backed development. Such a holistic and general approach would be too ambitious and unrealistic. Therefore, MaP focusses its scope on selected areas of Computer Science and Engineering. We, thus, start with four application areas of model-backed development and then use the experiences gained to extend our scope.

Already literate programming [Kn84] considered a central program together with satellite programs, especially for interfacing and documenting. We generalise and extend this new paradigm of programming with GibHub, 'holon' programming, and schemata of cognitive semantics. Projects like the Axiom project or the mathematical problem solver [Pod01] have already shown the real potential of literate programming. The association among models must become manageable and be supported by computational features. The design and development of model suites has realigned the model ensemble approach to meet this challenge. One reason that literate programming has not become a mainstream paradigm is that implicit and intrinsic components remain largely unknown. Another reason is the missing representation of models behind the code and the missing representation of thoughts of people. A third reason is the hidden representation of the intention and the application task that has been the reason for developing a program. A fourth reason is the implicit usage of experience and of generic models behind the program solution. Our approach will reveal intentions, strategic and tactical issues (see Figure 2).

Model-backed development of systems will not be a universal solution to all computational problems. It is however a solution for those application cases for which users have an idea that can be expressed as a mental model. These models can be understood as interfacing or communicated models. In this case, the mental model can be enhanced by models characterising the problem space according to the needs, interest, and intentions of users. Users have their own understanding of the problem space, their educational and work environment, and their culture as 'programming of their mind' [Hof01]. Different users might use different models for the same application case. That means, we support modelling as *literate modelling*. It frees the modeller from the inherent and implicit parts of a model as modelling is understood at present and imposed by modelling languages and means that the modeller can develop models in the order of the flow of their thoughts. A model suite also explains the model and its intrinsic components in a natural language and is interspersed with snippets of representation and realisation models. This means that models are very easy to understand, to justify, and to share, as all its details are well explained. Literate modelling is a change of the mindset by making the story of the model suite explicit. It reduces bugs, misconceptions, and flaws in a model. Models are communicated to both people and machines.

Models can be transferred to programs if all details within the model are known and the models themselves are well-structured based on a sophisticated model language, e.g.

extended entity-relationship models with stereotyped and refinable profiles and directives for realisation (among stereotypes we may select the default one) [KT16]. A general model language would be the basis for a universal solution and thus cannot exist. We can however use modern engineering approaches. Engineers already develop systems based on standardised components. They use composition pattern and some kind of quality and failure management. Components and compositions can be coherently specialised in machine tool building. They are based on standards in this case. Database and workflow models can also be built in this form [MNS+13]. Standards are in these cases generic models or reference models. We restrict our approach to this kind of models. This standard-backed approach can also be applied to model suites. All models that are not directly derived from mental models are standard-backed models. Mental models are going to be enhanced and generalised in such a way that they become the source for a generic or reference model. This harmonised treatment then supports model-backed development of programs. Thus model suites become the source for programs.

3.2. Towards Next Generation Programming as Starting Point for True Fifth Generation Programming

The rationale behind the initial fifth generation program language project was very ambitious [Mo82]: development of a general-purpose, multilingual environment and general-purpose problem solver that also supports natural language communication and multimedia processing; support for general knowledge representation, storage, processing, and retrieval; automatic problem-solving after accurate and abstract problem specification; closing the mental and the language gap between users and computers; development of sophisticated logical and functional machines for backend computation; developing an advanced architecture for support of these features; development of sophisticated theories and technology for support; development of a distribution and collaboration architecture. However, the initial fifth generation programming languages project was never completed. It failed despite its great technological and social advances since it was too early for the hardware progress, it was highly dependent on AI technology, it did not achieve an integration of AI and human-computer interface techniques, it was oriented on one programming paradigm and on mathematical logics, it tried to provide a universal solution for any kind of programming, it routed granularity to basic program blocks, and it was oriented on one final solution instead of coherent reasoning of coherent variants of final solutions depending on the focus and scope of a user[4].

MaP now aims at *true fifth generation programming* where *models are essentially programs of next generation* and models are translated to code in various third or fourth generation languages. Programs of next generation programming must at the same time be understandable by all parties involved, support abstraction, be as accurate and precise for the problem space and the issues to be solved, transferable and distributable to other parties, commonly deployable by all parties, and support quality management and reasoning.

Due to its user orientation, next generation programming cannot rely on single language paradigms. Instead, models as programs must become *language independent*. Languages of third and fourth generation of programming languages became already hard-

[4]Our approach has been inspired by H. Aiso [Ais88] who chaired the architecture sub-committee in the Japanese fifth generation computer project [Mo82].

ware, storage, operating system, firmware, and optimiser independent. Language and platform independence will support non-specialists to program based on their models without forcing programmers to certain style of thinking and programming.

Our approach is based on stereotypes for deep models and on generic and reference models as a starting point for normal models. Model-driven development, engineering and architecture (MDD, MDE, MDA) taught some valuable lessons reported about model-driven approaches, e.g. those in the list in [Web95]. Two main problems limited the applicability of MDA and MDD: the intrinsic and not explicitly stated deep model and the restrictions in layering development.

Models are a more natural kind of human reasoning than programs could be. Programs are often oriented on algorithmic thinking. Most programming languages use simple variable spaces. Object-orientation has added essentially user defined object-relational structures. Network diagrams are one of the reasons why the entity-relationship structuring became quickly popular. All these structures are, however, one-facetted, cognitively simplistic, and without multiple viewpoint representation. *Human communication* is partner-oriented, is ambiguous in structure and meaning, uses partially semantics, is culture-dependent, is oriented on parsimony instead of completeness or preciseness, uses previous communication histories, considers principles of communication such as politeness, uses background information, and incorporates ellipses and context-dependent abbreviations. Models are represented according to the communication flow and the communication partner. Models thus must have a number of faces (or contrast classes and relevance classes [Fra80]) that can be used interchangeably. *Human thinking* does not separate syntax, semantics, and pragmatics but treats them as a coherent and larger whole. It is rather based on mental models such as collections of interrelated personal perception models or environment- and culture-oriented domain-situation models. Moreover, it is complex, multi-facetted, highly adaptable to different viewpoints and opinion, multi-viewpoint oriented, and network-connected.

Non-professional programmers are confronted with problems of transferring their understanding and their models to algorithmic and computer-oriented environments. The transformation process from thoughts to programs is error-prone, is oriented on the normal case without consideration of the entire picture, requires one central representation, and some understanding of computing technology.

Therefore, it is far better to support *non-professionals by model-backed programming* instead of forcing them to learn and to fully understand programming in third or fourth generation languages. True fifth generation programming is a better model-based representation of problems. In this case, the model must be understood both in their extrinsic, directly represented components and their intrinsic background. The second part has not taken into consideration in model-driven development and conceptual-model programming. This second part is, however, a central necessity for model-backed next generation programming. The explicit treatment of this part will become a 'silver bullets' for the new programming.

4. The Programme and Its Realisation Path

4.1. The Layered Model Development Framework

The layered approach has already often and successfully been used in Computer Engineering. Most program language realisations follow this approach since COBOL and ALGOL 60 development (e.g. infrastructure definition; variable space; program space; interpreted or compiled code) and application development (e.g. application case; infrastructure; design; specialisation & tuning; Deliver). Layering has also been the guiding paradigm behind text processing, e.g. behind the TeX and LaTeX realisations [Knu86,Lam94] with a general setup layer, the content layer, the adaptable device-independent layer, and the delivery layer. We assume that this approach is the universal basis for treatment of models as programs at least for programming by non-specialists. The approach for professional programmers is different. It can, however, also be supported in this manner how the success of programming environments such as Eclipse has already been demonstrating. These toolboxes have become accepted for their ease of use.

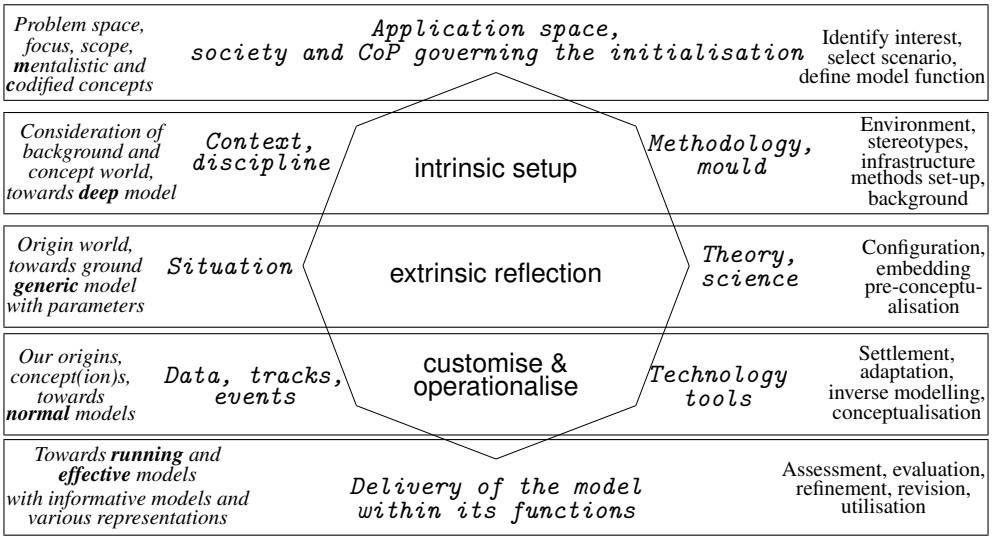

Figure 2. The layered approach to model suite development and program generation

The *model suite* will be layered in Figure 2 into models for initialisation, for strategic setup, for tactic definition, for operational adaptation, and for model delivery. At the left side the issues for the model suite are represented. The right side displays the activities and methods for the development. The corners of the octagon represent the starting and final stages as well as sources and enablers of the intermediate stages. We restrict the picture to the layered model development process. The complete model suite thus becomes the source for the code of the problem solution, and for the system to be built. Currently, one model is considered to be the final product. Model suite development results in a number of models: deep, generic, specific, and normal models. Since any model has its deep elements, we start with the development of this deep model. In many cases, we use reference models or generic models (or tactical model frames like those used in

data mining and analysis). Models have their own background that is typically not stated explicitly but intrinsically. Methods for developing and utilising models are considered to be given. The intrinsic part of a model and these methods form the deep sub-model [Tha18]. The deep model is coupled with methodologies and with moulds that govern how to develop and to utilise a model. The deep as well as the general model are starting points for developing the extrinsic or "normal" part of a model. Consideration of modelling is often only restricted to normal models similar to normal science [Kuh70]. Classical modelling often intentionally presupposes the initialisation and intrinsic layers and assumes that these layers cannot be reconsidered and specifically changed according to the functions. The developer thus loses the understanding of the model and why the model is dependent without an understanding of these layers. Model suites, however, integrate these models over all layers. Another main obstacle why model-driven development and conceptual-model programming has not yet succeeded is the non-consideration of modelling moulds.

Intention modelling extends rational-based software engineering [BCM+08]. The W*H specification pattern [DT15] can be applied to *model initialisation* as well as includes then the following set of statements: (1) a plan, function, and purpose dimension (model as a conception: 'wherefore', 'why', 'to what place or end', 'for when', 'for which reason') within a scenario in which the model is going to be used as an instrument; (2) a user or CoP dimension ('who', 'by whom', 'to whom', 'whichever') that describes the task portfolio in the CoP and profile of users including beliefs, desires and intentions; (3) an application and a problem dimension ('in what particular or respect' , 'from which', 'for what', 'where', 'whence'); the added value dimension (evaluation). The initialisation layer may also be enhanced by a contrast space for user-related separation of a model and a relevance space that is dependent on the user [Fra80]. The contrast and relevance spaces as a form of mind-setting also define what is not of interest.

The *enabling intrinsic setup layer* defines the opportunity space and the infrastructure for the model. The results will be on the one hand a deep model and from the other hand a modelling framework or modelling mould that guides and govern next activities. In future, the developer will define the *context* and the most of the *background* (the grounding (paradigms, postulates, restrictions, theories, culture, foundations) and the basis (assumptions, concept world, practices, language as carrier, thought community and thought style, methodology, pattern, routines, common sense)) of the model. The context, extrinsic, and strategic dimension answers question like 'at or towards which', 'where about', 'to what place or situation', and 'when'. Additionally, developers decide which methodology and environment seem to be the most effective and purposeful. The development and deployment dimension ('how', 'whence', 'what in', 'what out', 'where') defines the modelling methodology, i.e. the modelling mould.

Deep model elements will be separated from elements of the normal model at the extrinsic source reflection layer. According to the model function, the normal model represents extrinsic elements of potential origins based on their content and thus answers questions such as 'what', 'with which', and 'by means of which'. It reflects the extrinsic theory essentials that are necessarily to be represented, e.g. conceptions or pre-conceptions from the theory that is underpinning the application. The normal model can be built from scratch ('greenfield' modelling). It is usually based on experience gained. The latter case thus starts with a generic or reference model that might incorporate parameters. The extrinsic source reflection layer can be understood as a tactical layer.

Generic or general normal models are adjusted to those that a best fitted to those origins that are considered for the application in the *operational customisation layer*. This layer is sometimes holistically handled with extrinsic reflection. Inverse modelling uses this layer for adaptation of the model to the observational data (e.g. data adaption in astrophysics or parameter instantiation in most data mining processes). In some cases, this layer seems to be trivial. It is not trivial in the general case however. It instantiates parameters, adapts the normal model to those origins (or data sources) that are really under consideration, prepares the model for the special use and to the special - most appropriate - solution, and integrates the deep model with the normal model. The normal model is typically pruned in order to become simpler based on Solomonoff and Occam principled deviation and error-prone. The (normal) model could be enhanced by concepts and thus become a *conceptual model*.

The final result of the modelling process is a model suite that is adequate for origins, properly justified, and sufficient at the *delivery and product layer*. We cannot expect that one single model is the best instrument for all members of the community of practice. A sophisticated model that integrates deep and specific normal models is delivered to some members. An informative model that is derived from this model can be better for other CoP members. Models delivered in the finalisation stage are often enhanced by additional annotations, e.g. relating the model to the demands for members of the CoP by answering the 'with', 'by which', 'by whom', 'to whom', 'whichever', 'what in', and 'what out' questions. At the delivery and product layer we, thus, generate a number of associated models.

Models delivered in this approach become more reliable and - in the general sense - dependable on the explication of the deep model and of the initialisation. Dependability can now be considered according to dependability that is already given by deep models from one side and by generic and reference models from the other side. For both kinds of models we use stereotypes and pattern similar to the usage of class and setup libraries in LaTeX and the special document templates that provide a specific parameterised structure for content development for LaTeX content input (e.g. .tex and .bib files). Skipping the operational layer can be an option if a single model is delivered as a program collection. The typical case, however, is adaptation, fitting, pruning, specialisation, operationalisation, and exemplification at the operational layer.

Transformation is based on standardised combinable components (not only basic elements) as pattern and templates. *Generic and deep models* are going to be developed on the basis of standardized stereotypes and pattern. The specialisation and combination of models is supported by a model algebra that generalises the ER algebra [MNS+13]. Each specialisation can be enhanced by directives similar to pragmas in C++.

4.2. A Path towards Modelling-as-Programming

Computer Science and Engineering has resulted in many tools for support of programming. However, we observe that most of these tools have been oriented on bottom-up representation of program language elements and constructors for programs. Some of the tools provide some kind of abstraction. Very few tools also allow introduction of *components* and support modularity with refinement. Many tools are based on their own language variant and own interpretation, e.g. [Kru04]. Tools should however be based on *standard components* that can be refined for specific purposes. This path of *componenti-*

sation is essentially implemented with programming languages of the second, third, and fourth generation - at least for bottom-up elements and block concepts. Many tools tend to be unnecessarily complete in order to provide the full flexibility. All tools consider syntax on its own, define semantics of elements and construction on top of syntax, and do not consider personalised pragmatics. Natural languages have however collocations for words, holistic syntactical-semantical constructions, and their special interpretation in dependence on the context and the community of practice.

We are going to partially represent generic normal models in three frameworks:

- ADOxx [KMM16] is a *configurable meta-modelling development and configuration platform* that supports specification in a larger variety of graphical modelling languages. It follows the MOF (meta-object-facility) approach by OMG [PM07] based on a separation of abstraction layers of specification languages: M1 as the layer of model creation and description; M2 as the layer model language specification (considered as meta-model); M3 as the layer of frames for language specification (considered as meta-meta-model). The compiler approach can be integrated into ADOxx.
- Ptolemy II [Pto18] focuses on *actor-oriented modelling of complex systems*. The application of Ptolemy II in our approach must, however, consider a number of specific problems and must develop solutions to them. Ptolemy is oriented on bottom-up level of components. Abstraction in specifications is still an issue. There is a high freedom for specification and thus the approach struggles still with standardisation. Generic normal model can be used for standardisation. Intrinsic strategic and tactic parts of models are not yet considered. The model suite concept fits well into Ptolemy II.
- KIELER [Kie18] provides an *eclipse-based framework for diagrammatic model specification*. It aims at improving comprehensibility of diagrams, in decreasing development and maintenance time, and in providing facilities for analysis of dynamic behaviour of diagrammatically represented processes. Semantics is based on sequentially constructive sequence charts. Normal models can be represented as long as they are given in diagrammatic form and as long as their semantics if based on sequence charts.

Model-based development and architecture as well as conceptual-model programming have also been bound to imperfect tools. Moreover, they fail since the *deep model* is not taken into consideration. They meet thus all the classical impedance mismatches. A *proper transformation* can only be developed if either the source and target share their deep models or the deep models are transformed as well. Above all, programs are developed by people who have their culture, esp. programming culture and who are biased and framed by their way of programming and working.

This approach is based on a number of *new assumptions*: models consist of specialised and refined components that are combined via construction expressions; model components can be stereotyped and refined based on a specialisation approach; interdependence of refinement can be handled by attribute grammar constructions; construction expressions can also be stereotyped; stereotypes form the strategic layer of description; stereotypes can be specialised to pattern at the tactical layer and to templates at the operational layer; stereotypes, pattern and templates form semiotic units with their own specific syntax and with their fully integrated semantics.

Each sub-discipline in Computer Science has developed its specific style of modelling. This style is based on specific languages which have their specific grammar. Following the Eugenia [PKP14] framework, attribute grammars can be developed for these languages. In this case compiler-compiler approaches become applicable [BL74].

The Kiel team has been participating in tool development for database design, database engineering, and database performance management. Starting in the 1980s with the RADD (Rapid Application and Database Design) workbench (e.g. [AAB+95]), we systematically extended the domain of structure specification by database programming (finally with the VisualSQL tool (e.g. [JT03])), by performance management and tuning (e.g. [TT11]), by integration of workflow specification (e.g. [BR18]), by integrating web information systems design (e.g. [ST04]), and by codesign (e.g. [Tha04]). These specification methods have been extended to a methodological framework (e.g. [JMTV05]) that finally reached maturity level 3 in SPICE in 2005 in one of our collaboration projects. We have also developed the translation tools for transformation of (conceptual) models to code.

4.3. A Proposal for the Realisation Approach

The implementation approach to MaP is inspired by four projects.

(a) *Transformators and compilers for conceptual database models*: The RADD toolbox (Rapid Database Development) is based on the conceptual entity-relationship modelling language. This graph-based language supports conceptual development, documentation, reasoning, and requirements engineering for database analysis, design, and development. The graph-oriented approach has been compiled on the basis of graph grammars [AAB+95,Run94,Tha00] combined with attribute grammar approaches. The conceptual schema formulated in this language or enhancements of this language can be used for derivation and compilation of realisation schemata, especially for object-relational and XML platforms. It is enhanced by view suites, visual query systems (VisualSQL), and by performance optimisers. Currently, the development is transferred to ADOxx [KMM16] from OMIlab. The compilation approach is presented in [KMM16].

(b) *DEPOT-MS (DrEsdner PrOgrammTransformation)* [BL74]: DEPOT system is a compiler-compiler for domain-specific languages (DSL) (historically: little languages, application-domain languages (Fachsprache)) that has been used to compile specific language programs to programs in the mediator language (first BESM6/ALGOL, later PASCAL, finally PL/1 [GHL+85]) which can then be translated to executable code. The approach integrates the multi-language approach [Ers81], attribute grammars [GRW77,RF87], and theory of grammars [Hut86,Tha75]. The system is similar to the MetaCASE toolbox [Dah97] or development environments, e.g. Ptolemy II.

(c) *LaTeX and TeX* [Knu86,Lam94]: The TeX and LaTeX approach is based on a strict onion or layered approach with (1) an internal layer for formatting and general initialisation (e.g. .fmt, .tfm, .fd, .def, .ltx, .dat, .afm, .cfg. .clo files), (2) a structure-style-language layer (e.g. .cls, .sty, .ldf, .bst files) that includes many additional library packages, (3) the input document suite (mainly .tex, .bib, .ist files), (4) the internal supporting and generated layer (e.g. .aux, .log, .lof, .bbl, .ind, .toc, .lof, .log files) that also support related applications, (5) a generic intermediate output layer (especially .dvi files), and (6) a delivery layer (e.g. .pdf, .ps, .html, specific printer files) for multiple output variants.

(d) *Libraries of reference models* (e.g. [BKV17,FL07,KMM16]): Libraries and toolboxes of solutions and programs are widely used in science and engineering. Specific

reference models are universal models [MJ04, Sil01] as well as generic models [Tro16]. Universal and enhanced models may be algebraically combined [MNS+13] and refined [dRE98] based on a model calculus. Models may also be enhanced by metadata descriptions and by informative models [DT12,Kra18]. Models and model suites may be evaluated based on their potential and capacity [BT15].

The integration of these four technologies is very ambitious. Generation of programs from models extends the models@runtime initiative [BFCA14] by direct compilation of programs from given models instead of enhancing runtime environments by models and abstractions. The proposed layering might however provide a solution for this integration and the necessary harmonisation. The variety of application-domain languages is as large (an estimation stated about 2.500 such languages in 1985) as the one of DSL [Fra11] or multi-level languages. Our layered architecture for models is going to be combined with the abstraction/refinement approach [Bör07,dRE98]. The layered architecture became a common culture in Computer Science. Modern systems have been built on thus kind of layering for system development (e.g. by layering into application case - infrastructure - design - specialisation & tuning - delivering), for problem solving frameworks (e.g. by task ordering ((1) problem case, (2) setting, (3) incubation, (4) enlightening, (5) finalising), for data analysis (e.g. by workflow pattern ((1) define & identify, (2) select solution class, (3) select solution pattern, (4) derive parameter values, (4) fit & prune, (5) finalize), and for engineering (based on general approaches ((1) know it, (2) understand it, (3) construct it, (4) configure it, (5) use it)).

The first three inspiring projects are based on compiler technology, attribute and graph grammars, pattern and stereotype architectures [ANT14], and principles of programming languages (starting with early thoughts [Lan73] to more advanced ones [GGZ04,SGM02,Wir96]). We oriented model transformation on macro-level, component-oriented, and refinable translation [FG11] instead of meso- or micro-level transformations used for most syntax-oriented translators. Models typically consist of associated and bundled components that have their inner specialisation.

The translater is also used for generation of warnings and error messages for systematic improvement of models. Since we start with development of generic normal models and deep models, we concentrate on quality assessment and improvement for normal models. Normal models should be as adequate and dependable for the given application. Later on, quality establishment is extended to the strategic and tactical layers.

5. Conclusion

MaP aims at true fifth generation programming as a new programming paradigm where models are essentially programs of next generation and models are translated to code in various third or fourth generation languages. Users program by model development and rely on the compilation of these models into the most appropriate environment.

5.1. The Intended Outcome of Our Approach

The main outcome of MaP is a *proposal for true fifth generation programming as programming by models*. The capacity and the potential of MaP are evaluated. The strengths, weaknesses, opportunities, and the threats are demonstrated in the four application areas in such a way that they can be used for an extrapolation to other application areas.

An essential outcome of MaP is the *layered architecture* and a realisation of modelling as programming. Model suite are used as a foundation for literate programming. Quality and literate models are understandable, transferable, distributable, and commonly usable. MaP users may design their own modelling styles, templates, stereotypes, and configuration. They also may concentrate on development of normal models while inheriting the initialisation and configuration, the deep models, the methodology, and the techniques for model realisation and model representation.

The MaP approach supports *programming by everybody at any time*. Models become the main means for collaboration among partners. Models may evolve and therefore evolution and modernisation are less painful tasks. For non-specialists in programming, models are typically of higher quality than the programs. There-fore, the generated programs are of higher quality. Models can also be used for communication and exchange of experience. Modelling as programming thus support sustainability of developed solutions. The model is then the code. The compiler assures that program execution corresponds to the conceptual specification thus making the model directly executable. At the same time, model suites treat the model in an explicit, complete and holistic way without any intrinsic and hidden details. Elements of a model suite are conceptualisations of the thoughts and understanding of developers. They are precisely defined and commonly agreed with the concepts in the application space. I will thus attain a good level of parsimony for model and therefore program developers. Furthermore, application evolution is going to occur at the level of the model suite. Modernisation, migration, and evolution occurs at the level of the model suite and does not require consideration of lower level details.

MaP supports model-based reasoning as a natural kind of reasoning. Solutions can be developed in a large variety of reasoning styles including hypothetical, abductive, inductive, deductive, and other advanced reasoning methods. Models can be refined. MaP thus also supports inverse modelling.

5.2. Are You Still Programming or Are You Already Modelling as Programming?

Programming has become a central technique in science and engineering. Software systems are often developed by non-specialists in programming without a detailed knowledge and skills, without an insight into the culture of computer science, and without plans for systematic development. These systems and programs often have a poor structure and architecture, little documentation, and lost their insight and knowledge of specific solutions.

Programs of the future must be understandable by all parties involved, must be accurate and precise enough for the task they support, and must support reasoning and controlled realisation and evolution at all levels of abstraction. Programming languages are currently languages of third or fourth generation. Those generations have so far provided hardware independence, linker independence, operating system independence, and execution code independence. Programs are nowadays compiled or at least interpreted and do not require system knowledge by the ordinary programmer.

In the past, the fifth generation computing project sought to develop systems and programs that are closer to people in their communication and knowledge processing capabilities. It should have been a shift to a new paradigm of human-oriented computing in the sense of T. Kuhn [Kuh70]. This world-wide project failed despite its great technolog-

ical and social changes because it was too early, it was highly dependent on AI technology, it did not achieve an integration of AI and human-computer interface techniques, it was oriented on one programming paradigm and on mathematical logics, it routed granularity to basic program blocks, and it was oriented on one final solution instead of coherent reasoning of coherent variants of final solutions depending on the focus and scope of a user.

5.3. Envisioned Deliverables of MaP

This paper develops the general approach to true fifth generation programming. The realisability of the approach has already been demonstrated for database development [KT16]. Database specification follows the *global-as-design* approach. BPMN specification follows the *local-as-design* approach. This approach requires view schema specification for data support of the workflow diagrams. The co-design approach [Tha04] is the basis for integration of workflow specification to database specification.

The general proof of concept is however a task of the future. Our programme can be based on development of the following deliverables:

Model suite description language: The model language consists of an associated bundle of languages for model elements that users may modify depending on their needs or simply reuse them as already established sub-cultures. These elements are different from ordinary programs because they are essentially declarative rather than imperative. Similar to UML stereotypes, MaP model class and model style languages are ready for application, can be extended, combined, and adapted. Users don't have to work on the details of the models as programs. The system takes over the integration and composition work as it deduces the consequences of the model. It also provides a new discipline of modelling according to which principles of a particular modelling language design can be stated precisely. The underlying intelligence does not remain the secret of the modellers. It is spelled out in the style language and based on the model class language. Thus, coherence and consistency can readily be obtained where they are desirable. New model elements can readily be extended to new elements that are compatible with the existing ones. The model suite description language is developed as a collection of grammars, grammar-aware theories and software, and techniques for implementation.

Technologies, techniques, methodologies, and modelling moulds: The development of models in a model suite is based on a model library with models that can be used as an inherited or initial model for systematic composition of the model suite. The approach to model construction is canonised on the basis of methodologies and modelling moulds which systematically combine known and novel techniques and technologies for model development. Modelling moulds enable the modeller to reuse systematics and theories that have been successfully deployed in the past. They enable us to start with application space models, with deep models, with generic models, and with reference models without explicit reinvestigation of these models. The explicit agreement on a given mould eases, enables, and supports an economic development process.

Environment for an extension towards modelware as next generation literate modelling: Our approach aims at development of a general infrastructure for treatment of models as programs. This infrastructure includes also standardised solutions that can be reused in other applications. These solutions are based on application space models and deep models that are typically less volatile than normal models. Therefore, the library

allows quick and well-based modelling by non-specialists which may concentrate on development of normal models instead of developing the entire holistic model. They may accept the library models as a basis and then use generic and reference models as a starting point for normal model development. The model suite is also transformed to programs in programming languages of third or fourth generations. This approach disentangles modellers from programming and allows them to concentrate on the model development. The model is then the product. The transformed program is of a higher quality and more liable.

Compiler-compiler approach to model realisation for models as programs: The modelling infra-structure is an essential element for realisation of model suites and for treatment of models as programs of next generation programming. The metamodel represented in Fig. 1 is a model of a model. Its components and its associations are expressed as attribute grammar rules. The compiler-compiler approach is enhanced by the layered handling of models according to Fig. 2. All components of a model in the model suite are explicit and become thus transformable to representation models and to programs of third or fourth generation of programming. This generation is the basis for language and platform independence of the models themselves. Modellers thus become programmers of next generation programming languages. The quality of the generated programs is therefore higher than a non-specialised programmer could achieve. Compilation allows the integration of standards. Model suite libraries become then the kernel for modelware. Models become directly executable.

Tested, verified, and validated approaches for MaP: The MaP approach is gradually developed in four application areas for which I and my collaboration partners have sufficient experience. It will be assessed, evaluated, analysed, questioned, scrutinised and generalised in such a way that I will open the path to an extension of the approach to other application space, to other CoP with different interest and intentions, to other problem spaces, and to other concept space. This extension will be developed in our scientific network.

References

References

[Ais88] H. Aiso. The fifth generation computer systems project. Future Generation Comp. Syst., 4(3), pp. 159-175, 1988.

[AAB+95] M. Albrecht, M. Altus, E. Buchholz, A. Düsterhöft, and B. Thalheim. The rapid application and database development workbench - A comfortable database design tool. Proc. CAiSE'95, LNCS 932, pp. 327-340. Springer, 1995.

[Ama16] S. Amarasinghe. Third software crisis - the multicore problem.
http://habitatchronicles.com/2016/10/software- crisis-the-next-generation/, 2016.

[ANT14] B. AlBdaiwi, R. Noack, and B. Thalheim. Pattern-based conceptual data modelling. Information Modelling and Knowledge Bases, vol. XXVI of Frontiers in Artificial Intelligence and Applications, 272, pp. 1-20. IOS Press, 2014.

[BCM+08] J. E. Burge, J. M. Carroll, R. McCall, and I. Mistrik. Rationale-based software engineering. Springer, 2008.

[BFCA14] N. Bencomo, R. B. France, B. H.C. Cheng, and U. Aßmann. Models@run.time: foundations, applications, and roadmaps. LNCS, vol. 8378. Springer, 2014.

[BKV17] D. Bork, D. Karagiannis, and J. Vanthienen, editors. Proc. PrOse 2017, vol. 1999 of CEUR Workshop Proceedings. CEUR-WS.org, 2017.

[BL74] J. Bormann and J. Lötzsch. Definition und Realisierung von Fachsprachen mit DEPOT. PhD and Advanced PhD thesis, Technische Universität Dresden, Sektion Mathematik, 1974.

[Bör07] E. Börger. Modeling workflow patterns from first principles. Proc. ER 2007, LNCS 4801, pp. 1 -20. Springer, 2007.

[BR18] E. Börger and A. Raschke. Modeling Companion for Software Practitioners. Springer, 2018.

[BT15] R. Berghammer and B. Thalheim. Methodenbasierte mathematische Modellierung mit Relationenalgebren. Wissenschaft und Kunst der Modellierung: Modelle, Modellieren, Modellierung, pp. 67 -106. De Gryuter, Boston, 2015.

[Dah97] A. Dahanayake. An environment to support flexible information modelling. PhD thesis, Delft University of Technology, 1997.

[dRE98] W. P. de Roever and K. Engelhardt. Data Refinement: Model-oriented Proof Theories and their Comparison, vol. 46 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[DT12] A. Dahanayake and B. Thalheim. A conceptual model for IT service systems. Journal of Universal Computer Science, 18(17):2452 -2473, 2012.

[DT15] A. Dahanayake and B. Thalheim. W*H: The conceptual model for services. Correct Software in Web Applications and Web Services, Texts & Monographs in Symbolic Computation, pp. 145 -176, Wien, 2015. Springer.

[ELP11] D.W. Embley, S.W. Liddle, and O. Pastor. Conceptual-model programming: A manifesto. In Handbook of Conceptual Modeling - Theory, Practice, and Research Challenges, pp. 3-16. Springer, 2011.

[Ers81] A. P. Ershov. The transformational machine: Theme and variations. MFCS 1981, LNCS 118, pp. 16-32. Springer, 1981.

[Far16] F.R. Farmer. Software crisis: The next generation. http://habitatchronicles.com/2016/10/software-crisis-the-next-generation/, Oct. 14 2016.

[FG11] R. Farahbod and U. Glässer. The CoreASM modeling framework. Softw., Pract. Exper., 41(2) pp. 167-178, 2011.

[FL07] P. Fettke and P. Loos, editors. Reference Modeling for Business Systems Analysis. Hershey, 2007.

[Fra11] U. Frank. Some guidelines for the conception of domain-specific modelling languages. Proc. EMISA 2011, vol. 190 of LNI, pp. 93-106. GI, 2011.

[Fra80] B. Van Fraassen. The scientific image. Clarendon Press, Oxford, 1980.

[GEPG18] F.D. Giraldo, S. Espana, O. Pastor, and W. J. Giraldo. Considerations about quality in model-driven engineering - current state and challenges. Software Quality Journal, 26(2):685-750, 2018.

[GGZ04] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). it - Information Technology, 46(5), pp. 265-276, 2004.

[GHL+85] R. Grossmann, J. Hutschenreiter, J. Lampe, J. Lötzsch, and K.Mager. DEPOT 2a Metasystem für die Analyse und Verarbeitung verbundener Fachsprachen. Technical Report 85, Studientexte des WBZ MKR/Informations- verarbeitung der TU Dresden, Dresden, 1985.

[GRW77] H. Ganzinger, K. Ripken, and R. Wilhelm. Automatic generation of optimizing multipass compilers. In IFIP Congress, pp.535-540, 1977.

[Hei04] H.D. Heilige, editor. Geschichten der Informatik: Visionen, Paradigmen, Leitmotive. Springer, 2004.

[Hof01] G. Hofstede. Culture's Consequences, Comparing Values, Behaviors, Institutions and Organizations across Cultures. Sage Publisher, Thousand Oaks, 2001.

[Hut86] J. Hutschenreiter. Zur Pragmatik von Fachsprachen. PhD thesis, Technische Universität Dresden, Sektion Mathematik, 1986.

[JMTV05] H. Jaakkola, T. Mäkinen, B. Thalheim, and T. Varkoi. Evolving the database co-design framework by SPICE. Informaton Modelling and Knowledge Bases Vol. XVII, Series Frontiers in Artificial Intelligence, vol. 136, pp. 268-279. IOS Press, May 2006.

[JT03] H. Jaakkola and B. Thalheim.Visual SQL - high-quality ER-based query treatment. Proc. IWCMQ'2003, LNCS 2814, pp.129-139. Springer, 2003.

[JT10] H. Jaakkola and B. Thalheim. A framework for high quality software design and development: A systematic approach. IET Software, pp. 105-118, April 2010.

[KBF+05] T. Kruscha, B. Briel, G. Fiedler, K. Jannaschk, T. Raak, and B. Thalheim. Integratives HMI-Warehouse für einen durchgängigen HMI-Entwicklungsprozess. Elektronik im Kraftfahrzeug 2005. 12. Internationaler Kongress Electronic Systems for Vehicles, no. 1907 in VDI-Berichte. VDI-Verlag, 2005.

[KD17] H. K önig and Z. Diskin. Consistency checking of interrelated models: long version. Fachhochschule für die Wirtschaft Hannover, 2017.

[Kie18] Website Kieler. Kiel Integrated Environment for Layout Eclipse Rich Client.
https://www.rtsys.informatik.uni-kiel.de/en/research/kieler, 2018. Accessed July 29, 2018.

[KMM16] D. Karagiannis, H. C. Mayr, and J. Mylopoulos, editors. Domain-Specific Conceptual Modeling, Concepts, Methods and Tools. Springer, 2016.

[Knu84] D. E. Knuth. Literate programming. Comput. J., 27(2) pp. 97-111, 1984.

[Knu86] D.E. Knuth. The METAFONTbook. Addison-Wesley, 1986.

[Kra18] F.F. Kramer. Ein allgemeiner Ansatz zur Metadaten-Verwaltung. PhD thesis, Christian-Albrechts University of Kiel, Technical Faculty, Kiel, 2018.

[KRT+16] S. Karg, A. Raschke, M. Tichy, and G. Liebel. Model-driven software engineering in the open project: project experiences and lessons learned.Proc. Model Driven Engineering Languages and Systems 2016, pp.238-248. ACM, 2016.

[Kru04] P. Kruchten. The Rational Unified Process - An Introduction, 3rd Edition. Addison Wesley object technology series. Addison-Wesley, 2004.

[KT08] S. Kelly and J. - P. Tolvanen. Domain-Specific Modeling - Enabling Full Code Generation. Wiley, 2008.

[KT16] F. Kramer and B. Thalheim. Holistic conceptual and logical database structure modelling with ADOxx. Domain-Specific Conceptual Modeling, Concepts, Methods and Tools, pp.269-290, Springer, 2016.

[KT18] Y. Kropp and B. Thalheim.Viewpoint-oriented data management in collaborating research projects. Models: Concepts, Theory, Logic, Reasoning, and Semantics, Tributes, pp.146-174. College Publications, 2018.

[Kuh70] T. Kuhn. The Structure of Scientific Revolutions. University of Chicago Press, Chicago, Illinois, 2nd, enlarged, with postscript edition, 1970.

[KWB06] A. Kleppe, J. Warmer, and W. Bast. MDA Explained: The Model Driven Architecture - Practice and Promise. Addison Wesley, 2006.

[Lam94] L. Lamport. LaTeX: A document preparation system. Addison-Wesley, 1994.

[Lan73] H. Langmaack. Übersetzerkonstruktion. Internes Skriptum zu Vorlesungen WS 72/73, SS 73 an der Universität des Saarlandes, FB Angewandte Mathematik und Informatik, 1973.

[Lev12] N.G. Leveson. Engineering in a safer world: System thinking applied to safety engineering systems. MIT Press, 2012.

[LSS11] M.S. Lund, B. Solhaug, and K.Stolen. Model-Driven Risk Analysis - The CORAS Approach. Springer, 2011.

[MGS+13] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? experiences from three industrial cases. Software & Systems Modeling, 12(3):619-639, 2013.

[MJ04] D. Marco and M. Jennings. Universal meta data models. Wiley Publ. Inc., 2004.

[MMR+17] H.C. Mayr, J. Michael, S. Ranasinghe, V.A. Shekhotsov, and C. Steinberger. Model centered architecture. Conceptual Modeling Perspectives, pp.85-104, Springer, 2017.

[MNS+13] Hui Ma, R. Noack, K.-D. Schewe, B. Thalheim, and Q. Wang. Complete conceptual schema algebras. Fundamenta Informaticae, 123:1-26, 2013.

[Mo82] T. Moto-oka, editor. Fifth generation computer systems. North-Holland, Amsterdam, 1982.

[Mül16] R. Müller. Geschichte des Systemdenkens und des Systembegriffs.
http://www.muellerscience.com/ SPEZIALITAETEN/System/systemgesch.htm, 2016.

[OMG17] OMG: Meta-object facility (MOF), core. Object Management Group TR, 2016.
www.omg.org/spec/MOF/.

[PKP14] R. F. Paige, D. S. Kolovos, and F. A. C. Polack. A tutorial on metamodelling for grammar researchers. Sci. Comput. Program., 96:396-416, 2014.

[PM07] O. Pastor and J. C. Molina. Model-driven architecture in practice - a software production environment based on conceptual modeling. Springer, 2007.

[Pod01] A.S. Podkolsin. Computer-based modelling of solution processes for mathematical tasks. ZPI at Mech-Mat MGU, Moscov, 2001. (In Russian).

[Pto18] Website PtolemyII. Ptolemy project: heterogeneous modelling and design.

http://ptolemy.berkeley.edu/ptolemyII/, 2018.

[RF87] G. Riedewald and P. Forbrig. Software specification methods and attribute grammars. Acta Cybern., 8(1):89–117, 1987.

[Run94] N. Runge. Scheme transformations on the basis of optimizing combinations of partially applicable elementary transformation methods. PhD thesis, Karlsruhe University, Computer Science Dept., 1994.

[Rus13] L. Rüschendorf. Mathematical risk analysis. Springer Ser. Oper. Res. Financ. Eng. Springer, Heidelberg, 2013.

[SGM02] C. A. Szyperski, D. Gruntz, and S. Murer. Component software - beyond object-oriented programming, 2nd Edition. Addison-Wesley component software series. Addison-Wesley, 2002.

[Sil01] L. Silverston. The data model resource book. Revised edition, vol. 2, Wiley, 2001.

[ST04] K.-D. Schewe and B. Thalheim. Structural media types in the development of data-intensive web information systems. Web Information Systems, pp.34-70. IDEA Group, 2004.

[Ste80] W. Steinmüller. Rationalisation and modellification: Two complementary implications of information technologies. IFIP Congress, pp. 853-861, 1980.

[SV05] T. Stahl and M. Völter. Model-driven software architectures. dPunkt, Heidelberg, 2005. (in German).

[Tha75] B. Thalheim. Theorie deterministischer kontextfreier Grammatiken. Diplomarbeit, Technische Universität Dresden, Sektion Mathematik, 1975 (In German).

[Tha00] B. Thalheim. Entity-relationship modeling - Foundations of database technology. Springer, Berlin, 2000.

[Tha04] B. Thalheim. Codesign of structuring, functionality, distribution, and interactivity. Australian Computer Science Comm., 31(6):3-12, 2004. Proc. APCCM'2004.

[Tha10] B. Thalheim. Model suites for multi-layered database modelling. Information Modelling and Knowledge Bases XXI, volume 206 of Frontiers in Artificial Intelligence and Applications, pp. 116–134. IOS Press, 2010.

[Tha17] B. Thalheim. General and specific model notions. Proc. ADBIS'17, LNCS 10509, pp.13-27, Cham, 2017, Springer.

[Tha18] B. Thalheim. Normal models and their modelling matrix. Models: Concepts, Theory, Logic, Reasoning, and Semantics, Tributes, pp. 44-72. College Publications, 2018.

[TN15] B. Thalheim and I. Nissen, editors. Wissenschaft und Kunst der Modellierung: Modelle, Modellieren, Modellierung. De Gruyter, Boston, 2015.

[Tro16] M. Tropmann-Frick. Genericity in Process-Aware Information Systems. PhD thesis, Christian-Albrechts University of Kiel, Technical Faculty, Kiel, 2016.

[TT11] M. Tropmann and B. Thalheim. Performance forecasting for performance critical huge databases. Information Modelling and Knowledge Bases, vol. XXII, pp.206-225. IOS Press, 2011.

[VM11] H. Vandierendonck and T. Mens. Averting the next software crisis. Computer, 44(4):88-90, 2011.

[Web95] B.F. Webster. Pitfalls of object-oriented development: a guide for the wary and entusiastic. M & T books, New York, 1995.

[Wir96] N. Wirth. Compiler construction. International computer science series. Addison-Wesley,1996.

[WVH17] H. Werthner and F. Van Harmelen editors. Informatics in the Future: Proceedings of the 11th European Computer Science Summit (ECSS 2015), Vienna, October 2015. Springer, 2017.