From Models_For_Programming To Modelling_To_Program and Towards Models_As_A_Program

Bernhard Thalheim[0000−0002−7909−7786]

Department of Computer Science, University Kiel, Germany
bernhard.thalheim@email.uni-kiel.de

Abstract. The history of programming languages can be separated into four or five generations. Most languages are nowadays at the level of the third or fourth generation. The fifth generation programme failed mainly due to the infrastructure that has been available at that time. We are going to revive this firth generation programming efforts by deployment of models as a source code for compilation of programs. Currently models are used as a blueprint or as some inspiration for programmers. At present, we are able to develop an approach of modelling to program. In future, we might have models as programs. This programme will then result in true fifth generation programming.
This Ansatz requires models at a higher quality level. Models at this level will become executable if they are precise and accurate. This paper summarises and discusses the vision that models will become programs in future. Models are used for programming as a communication mediator. Modelling as an art will be an integrative part of program development. Mastering modelling as technology will result in modelling as programming. In future, models will become itself programs. We present some of the main issues for research.

Keywords: models for programming, modelling to program, modelling as programming, models as programs, true fifth generation programming

1 The Vision: Next Generation Programming

Modelling can be considered as the fourth dimension of Computer Science and Computer Engineering beside structuring, evolution, and collaboration. Models are widely applied and used in everyday life and are widely deployed in our area. Typical scenarios are: prescription and system construction, communication and negotiation, description and conceptualisation, documentation, explanation and discovery for applications, knowledge discovery and experience propagation, and explanation and discovery for systems. Programming is currently based on intentional development models. Models are often explicitly specified. Sometimes they are implicit. We concentrate in this chapter on the first scenario, i.e. models as a means for construction. Generation zero of model usage is the starting point. The current state-of-the-art can be characterised as a movement from *Modelling-for-Programming (M4P)* where models are used as yet-another-development document for programming or throw-away-inspiration for the coding towards *Modelling_to_Program (M2P)* which is essentially a revival or reactivation or renaissance of modelling as an art. The main activity for M2P is model-based development. They are useful for developing the program in its essentials. But there are other ways to code without models.

We may distinguish three generations for this agenda for the system construction scenario after revival, reactivation, and enlivenment of models instead of only using models beside blueprint and inspiration:

First generation: **Modelling to Program** (M2P). Models can also be used as a source code for program generation. The result of such generation process may be enhanced by program refinement approaches. The model must be of a higher quality compared to current approaches. It must be as precise and accurate as appropriate for the programmer's task without loss of its explanation power. It must become adaptable to new circumstances. Maintenance of programs can be supported by maintenance of models. The first generation is thus based on *model-based development* and reasoning.

Second generation: **Modelling as Programming** (MaP). Modelling becomes an activity as programming nowadays. Some models can directly be used as an essential part of sources for programs, esp. executable programs. They are neatly integratable with other source codes. Modelling languages have reached a maturity that allows to consider a model at the same level of precision and accuracy as programs. Models are validated and verified and as such the basis for program validation and verification. Models will become adaptable and changeable. Computer Engineering incorporates modelling.

Third generation: **Models as a Program** (MaaP). Models can be directly used as a source code for or instead of programs. If a separation of concern approach is used for model development then the source for programs consists of a *model suite* with well associated sub-models. Compilers will translate the model or the model suite to executable programs. Maintenance of programs is handled at the model level. Models can be directly translated as programs, i.e. programming can be performed almost entirely through modelling. Third generation modelling is then *true fifth generation programming* which essentially frees the program developer from writing third or fourth generation programs. Everybody – also programming laymen and non-programmers – who can specify models will become a programmer.

The current-state-of-the-art is going to be analysed in the next section. The following sections discuss then our vision for the second and third generation. Figure 1 visualises the vision towards models as programs with the current situation, towards modelling to program and modelling as programming, and finally models as a program.

The final stage of model-based reasoning for system construction will be the usage of models as a program. We shall then partially replace programming by models. The transformation of the model suite to a program (MaaP) will also

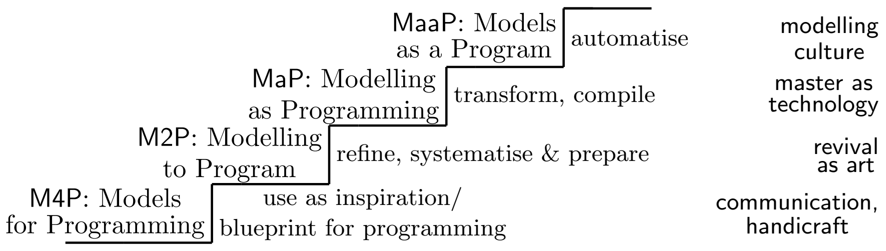

MaaP: Models as a Program
automatise
modelling culture
MaP: Modelling as Programming
transform, compile
master as technology
M2P: Modelling to Program
refine, systematise & prepare
revival as art
M4P: Models for Programming
use as inspiration/ blueprint for programming
communication, handicraft

Fig. 1. From current state-of-art to next generation programming as model-based thinking

be based on a compiler. Programming will be mainly modelling. In this case, we may consider modelling as true fifth generation programming.

The Storyline of this Vision Paper

This paper presents the vision for first, second and third generation modelling as an outline for a series of workshops. Visions are visions. We thus need to sketch the path towards a new way of programming that is called true fifth generation programming (5PL) in Section 5. This paper aims at introducing the entire M2P, MaP, and MaaP programme. We abstain from long citation lists in the next sections. Subsection 3.2 summarises the large body of knowledge that is already available.

The current Ansatz in Computer Science is to utilize models as an blueprint for programming or at least as inspiration. Models can be thrown away after programs have been developed. The software crisis, the data crisis, and the infrastructure crises show that the model_for_programming approach must be revised. Section 2 discusses well-known approaches to model-based development, its issues, and its problems. Typical specific approaches are model-driven development, conceptual-model programming, model-driven web engineering, and models@runtime. We realise that model-based development is still often essentially model-backed development. For this reason, Section 3 revisits the current state-of-art. The body of knowledge already supports to step towards modelling to programming. There are tools available. These tools may be combined with heritage programming. Since a model typically focuses on some aspects on certain abstraction level, we have to use a collection of well-associated models, i.e. a model suite. Model suite also enable in layered model-based development.

Section 4 discusses features, approaches, and ideas for modelling to program and furthermore for modelling as programming. These modelling approaches can be supported by literate modelling that uses model suites. Section 5 discusses then some of the deliverables we envision for true fifth generation programming. We might use layering for generation of programs based on a suite of normal models and a landscape initialisation based on corresponding deep models. This approach will be discussed mainly on the experience we have learned with the separation into initialisation and specification. This approach is a typical onion Ansatz that has already been used for LaTeX. We thus generalise this onion Ansatz and discuss its potential. We use two case studies from [50, 65, 74, 99]. for the discussion. A summary is sketched in the final Section 6.

2 Current Approaches to Model-Based Development

Models are used as instruments in many reasoning and especially engineering scenarios. The reasoning and development process is then model-determined. Depending on the function that a model plays in a programmer's scenario we distinguish three general roles for models: (1) models are instruments for reasoning which implies their prior construction and the reasoning necessary for their construction; (2) models as targets of reasoning; (3) models as a unique subject of reasoning and its preliminary. These roles have to be supported by sophisticated reasoning mechanisms such as logical calculi. Modelling uses beside the classical deductive reasoning also other reasoning approaches such as Programming and reasoning must be supported by a mechanism, e.g. logical calculi with abduction or induction.

2.1 M4P: Models for Programming

Model-based reasoning is an essential feature of all mental models such as perception models which are representing some (augmented) reality and domain-situation models which are representing a commonly agreed understanding of a community of practice. Model-based reasoning supports modelling based on data similar to inverse modelling. It comprehends the background of each model considered. It allows to consider the corresponding limitations and the obstinacy of models, esp. modelling languages. As a reasoning procedure, it is enhanced by well-formed calculi for reasoning. Model-based reasoning is compatible with all kinds of development procedures (water, spiral, agile, extreme). It also allows to handle uncertainties and incompleteness of any kind.

The model-for-programming approach uses models as some kind of blueprint and as a description mediator between the necessities in the application area and the program realisation. After programming has

Model-based development and engineering is a specific form of engineering where the model or model suite is used as a *mediator* between ideas from the application domain and the codification of these ideas within a system environment. It has been considered for a long time as a *greenfield* development technique that starts with requirements acquisition, elicitation, and formulation, that continues with system specification, and terminates with system coding. Models are used as mediating instruments that allow to separate the description phase from the prescription phase. Engineering is, however, nowadays often starting with legacy systems that must be modernised, extended, tuned, improved, etc. This kind of *brownfield* development may be based on models for the legacy systems and on macro-models representing migration strategies that guide the system renovation and modernisation. The heritage model (or legacy models) is used as an origin for a sub-model of the target system.

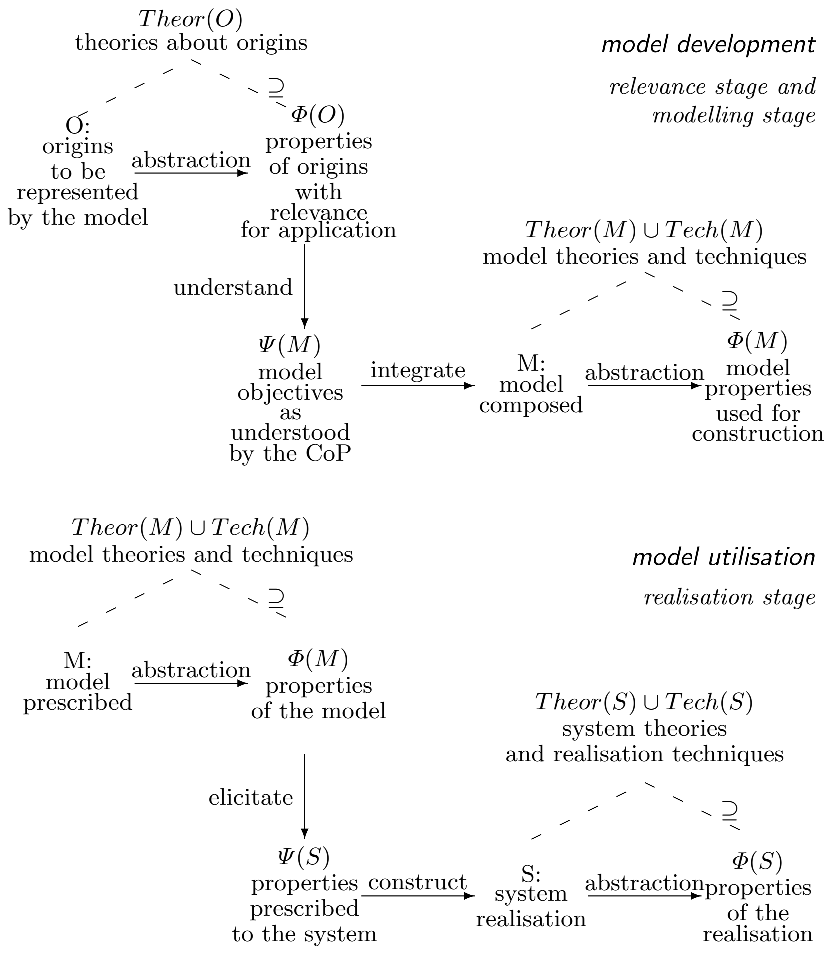
The four supporting means for model-based engineering are the modelling know-how, the experience gained in modelling practices, the modelling theory, and finally the modelling economics. The last two means, however, need a deeper investigation. Specific forms of model-based reasoning for system construction are, for instance,

– *model-driven architectures and development* based on a specific phase-oriented modelling mould,
– *conceptual-model programming* oriented on executable conceptual models,
– *models@runtime* that applicabies models and abstractions to the runtime environment,
– *universal applications* based on generic and refinable models and with generators for derivation of the specific application,
– *domain-specific modelling* based on domain-specific languages,
– *framework-driven modelling* (e.g. GERA or MontiCore),
– *pattern-based development* based on refinable pattern,
– *roundtrip engineering* supported by tools that maintain the coexistence of code and model,
– *model programming* where the model is already the code and lower levels of code are simply generated and compiled behind the scenes,
– *inverse modelling* that uses parameter and context instantiation for model refinement,
– *reference modelling* that is based on well-developed and adaptable reference models, and
– *model forensics* which starts with model detection through analysis of the code, i.e. the model origin is the code.

These approaches develop models by stepwise refinement of the root or initial model, by selection and integration of model variations, and by mutation and recombination of the model. Models are typically model suites.

The mediator function of models is illustrated for 'greenfield' system construction in Figure 2. System construction is a specific kind of modelling scenario that integrates description and prescription. It might also be combined with the conceptualisation scenario. B. Mahr pointed out that origins also come with their theories. We may add also the deep model behind the origin.

During a *relevance stage* (or cycle), we first reason about the ways of operating in an application. This step may also include field testing of current artifacts within the given environment. Next we select, reason about, and revise those properties $\Phi(O)$ that are of relevance for system construction. The next phase is based on classical requirements engineering. We identify requirements (or business needs). These requirements become objectives $\Psi(M)$ which must be satisfied by the model. These objectives are biased by the community of practice (CoP).



$Theor(O)$
theories about origins
O: origins to be represented by the model
abstraction
$\Phi(O)$ properties of origins with relevance for application
understand
$\Psi(M)$ model objectives as understood by the CoP
integrate
$Theor(M) \cup Tech(M)$ model theories and techniques
M: model composed
abstraction
$\Phi(M)$ model properties used for construction
model development
relevance stage and modelling stage
$Theor(M) \cup Tech(M)$ model theories and techniques
M: model prescribed
abstraction
$\Phi(M)$ properties of the model
elicitate
$\Psi(S)$ properties prescribed to the system
construct
$Theor(S) \cup Tech(S)$ system theories and realisation techniques
S: system realisation
abstraction
$\Phi(S)$ properties of the realisation
model utilisation
realisation stage

Fig. 2. Straightforward model-based system development starting with description models which are transformed to prescription models for system realisation (modified from [96])

The *modelling stage* starts with the objectives as they are understood in the community of practice. Modellers compile and integrate these objectives in a model (or model suite). The 'ways of modelling" is characterised

– by the *modelling acts* with its specifics,
– the *theories and techniques that underpin modelling acts*,
– the *modellers* involved into with their obligations, permissions, and restrictions, with their roles and rights, and with their play;
– the *aspects* that are under consideration for the current modelling acts;
– the *objectives* that are guiding the way of modelling;
– the *resources* that bias the modelling act.

The *realisation stage* uses prescription properties as a guideline for construction the system. Model-based engineering is oriented on automatic compilation of the entire system or at least of parts of it. If the system is partially constructed and thus must be extended then the compilation must provide hooks in the compiled code. These hooks allow to extend the system without revising the rest of the compiled code.
Validation compares the properties of origins $\Phi(O)$ with the properties of the model $\Phi(M)$. *Verification* compares the properties of model $\Phi(M)$ with the properties of the system $\Phi(S)$. *Model assessment* compares the model objectives $\Psi(M)$ with the properties of the model $\Phi(M)$. *System assessment* compares the system objectives $\Psi(S)$ with the properties of the system $\Phi(S)$.

2.2 Model-Driven Development

In general, model-driven development is nothing else than a very specific form of the model development and usage process. We use only the last three in a specific form. The first two levels are assumed to be given.

Essentially, model-driven development (MDD) does not start with a computation-independent model. The process starts with modelling initialisation. The CIM is already based on a number of domain-situation models and some insight into the application situation or a model of the application situation. These models are the origins of the models under development and are then consolidated or toughened. The CIM describes the system environment, the business context, and the business requirements in a form that is used by the practitioners in the application domain. This model becomes refined to the PIM by services, interfaces, supporting means, and components that the system must provide to the business, independent of the platform selected for the system realisation. The chosen platform forms the infrastructure for the system realisation. The PIM is then refined to a platform-specific model PSM. In general, a number of models are developed, i.e. a model suite as shown in Figure 3.

The main concept behind this approach is the *independence* of the models on the platform respectively on the algorithmics setting. This independence allows to separate the phases. A system is then specified independently of the software execution platform and independently of the chosen set of algorithms.

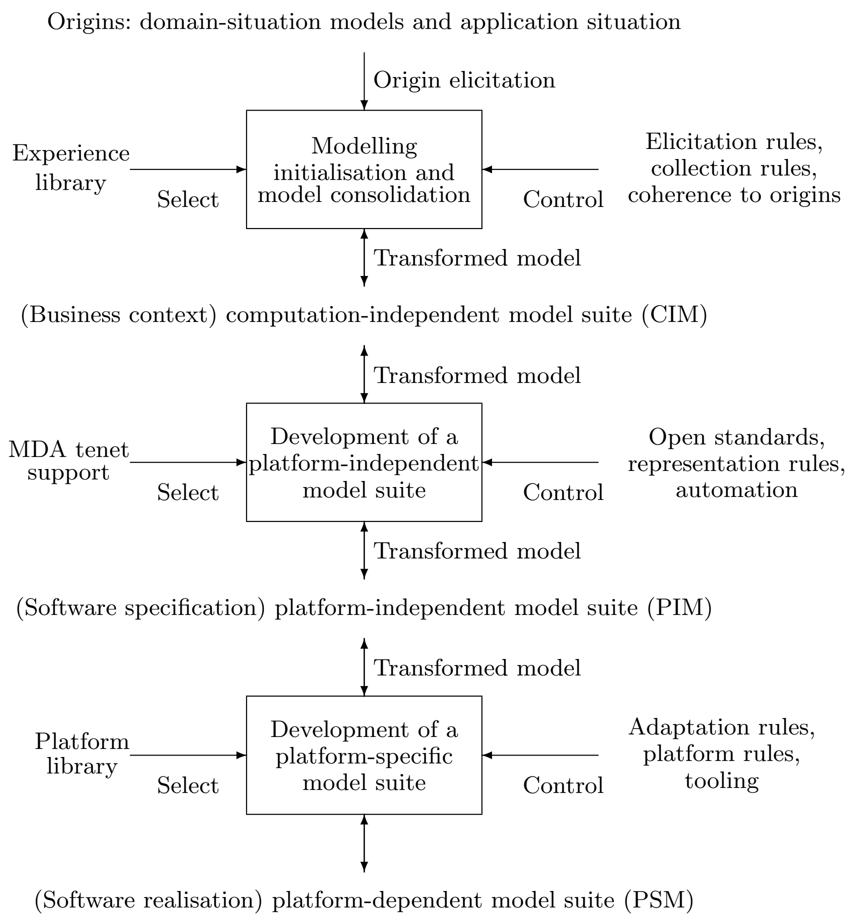

Origins: domain-situation models and application situation
Origin elicitation
Experience library
Select
Modelling initialisation and model consolidation
Control
Elicitation rules, collection rules, coherence to origins
Transformed model
(Business context) computation-independent model suite (CIM)
Transformed model
MDA tenet support
Select
Development of a platform-independent model suite
Control
Open standards, representation rules, automation
Transformed model
(Software specification) platform-independent model suite (PIM)
Transformed model
Platform library
Select
Development of a platform-specific model suite
Control
Adaptation rules, platform rules, tooling
(Software realisation) platform-dependent model suite (PSM)

Fig. 3. The three phases in model-driven development for construction scenarios

The transformation is typically a two directional. Model-driven architectures are based on three tenets: direct and proper representation of concepts and terminology in the application domain, automation for generation of routine components instead of human interaction, and open standards promoting reuse and proper tool deployment.

Model driven development assumes that the models at each phase can be transformed to each other. This rigid requirement cannot be used in many construction scenarios. For instance, database model development is based on forgetful mappings. The conceptual model contains conceptual information that is irrelevant for the "logical" or "physical" models and thus neglected during development.

2.3 Conceptual-Model Programming

Conceptual-model programming uses a compiler approach to programming. The model will be the source code for the compiler or transformer. The result of compilation should be then code which might be used for another compilation. The compiler assures that the model might already be used as (the final) code. The execution of the code corresponds to the conceptual specification.

Advanced conceptual-model programming is based on three theses:

- Conceptual-modelling languages must provide compiler facilities that allows to directly execute models.
- Conceptual-modelling languages must support co-design of database structure, of user interfaces, and of database access and user interaction.
- "The model-compiled code is beyond the purview of conceptual-model programming programmers — both for initially creating the application system being developed and for enhancing or evolving the application system." [31].

As a consequence, application-system development becomes entirely model-driven. Conceptual-model programming constitutes model-complete software development.

This approach requires that models are of high quality.

- *Models are complete and holistic.* The conceptual model supports all necessary co-design elements such as structuring, behaviour, and user support. The model itself can be a model suite.
- *Models are conceptual but precise.* All model elements must be precisely defined through proper element definition. At the same time, these elements have to be properly associated to the concept space that is used for conceptualisation of the model. Parsimony and economy of model elements guide the notation.

Provided that the model has high quality then evolution of an application can be directly represented through evolution of the model. It, thus, asserts that conceptual-model programming is essentially programming. The model is the kernel of code that can be easily adapted to the specific platform requirements.

2.4 Model-Driven Web Engineering

Model-driven development and engineering can be based on generic models. These models ease the selection of the normal model in dependence on a computation-independent model suite. They are also early serving as an additional entry at the modelling initialisation phase (Figure 3) and at the landscape (Figure 4) determination. The problem space, focus, scope, and codified concepts will be set through the utilisation of such generic models. Additionally, generic models will be used at the extrinsic (source) reflection layer (Figure 4). Generic models and reference models can be refined by the data and other information on hand.

Website development and web information system development is a typical example of model-based reasoning on the basis of generic models and of refinable specifications. Website construction benefits from the usage of previous models and programs, esp. generic ones. These generic models compile experience gained in similar web engineering projects. This experience elicitation, evaluation, appreciation, and acquisition is a specific *rigor stage* (or cycle). Generic models stem from models that have been already used. The generalisation procedure that led to the generic model allows to derive the specific refinement mechanisms for mapping the generic model to the old specific one. Generic models come then with a *refinable specification* and with a *refinement calculus*.

Model-based reasoning and website model-based development is based on the specification language SiteLang (see below). Website development is page-oriented where pages are essentially media objects with their structuring, their functionality (especially navigation, import/export, search), their runtime adaptation features to actors or users, their databases support, and their specific navigation-oriented flow of work. Websites can be categorised, e.g. business websites for a business with customer collaboration according to the used business culture. This categorisation is the basis for generic model suites of the website. The normal model can be then derived from the generic model and the computation-independent model.

2.5 Models@Runtime

Most model-driven development and most-driven architecture are concerned with the development process. Models@Runtime take a different turn towards support of software after it has been installed. Beside model-based evolution support, models are additionally developed on top of the code. The code will then be the origin. The model suite reflects essential properties of a system. This approach aims at development of proper performance models that characterise the software and its behaviour.

The Model@Runtime approach extends the applicability of models and abstractions to the runtime environment. This approach provides means for handling complexity of software and its evolution if effective technologies exist for code analysis and change in a given environment. Models are used for reasoning about runtime behaviour, for appropriate adaptation of systems at runtime, and for capturing runtime phenomena in systems such as time, memory, energy, location, platform, and personalisation concerns. Models provide meta-information, for instance, in order to automate runtime decision-making, safe adaptation of runtime behaviour, and optimal evolution or modernisation of systems. Modern systems are increasingly complex. This complexity is a challenge that needs to be managed especially in the case of evolving hybrid infrastructures consisting of a manifold of heterogeneous services, resources, and devices.

Typical tasks are the following ones: (a) creating or updating the models suite that represents a system according to evolution of the system or to changes in system's environments; (b) developing appropriate adaptation strategies; analysing and maintaining model suites while parts of the corresponding systems are changing; (c) propagating changes in the model suite back to the system; (d) adaption of emerging systems and their model suites in ways that cannot be anticipated at model development and system programming time; (e) enabling features supporting continuous design, deployment, and quality of service analysis for performance optimisation; (f) optimisation and tuning; (g) reducing uncertainty that arise due to environment changes, due to system integration and migration, due to changes of quality concerns, and due to changes and extensions of the system user communities and evolving viewpoints.

We envision that the modelling-as-programming approach allows to solve some of these challenges. Models@Runtime are an example of model-based reasoning despite the classical system construction scenario. Models are used for exploration, for exploration, for discovery of obstacles, for observation, and improvement of current solution, i.e. the scenarios targeted on system modernisation beside model-based development. It integrates also model checking.

2.6 Lessons Learned with Model-Based Development

Model-driven development highly depends on the quality of models. Although model-driven development will be supported by compilers in the future, it is currently mainly using an interpreter approach. Models must be doubly well-formed according to well-formedness criteria for the model itself and well-formed for proper transformation in the interpreter approach. For instance, database schemata have to be normalised. BPMN diagrams must be well-formed in order to be uniquely interpretable. Otherwise, the interpreter approach will not result in models that have a unique meaning. Model-based development has at least four different but interleaved facets: (i) model-driven development with stepwise adaptation of the model (suite), (ii) model-driven programming as conceptual-model programming, (iii) model-based development with generic or reference models as the starting point, and (v) model-based assessment and improvement of programs. The interpreter approach can be sufficient in the case of high-quality models. The classical approach to database modelling that uses a normalisation after translation, however, shows that compiler approaches are better fitted to model-based reasoning and development and to Modelling as Programming. If a model suite is going to be used as the basis for model-based development then the models in the model suite must be tightly associated. Tracers and controllers for associations among sub-models and coherence within a model suite are an essential prerequisite for proper model-based development. Often model suites are only loosely coupled. A compiler has to support sophisticated integration and harmonisation in the last compiler phase what is also a theoretical challenge.

3 Revisiting The State-Of-Art

3.1 Currently: Model-Backed Development

Model-based development, model-based design, and model-based architecture use a world of models. So far, we did not reach the stage that models are used for generation of programs. Programming is rather based on blueprint models or inspiration models. These models will not be changed whenever the program is under change. They are then throwaway origins in the programming process. Development is therefore model-backed. The potential for model-based development is, however, obvious. Software engineering is now paying far more attention to the needs, cultures, habits, and activities of business users. Users have their own understanding of the application that must be harmonised with the perception of developers.

Model-based development is currently revitalised [1], e.g. in the MontiCore project (see, for instance, [44]). The revival led to a new stage in generative software engineering for domain-specific languages which reflect the worlds of business users. In general however, modelling is still not considered to be a mandatory activity for programmers. It is still considered to be some kind of luxury.

We further observe that model-backed development has already been applied directly with the beginning of programming. Programmers have had their models. These models have, however, been implicit and have been stated rather seldom. Changes in the software did not use the models behind. They rather led to additional hidden models. These implicit models became hidden legacy since documentation of software has been and is still a big issue and is often completely neglected. The explicit model-backed development became important with the advent of the internet software and the turn towards user-oriented software.

3.2 The Body of Knowledge

Our approach to model-based reasoning is based on [20, 69, 77]. Figure 2 follows the reconsideration in [96] of the work by B. Mahr [70] Other variants of model-based development are conceptual-model programming and model driven architectures [31, 81], universal applications with generators for derivation of the specific application [78], pattern-based [6], and many other project like the CodeToGo project. SPICE and CMM added to this approach quality issues and matured co-design methodology [10, 36, 48, 86, 95]. Model-driven development, engineering and architecture (MDD, MDE, MDA) taught some valuable lessons reported about model-driven approaches, e.g. [27, 37] and the list in [108]. Model-driven development can be extended by literate programming [60], database programming [88], programming with GibHub [52], 'holon' programming [29], refinement [13], multi-language approaches [33], and schemata of cognitive semantics [66]. Projects like the Axiom project [25], the mathematical problem solver [83], and RADD (Rapid application and database development) [7, 92, 94] show how MaP can be accomplished.

An essential method for model-based programming is based on refinement styles. A typical refinement style is practised in the abstract state machine approach [15, 12]. A program is specified in a rather abstract block-oriented form based on changeable functions. These function can be refined to more detailed ones without context dependence to already specified ones.

Our vision approach can also be based on interpreters instead of compilers or compiler-compilers. Typical examples are [24, 23] using MetaCASE [55, 54], Come-In-And-Play-Out [43] based on algorithmics [19, 42], and model-driven web engineering approaches (beside [87], for instance, Hera [45], HDM [39], MIDAS [107], Netsilon [76], OOHDM [89], OOWS [80] RMM [47], UWE [61], WAE2 [21], Webile [85] WebML [18], WebSA [73], W2000 [8], and WSDM [104]). The interpreter approach is useful in the case of relatively simple modelling languages.

The interpreter approach to partially (and fragmentary) program generation can be applied as long as languages are strictly layered and there is no dependence among the layers. Optimisation is not considered. First interpreter approach to database structuring followed this approach for the entity-relationship model (based on rigid normalisation of the source schema before interpreting with attribute, entity, relationship layers whereas the last one allows very simple cardinality constraints). Constraint enforcement is a difficult problem which requires compilation, denormalisation, and specific supporting means.

The compiler approach [82, 110] allows to generate proper programs. The rule-based approach to database schema transformation in [63] extends [32]. It uses the theories of extended entity-relation modelling languages [94] and the insight into techniques such as web information systems [87] and BPMN semantification [14, 16]. We use advanced programming techniques and theories like attribute grammars [26], graph grammars [30, 94], database programming through VisualSQL [49], performance improvement [102], and normalisation techniques like those for storyboards in [74].

The fifth generation computer project [2, 3, 34, 75, 106, 105] inspired our approach to modelling as programming. We base our changes on advices by H. Aiso [4] who chaired the architecture sub-committee in the Japanese fifth generation computer project.

3.3 Experience Propagation Through Reference Models

Computer Engineering developed a rich body of successful applications that form a body of experience. This experience can also be used in a completely different development style instead of 'greenfield' or 'brownfield' development. Already by investigating so-called 'legacy' systems, we have had to realise that solutions incorporate tacit knowledge of programmers. We should therefore call these older solutions better *heritage* since they allow us to inherit the skills of generations of programmers despite changing hardware and software. Heritage systems provide a rich body of already available solutions.

A typical direction of heritage system development are reference models. Reference models [9, 35, 71, 101, 91] are generalisations of existing successful system solutions. The experience gained in such applications can be generalised to classes of similar solutions. The generalisation procedure allow to invert to generalisation to a specialisation to the given solution. Universal programming [78] and generic solutions [103] use generation facilities for deriving an essential part of a solution from already existing more general ones.

3.4 Tools for Model-Based Development

The fundamental idea behind MetaCASE and its incorporation to a sophisticated tool support for modelling [23] is the separation of models from their visual representations. MetaCASE is a layered database architecture consisting of four OMG layers: signature, language, model and data.

The computational environment for the approach can be based on systems ADOxx, Eclipse, Eugenia, GMF, Kieler, mathematical problem solvers, MontiCore, and PtolemyII [53, 28, 44, 56, 62, 83, 84]. The two case studies mentioned below have been discussed in [50, 65, 74, 99]. More examples on models in science are discussed in [98].

The compiler-compiler approaches [17, 41, 46, 68] are far more powerful. They have been developed for domain-specific languages (at that time called 'Fachsprachen') since 1973. This approach is our main background for modelling as programming. It can be combined for 'brownfield' development (migration, modernisation) with the strategies in [58]. The directive and pragma approaches have been developed already for FORTRAN and have been extensively used for C, C++, and especially ADA [93]. The layered approach used in this chapter follows the realisations already known and widely applied for programming languages since COBOL and ALGOL60. Layering is also the guiding paradigm behind LaTeX and TeX [59, 67] with a general setup layer, the content layer, the adaptable device-independent layer, and the delivery layer. The compiler-compiler approach additionally integrates generic models [11, 87, 100, 103], reference libraries [35, 101], meta-data management [64], informative models [97], model-centric architectures [72], and multi-level modelling [38].

3.5 The Background: Model Suites

Modelling should follow the *principle of parsimony*: Keep the model as simple and as context-independent as possible.

Models that follow this principle provide best surveyability and understandability. Context-dependence would otherwise require to consider all other models and origins together with the model. The result would be a model that is not really ready for use.

Another principle is the *separation of concern*: Instead of considering a holistic model which allows to consider all aspects and functions, we use a number of models that concentrate on few aspects and support few functions.

This second principle requires consideration according to aspects such as structure, functionality, behaviour, collaboration as the triple (communication, cooperation, coordination), interactivity, infrastructure, storage, and computation. The classical co-design approach to database system development follows this principle by integrated consideration of database structure models, functionality and behaviour models, viewpoint models, and realisation models. The principle can also be based on the W*H separation [24], i.e who (relating to user profile, habits, personal culture), where (wherein, wherefrom, which orientation), what (whereof, wherefore, wherewith), why (whence, whither, for what, whereto), in what way (how, by what means, worthiness, which pattern, on which basis), when (at what moment and time frame), whom (by whom, whichever), and whereby (which enabling system, infrastructure, languages with their own obstinacy and restrictions ...). Separation of concern enables to consider from one side inner and implicit models as deep models and from the other side outer and explicit models as normal models. Deep models are typically stable and will not change. Normal models are more tightly associated with the origins that are really under consideration.

Following the two principles, we use a collection of models. The entire picture is then derived from these models similar to the global-as-view approach [111]. These models do not need to be entirely integrated. It is only required that the models in the collection are coherent. Utilisation of several models requires support for co-existence, consistency among abstractions, and integration of deep and normal models. Models in such collections or ensembles are also partially governing other models, i.e. we can use a synergetic separation into master and slave models. This separation provides a means to use control models for master models. We do need control models for slave models.

Coherence becomes then a main property of such model collection. This coherence must be maintained. The simplest way for support of coherence is to build explicit associations among these models. Associations can be build on general association schemata. In this case we can develop tracers and controllers for the model association. The association represents then the architecture of the model collection. Associations among models may also based on association among sub-models or on substitutability of a submodel by another submodel or on composers of models.

A *model suite* is a coherent collection of models with an explicit association schema among the models, tracers for detection of deviations from coherence, and controllers for maintenance of the association.

3.6 The Trick: Layered Model Development

The model suite in Figure 4 will be layered into models for initialisation and landscaping, for strategic intrinsic set-up, for tactic extrinsic reflection and definition, for customisation and operationalising including adaptation, and for model delivery. Figure 4 displays layering for greenfield development. Brownfield development is based on revision for modernisation, integration, evolution, and migration. It uses also reengineering of models that become additional origins with their normal models, deep models, mentalistic and codified concepts. Heritage development does not follow layering in this way.

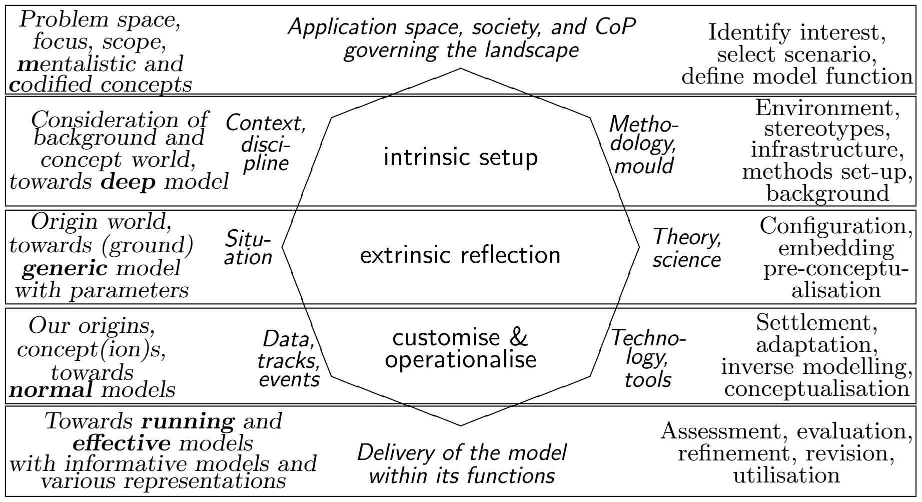

Problem space, focus, scope, mentalistic and codified concepts
Application space, society, and CoP governing the landscape
Identify interest, select scenario, define model function
Consideration of background and concept world, towards deep model
Context, disci-pline
intrinsic setup
Metho-dology, mould
Environment, stereotypes, infrastructure, methods set-up, background
Origin world, towards (ground) generic model with parameters
Situ-ation
extrinsic reflection
Theory, science
Configuration, embedding pre-conceptu-alisation
Our origins, concept(ion)s, towards normal models
Data, tracks, events
customise & operationalise
Techno-logy, tools
Settlement, adaptation, inverse modelling, conceptualisation
Towards running and effective models with informative models and various representations
Delivery of the model within its functions
Assessment, evaluation, refinement, revision, utilisation

Fig. 4. The layered approach to model suite development and program generation (revised and modified from [50])

We use the W*H characterisation for *landscaping* as initialisation of a layered model suite. Figure 4 separates from one side supporters such as sources, grounding and basis on the left side and from the other side enablers such as methodology, theories, techniques on the right side.

4 Model-Based Thinking

Model-based thinking is going to integrate modelling into programming. Models can be used for 'semantification' of programs. Model-based thinking is different from programming since it uses models of humans together with models of the computing infrastructure. These models can be oriented on the way of programming, controlling, and supporting infrastructure. The way of modelling allows to concurrently consider several variants and alternatives for a solution, to use preference assessment for 'good' solutions, and to separate concerns by well-associated with each other models.

Models are used in everyday life as mental prospect or idea. In this case they are less precise than those used in system construction scenarios. Program development needs a higher level of accuracy and precision. We concentrate here on high-quality models, i.e.

· with potential extensibility, adaptability, replaceability, efficiency, stability, validatable, testable, analysable, exactness, error-proneness, reliable spacial and temporal behaviour, dependability, maintainable, etc. from one side of external quality and
· with inner coherence, toleratable redundancy, constructivity, modularisation, coherence, and efficiency, etc. from the other side of internal quality.

4.1 The Modelling Method for Programming

Let us briefly remember generations of programming languages and derive that M2P is a natural and obvious continuation of the development of programming languages. First generation programming was oriented on executable processor-dependent code. The second generation brought processor independence and introduced symbolic and mnemotic code. It provided thus some hardware independence. Assembling programs were interpreted by an assembler. They have been oriented on the *way of controlling* the computation. Both generations allowed a very efficient and effective full treatment of all processes performed at the computer. At the same time, programming was rather low level machine-oriented programming and, thus, a challenging task for well-educated specialists.

The third generation languages standardised Von Neumann constructs to higher level constructs that can be used within a standardised structure of programs. These constructs are context-reduced and exempted from the plague of firmware dependence. The languages are oriented on an abstraction of the *way of computing* (or working) of computers. Syntax and partially semantics of languages became standardised and well-defined. Meanwhile, these constructs are near to human languages. Third generation programming became somehow independent on the enabling infrastructure. Structuring of programs has been standardised and component-backed. The main enablers for third generation languages are compilers, debuggers, linkers, and editors. Compilers incorporated also optimising programs for generation of efficiently executable code at the level of second or first generation languages. Programming could be thus performed by everybody who got computer engineering education.

Fourth generation languages provided implementation and groundware independence, e.g. data storage and management independence and management system independence. These languages are macro-based instead of command-oriented and are often script languages. Optimisation is now available for a bunch of concurrently executed programs. Groundware independence can nowadays also provided in networked computation. Third and fourd generation programming can be based on a number of different paradigms such as object-orientation, parallel computing, functional programming, imperative or procedural programming, symbolic computation, collection-oriented (e.g. set- or bag-oriented) programming, event-driven programming, and declarative programming.

Fifth generation languages raised the hope for program development in computational environments that could 'think' for themselves and draw their own inferences using background information. The approach was mainly based on representing programs as massives of formulas in first-order predicate logics.

One of the reasons that this program failed is this restriction to first-order predicate logics. Other reasons are: "it was highly dependent on AI technology, it did not achieve an integration of AI and human-computer interface techniques, ... 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." [50]

Our vision is to integrate the way of programming and the way of modelling. Models are typically model suites and allow to concentrate on certain aspects while not loosing the association among the models in the suite. This approach decreases the complexity of reasoning to a level that a human can easily capture it. Since the model suite is integrated, this reduction does not mean to loose the entire view on a program. Model-based thinking is also based on approximate reasoning since a model can be an abstraction of another more precise or more accurate model. Models incorporate their explanations, the necessary knowledge from the application domain, and the reasoning and thinking approaches of users. If models and model suites can be transformed to programs then maintenance of such programs can be transferred back to maintenance odf models.

4.2 The Near Future M2P: The Modelling_To_Program Initiative

M2P is based on a consolidation of the body of knowledge for models, modelling activities, and modelling. The renaissance of modelling towards a real art of modelling allows to use modelling as a matured practice in Computer Science and Computer Engineering, esp. for system development and software system development. The previous chapters contributed to this development. M2P can be understood as a technology for model-based development and model-based reasoning.

Model-based reasoning and thinking is the fourth dimension of human life beside reflection and observation, socialising and interacting, acting and fabricating (e.g. engineering), and systematising and conceptualising (typical for scientific disciplines). Programs reflect a very specific way of acting with a machine. Programming is so far bound to Von Neumann machines. It will be extended to other machines as well, e.g. to interaction machines [40, 109]. Model-backed development coexists with programming since its beginning, at least at the idea and comprehension level. Documentation or informative models are developed after programming. Modelling_to_program is coordinated modelling within program development and design processes. It enhances programming, contributes to quality of programs and enables in model-oriented and purposive maintenance, change management, and system modernisation. Model and systems complement one other using separation of concern and purpose. Non-monolithic programs and systems use a variety of languages. Their coherence and integration is far simpler if we use model suites and if programs are also following architecture principles similar to the associations in the model suite.

M2P requires high-quality models. Models must be precise and accurate. They must not reflect the entire picture. They can be intentionally incomplete.

These models may use dummies and hocks for code injection by the translator, e.g. similar to stereotype and pattern refinement [5]. Models may already contain directives and adornments as hints for translation. In most cases, we use an interpreter for transformation. A typical example is the ADOxx+ transformation for HERM++ interpreters discussed in [63]. In this case, views are represented by dummies. Late logical code optimisation is necessary after translation. M2P requires refinement and adaptation of the translated code. Models intrinsically incorporate deep models. Modelling is then mainly normal model development. An early variant of this approach is the RADD toolbox for database structure design and development [94]. Website generation [87] is another typical example of this approach.

Modelling has now reached the maturity to properly support programming at the intentional level. It is also going to be used at the meta-reasoning level [22] for guiding methodologies and methods. Modelling becomes an enabler for development of large and complex systems. It is already a commonly used technique for user-oriented and internet-based web information systems [87]. These systems support a wide variety of user frontend systems according to business user's needs. They integrate a number of different backend systems.

Modelling_to_program does not mean that programs have to be developed only on the basis of models. Models can be the source or pre-image or archetype or antetype of a program. Still we also program without an explicit model. M2P eases programming due to its explanation power, its parsimony, and comprehensibility.

4.3 From Literate Programming to Literate Modelling

Literate programming co-develops a collection of programs with their informative models [97] that provide an explanation of the programs in a more user-friendly way. It can be extended by approaches to MaP with proper compilation of source code on the basis of small programs, program snippets, and a model suite. Programs and the model suite become interwoven. The first approach to literate programming has been oriented on programs with derived libraries of user interfaces and informative models [60]. It became far more advanced. Nowadays programs are interspersed with snippets of models. Most of these models are, however, documentation models. In this case, models formulate the meaning and understanding of programs. They connect this documentation to the program code at a high level and mostly in natural language. Documentation models integrate the code documentation with the developer idea documentation, the usage documentation, and the interface documentation. Literate programming thus overcomes the habit that the documentation is going to live in the program itself. The program is going to live in the documentation model. In a similar way, interface and informative models can be handled.

Literate programming can be based on models for the central program and models for interfacing and documenting. Since the central program is the governing program, literate modelling will essentially be a global-as-design approach for a model suite. Different users might use different models for the same application case.

Literate modelling is essentially modelling with integrated vertical and horizontal model suites. Horizontal model suites consist of models at the same abstraction level. A typical horizontal model suite in the global-as-design approach is the conceptual database structure model together with the view(point) external models for business users. Vertical model suites integrate models at various abstraction and stratification levels. Software development is often based on some kind of waterfall methodology. Models at a high level of abstraction are, for instance, storyboard and life case models. They are refined to models representing data, events, processes, infrastructure, and support systems. Modelling the OSI communication layer structure results in a typical vertical model suite.

Literate modelling incorporates a variety of models such as representation and informative models in a natural language into program development. It provides a high level abstraction and is thus program-language independent. The meaning of programs is provided prior to coding. Many-model thinking [79] can be developed towards model suite thinking. There are high-level introductory models such as informative models. These models are refined to models that reflect certain aspects of an application and to models that serve as origins for implementation models.

4.4 MaP: Towards Second Generation – Modelling as Programming

A central challenge of (conceptual) modelling is to facilitate the long-time dream of being able to develop (information) systems strictly by (conceptual) modeling. The approach should not only support abstract modelling of complex systems but also allow to formalize abstract specifications in ways that let developers complete programming tasks within the (conceptual) model itself. It thus generalises the model-driven and modelling_to_program approaches and develops a general approach to modelling as high-level programming.

Modelling is an activity guided by a number of postulates, paradigms, principles, and modelling styles. Already nowadays, we use paradigms such as global-as-design and principles such as meta-modelling based on generic and reference models. MaP is however dependent on deep models and the matrix. Next generation programming will also allow to be flexible in the postulates and paradigms.

Modelling can be organised in a similar way as structured programming, i.e. following a well-developed methodology and framework within an infrastructure of supporting tools. Models may be based on refinement approaches such as pattern-oriented development [5]. They contain enactor hocks for integration of source code.

Similar to system programming, modelling will be based on literate programming and literate modelling. For MaP, literate modelling becomes essential. Modelling_as_programming is oriented on development of complex systems. model suites Model suites can be developed on the basis of different frameworks as mentioned above,

A number of tools are going to support MaP:

- Sophisticated editors and frameworks have to be developed for this approach as extension and generalisation of existing ones, e.g. ADOxx, Kieler, MontiCore, and Ptolemy II.
- Code generation for the general MaP programme is still a matter of future development. There are already parts and pieces that can be used for generation and compilation: the RADD workbench realisation (Rapid Application and Database Design) [94], database programming by VisualSQL tool [49], performance management and tuning tuning (e.g. [90, 102]), advance high-level workflow specification [14], integrated web information systems design, and co-design.
- The implementation approach to MaP may be inspired by three solutions that are already common for programming languages:
  - Transformation and compilation is based on standardised combinable components. These components can also be reflected by specific models within a model suite.
  - Each specialisation can be enhanced by directives for compilation and by pragmas for pre-elaboration, convention setting, and exception handling like those in C++ and ADA. Model directives configure and pre-prepare a model for compilation. Models can be enhanced by default directives or by adornments detailing the interpretation of model elements. Pragmas are used to convey "pragmatic information" for compiler controllers, adapters, context enhancers. There are language-defined pragmas that give instructions for optimization, listing control, storage, import of routines from other environments, extenders for integration into systems, etc. An implementation may support additional (implementation-defined) pragmas.
  - MaP aims at programming-language independence. In this case, it has to be supported by multi-language compilers or compiler-compiler technology. For instance, database model suites are going to be mapped in a coherent and integrated form to object-relational, network, hypertext, etc. platforms. The association among various structuring of data structure is governed by the association schema of the model suite.

MaP requires a proper education for modellers. They have to master modelling, system thinking, programming techniques, reflection of models in various ways, communication with the application experts, and design of model suites. MaP knowledge and skills will become a central module in Computer Science similar to algorithmic thinking and programming. Model suites will become the mediating device between applications and their needs from one side and the system realisation from the other side. Already programming can be understood as an experimental science and as empirical theories by means of a computing device. Modelling continues this line. MaP thus needs a proper development of a theory and technology of modelling. Continuous model quality management will become a challenging issue.

5 Towards True Fifth Generation Programming

True fifth generation programming will be based on models which are considered to be programs due to generation of program codes from the models without programming at the level of 4PL or 3PL. Compilers or compiler-compilers are transforming the model suite directly to 3PL or 4PL code that can be compiled in the classical approach. A high-quality model suite can be used as a program of true fifth generation and will be mapped to programs in host languages of fourth or third generation.

A new generation of programming languages has to support a large variety of application areas since computers became an essential element of modern infrastructures and proper program support is necessary for all these areas, disciplines and daily life. We might try to develop a very large number of domain-specific languages. In this case, domain experts in a singleton domain are supported within their thought pattern. However, application are rather cross-domain applications with a wide variety of cultures, habits, and approaches. The literate modelling approach seems to be an alternative. In this case, model suites will thus become high-level programs and thus be the basis for true fifth generation programming in true fifth generation programming (5PL).

One potentially applicable realisation strategy is based on a layered approach discussed below similar to successful approaches such as LaTeX. We shall use this strategy in onion meta-model specification approach.

MaaP should also by partially independent on programmer's postulates, paradigms, and principles. It should also be tolerant and robust against changes in the 3PL and 4PL thus providing a programming language independence. At the same time, MaaP needs to be robust against a deviations from the normal application situation. We currently observe that application development is based on some kind of 'normal operating' in the application without taking into consideration potential but tolerable deviations from the normal case. At the same time, it must be supported by a specific MaaP education.

5.1 Ideas for 5PL on the Basis of MaaP

Models are often developed as normal models that inherit implicit deep models together with corresponding methodologies, techniques, and methods, i.e. its matrices. Normal models directly reflect origins according to the focus used for shaping the scope and non-scope of the model, functions that the model play in application scenarios, and analogy to be used for representing the origins. Justification and the quality sufficiency characteristics are often injected by deep models. Moreover, complex applications remain to be complex also in the case that a model suite is going to be used.

Modern applications are also interdisciplinary and are develop by many professionals which follow very different postulates, paradigms, and principles. They are not entirely harmonisable. They co-exist however in daily life. This co-existence can be expressed on the level of models but not on the level of programming languages.

Model suites will bond the applications, the domain and the computer support. They will become mediators between the application and the supporting infrastructure. Models are at the same time a means, an intermediary, and the medium for expressing separatable aspects of an application. Model suite will play the role of a "middle-range theory".

A model suite comes with its architecture and meta-model of the model suite. We may also use steering and governing models within a model suite. Some models in a model suite can be considered to be guiding ones since they are refined to more specific ones. A model suite can incorporate also quality-supporting meta-models (called checksum models) in order to provide means for quality control and for coherence support. Model suites incorporate also user-oriented interfacing models. We expect that these models will be developed as external and informative models for issues important for different users. In this case, models can be narrative as long as the association schema supports that. Synergetics approaches allow to develop master and slave models. Master models can be configured and adapted by control models. In this case, we need to build sophisticated editors for model suites.

The editors should be based on the same principles as the compilers for program generation from a given model suite. In the next subsection, we consider the onion meta-model for model suite composition. Editors should also include supporting means for check, control, and debugging.

Separation of concern can be based on application profiles [57]

normal model suites for the given application case

libraries of injectable deep model suites for landscaping and intrinsic strategic consideration steps

generic models and heritage model suites

from 3/4PL to 5PL by model suites representing thinking and reasoning

The MaaP approach will thus result in a complete model suite that becomes the source for the code of the problem solution, and for the system to be built. Figure 4

The delivered model (suite) is then going to be *compiled* to a program. The compiler has to transform the model to platform-dependent and directly executable programs after an optimisation phase. The compiler will typically be a 4-pass compiler (lexical analysis, semantical analysis, transformation and optimisation of intermediate code, target coding).

laymen programming

Von Neumann and Turing style of programming is only the beginning of a new era of development of computerised support mechanisms. This programming style does neither entirely reflect how humans reason and work nor represent the way how cultures, organisations, and societies act and evolve. We often map some kind of understanding to a program system and expect that the world outside computation is going to behave this way. Very soon after installation of such system it is going to be changed. Paradigms like programming-in-the-small and programming-in-the large can not be easily extended to programming-in-the-society or programming-in-the-world. The software and the data crises are essentially a result of the endeavour to ortho-normalise and to conquer the world by means of computer systems.

The simplest approach to change this situation is Models_as_a_Program that reflects human way of working and thinking as well as the understanding of a society.

5.2 One Potential Solution: The Onion Meta-Model Specification Approach

MaaP can be based on a stereotyped model suite specification. This specification may follow the style of LaTeX document specification onion in Figure 5. One typical solution (however only one of many) for system and also model suite development is vertical layering: (1) specify the surface and foundation; (2) provide an understanding of mechanisms of functions, processes, and operations; (3) develop means that the system functions; (4) develop the basis for functioning; (5) develop variants for a solution within a variation spreadsheet with adaptation facilities. This approach supports development and application of well-structured and composable models suites which are governed by the kind of model.

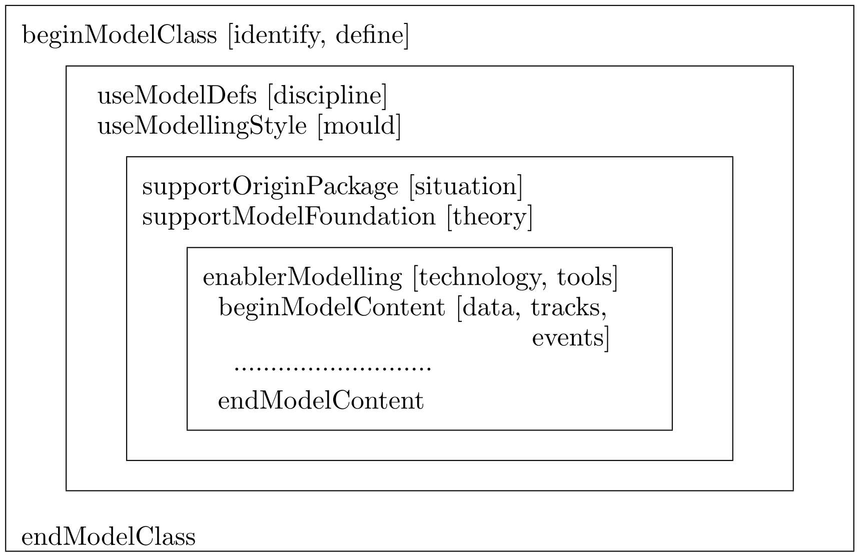

beginModelClass [identify, define]
useModelDefs [discipline]
useModellingStyle [mould]
supportOriginPackage [situation]
supportModelFoundation [theory]
enablerModelling [technology, tools]
beginModelContent [data, tracks, events]
....
endModelContent
endModelClass

Fig. 5. The onion approach to model suite specification (modified from [99])

Model class: Models are used as instruments and thus depend on the scenario in which the function. Different kinds of models can be stereotyped. This stereotypical categorisation can be used for the definition of the model class. The model class becomes the outer shell of the model specification onion. Model classes are based on the internal shell for model formating and general initialisation.

Model style and pattern: Depending on the model class, we may incorporate various libraries and methodological moulds at the model-style-and-discipline shell of the onion.

Model generics and foundations: Models consist of an internal deep model and of a normal model. The deep model is typically not completely revised. Its essential element is the model background. The grounding is accepted without any revision. The basis can be slightly extended and modified. The background forms the model foundation. The model foundation will be supported by a model package for representation of the specifics of the model situation. Additionally, generic models may be used as a basis for the normal model. These models may be collected in a library of generic models.

Model embedding and tools: The fourth shell is the support shell. Compilers are tools as well as a specification workbench. Technology support is a matter of convenience. A specific support is given for the combination of the given deep model with the normal model.

Normal model specification: The main part that is represents the specifics of the given set of origins is the normal model.

The specification setting follows the LaTeX compilation approach. The generation of the target model suite and the programs will result in a number of auxiliary elements similar to symbol tables for compilers of the first generation. The intermediate result of the transformation is a realisation-independent and infrastructure-independent model suite. The final result is then a realisation-dependent and infrastructure-dependent model suite or a program that may be executed in the given environment.

Pragmas and *model directives* are essential elements that we use for enhancement of conceptual models for system realisation. Pragmas can be considered as a language feature that allows adjusting or fine-tuning the behavior of a program compiled from a model. Model directives might also be used as additional control units for compilation. An essential element of a compiler is the *precompilation* based on a *prefetching* strategy of compiler-compilers.

Model correction is an essential element of prefetching. Already in the case of consistency maintenance for integrity constraints, we realised that models for translation must be correct. Schema-wide correctness is often neglected since most integrity constraints are declared at the local level. Cardinality constraints are a typical example which global correctness is partially based on local and type-based correctness. Since databases have to be finite and schemata are connected then we may derive implications for a given set of integrity constraints. We have to validate whether these implications are correct. Moreover, a set of cardinality constraints may be fulfilled only in infinite databases or in empty databases. Models must have a sufficient quality that is evaluated on the basis of corresponding evaluation procedures. If a model or a model suite is not correct then we have to improve the quality of a model before translation.

This approach is already currently realisable. Let us consider in brief two case studies [50, 65, 74, 99]:

MaaP for Database System Design and Development

Database structure modelling often uses extended entity-relationship models such as HERM [94]. HERM can also be extended to HERM+ with a specific algebra for database functionality and view collections. It is already well-known how to translate an existing entity-relationship schema to a so-called logical (or implementation) schema. ER schemata can be enhanced by directives for this translation.

Rule-Based OR compilation of HERM schemata and models uses the theory of extended entity-relationship models. In the case of extended entity-relationship schemata and of VisualSQL as a query and functionality specification language, we may use a rule system consisting of 12+1 translation phases for transformation. The phases for a compilation are the following ones:

0. Configuration of the HERM compiler, preprocessing, prefetching according to the model directives;
1. Schema and operation lexical analysis;
2. Syntactic analysis for schema and operations;
3. Semantical analysis schema and operations;
4. Generation of intermediate code that is also used as the ground schema for VisualSQL query, view, maintenance specification;
5. Preparation for schema and operation optimisation (or normalisation);
6. Schema tuning (operational optimisation);
7. Introduction and injection of controlled redundancy;
8. Redefinition and revision of generated types and operations (also UDT);
9. Recompilation for quantity matrix (Mengengerüst) (big and huge DB);
10. Toughening for evolution and change in data dictionary;
11. Derivation of support services and view towers;
12. Generation of data dictionary entries.

We can enhance this translation to more specific compilation and embedding into the corresponding platforms. Practitioners use in this case pragmas at least at the intentional level for object-relational technology. The translation includes also translation of view collections. Furthermore, HERM+ can be extended by VisualSQL that allows to declare queries at the level of HERM schemata. This translation results then directly in a performance-oriented structure specification at the level of physical schemato. We may envision that this approach can also be extended to other platforms such as XML or big data platforms. The result

Figure 6 specialises the general Figure 5 for database modelling as sophisticated database programming for a sample application (booking and financing issues in business applications based on the global-as-design specification approach).

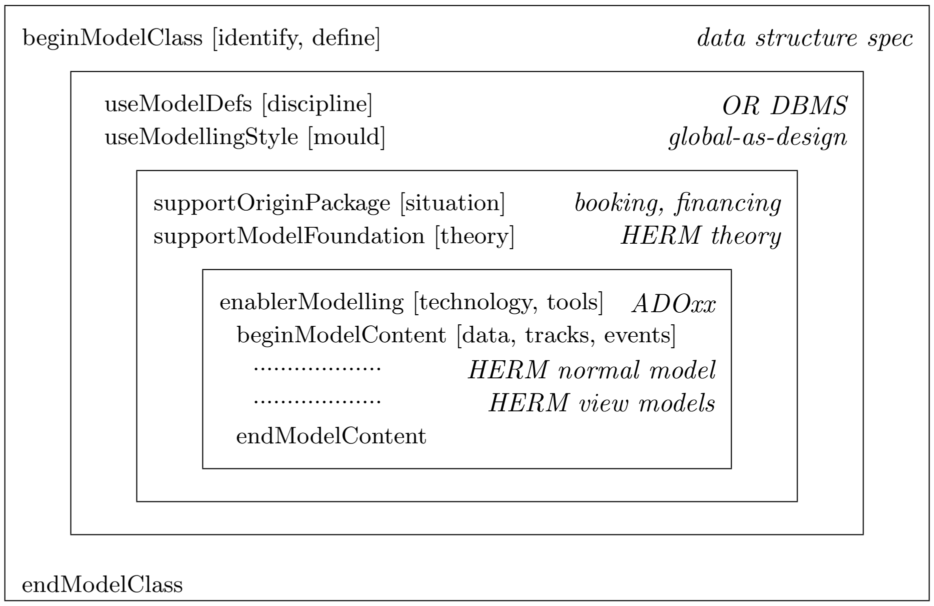

beginModelClass [identify, define]
data structure spec
useModelDefs [discipline]
OR DBMS
useModellingStyle [mould]
global-as-design
supportOriginPackage [situation]
booking, financing
supportModelFoundation [theory]
HERM theory
enablerModelling [technology, tools]
ADOxx
beginModelContent [data, tracks, events]
....
HERM normal model
....
HERM view models
endModelContent
endModelClass

Fig. 6. The model-centric database structure development based on HERM+

MaaP for Workflow Derivation From Storyboards

The second case study is going to discuss solutions if the deep models are not similar. In this case, we either normalise the source models in such a way that they can be transformed to the target models or programs or generate the full variety of potential target models or programs. The last approach is feasible if the generated models or programs are not changed on their own. All changes to them must be changes at the level of the source model suite. Web information system development has already successfully used this approach. Normalisation of source models is driven by well-formedness rules that are applied to the models.

Storyboarding and BPMN-based workflow specification are based on different deep models, i.e. we observe a deep model mismatch. The differences are similar to those observed for impedance mismatches between parallel database processing and sequential program computation. Therefore, the model transformation needs also approaches to overcome this mismatch. BPMN is strictly actor-oriented and based on a strict local-as-design paradigm. Storyboarding is more flexible. We might use global-as-design or partial local-as-design combined with global-as-design techniques. Storyboarding provides some freedom on the flow of activities. BPMN mainly uses a more strict form where diagrams are given with a static flow of activities. Diagrams are not adapted to the changes in user behaviour. Dynamic workflow specification is still based on flexibility at design time and on stability during runtime.

SiteLang specification allows to define scenes with plots for activities within a scene by different actors. This interaction among actors must be mapped to communication interaction between diagrams based on collaboration diagrams, to choreography diagrams among, or to conversations among actors. Scenes in a SiteLang specification can be visited by all enabled actors and completed by some of them. This freedom of task completion is neither achievable for normal BPMN diagrams nor for dynamic ones. Generic BPMN diagrams can however be used for adaptation to the actual actor behaviour at runtime.

The transformation of a storyboard can be based on language transformation rules. Typical rules are the following ones:

- An actor storyline is directly transferred to a BPMN pool.
- An atomic scene without atomic plots is transferred to a BPMN activity.
- A story sequence is represented by a sequence of BPMN activities. A well-formed story split with its own join (Fitch structure[1]) is transformed to the corresponding BPMN gateway structure. Optional scenes can be transformed to corresponding BPMN gateway structures. Iterations of simple stories can also be directly transferred to BPMN.
- Complex scenes are transformed to either BPMN diagrams or to complex activities or to sub-scenes.
- Story entries and completions are transferred to events in BPMN.
- Communication is based on BPMN communication pattern among diagrams, e.g. a link-scene-link combination among different actors.

The storyboard should also be normalised or transformed in some kind of well-formed storyboard. Parallel links between scenes are normalised either by introduction of intermediate scenes or by merging into a complex scene including plot transformation or link merging. The decision which next scenes are going to be chosen is integrated into the plot of the source scene. The naming of scenes is unified according to a naming scheme. Stories in a storyboard can be encapsulated into units with a singleton task. The storyboard is separated into mini-stories that will become workflows. The stories in a storyboard are decomposed into relatively independent mini-stories with starting and completing points. A mini-story must be representable by a singleton BPMN diagram if the actor in the mini-story does not change. Otherwise, we use superflow diagrams which call subflow diagrams as subprograms.

[74] introduced a small example for a storyboard of a trail system. The normalisation process leads directly to a diagram in Figure 7 that restricts the freedom and reduces the enabled actors to actors with encapsulatable behaviour within a mini-story.

There are several techniques and rules for storyboard conversion including data structure development:

Refinement preprocessing orients on data view mapping for each actor, on strict actor and role separation, on strict start-end flow, on session separation, and on additional communication.

[1] Each split must has its join in the diagram and each join has only its split what is a one-to-one association of splits and joins.

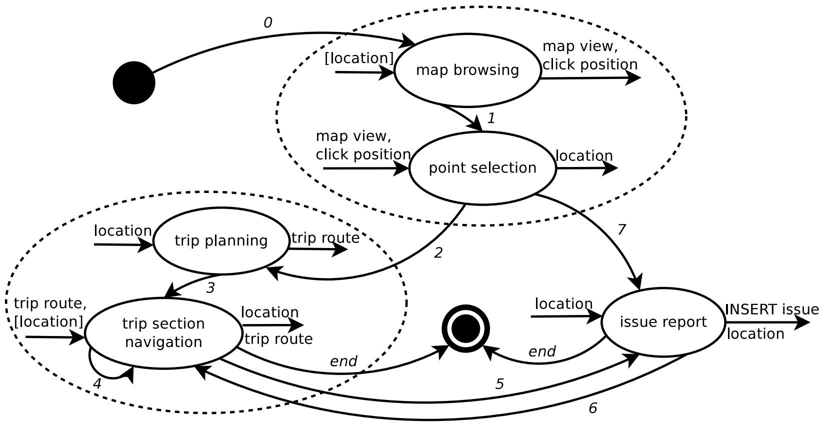


Fig. 7. The storyboard from [74] after refinement and normalization

Restriction of freedom is based on selection of the most essential flows since he storyboard that allows too much freedom. It downsizes the storyboard by restricting the flow of activities to the essential ones and by removing the rest.
Restricted and well-formed parallelism must yield to Fitch structuring.
Normalisation of the storyboard arrives at well-formed BPMN diagrams.
Plot integration for scenes with potential actor and role separation of these scenes into separate scenes with singleton actors.

We apply graph grammar rules to the stories since storyboarding uses a graphical language. The storyboard contains also the data viewpoint. This design information must be supported within a co-design approach to data structuring and workflow specification.

A normalised storyboard is now the basis for a BPMN diagram that displays only the visitors' viewpoint. The flow of activities is restricted to the most essential ones. Nothing else is supported. This transformation is information-loosing. A partial diagram after this transformation is displayed in Figure 8.

5.3 Towards Industrial Development On the Basis of Models

componentisation + standardisation + model suite architecture

6 M2P, MaP, MaaP: Summarising

The M2P, MaP, and MaaP approaches revise, combine, and generalise efforts and projects for model-driven development. We discussed some of those projects while knowing that is became already a major trend in software engineering.

We envision three already existing approaches:

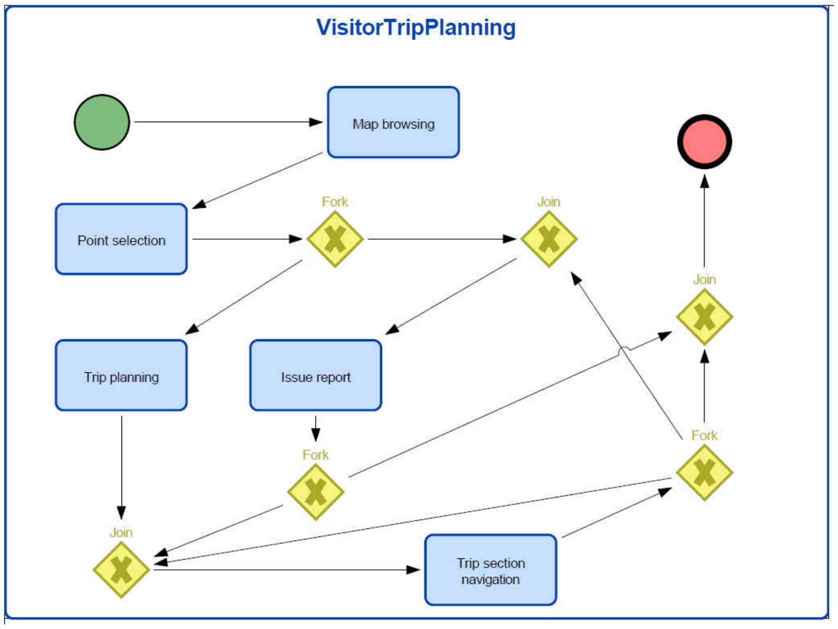


Fig. 8. Translation of the storyboard from [74] to some BPMN process

Finally we ask ourselves why we should target on true fifth generation programming. A number of reasons force us to use next generation programming:

- Computer engineering has not yet reached the level of being a science. It has not got its culture. Education is mainly education for handicraft work.
- Programming is more and more often performed by laymen, outsiders, casual users, non-specialists, self-made programmers, etc. They need a proper support.
- Apps and modern facilities are often developed without CS education. The uncontrolled growth worsens this situation.
- Users have not been the main concern for programmers. Operator and administrator thinking is still THE common sense understanding of the area.
- Migration, integration, maintenance, and modernisation are still a real nightmare. Half-life time of systems and software is far lower than lifespan of software deployment. New systems use new paradigms without downgrading features.
- Documentation is often luxury. Documentations are often generated from code (without inner documentation, without thoughtful architecture, with “monster” classes, only once and never after modification) because of it is otherwise not economic, it seems to be side work, it disturbs delivery in time, and there is no urgent need in better ones. Modernisation and maintenance becomes then a real vexation.

We finally complete this paper with a citation to a Genicore poster presented at ER'2017: "Modelware is the new software", i.e. domain-situation models instead of models of software. [51]

References

1. K. Adam, L. Netz, S. Varga, J. Michael, B. Rumpe, P. Heuser, and P. Letmathe. Model-based generation of enterprise information systems. *EMISA Forum*, 38(1):56–60, 2018.
2. H. Aiso. The fifth generation computer systems project. *Future Generation Comp. Syst.*, 4(3):159–175, 1988.
3. H. Aiso. 25 years of MITI and its influence on computing research in Japan. *IEEE Computer*, 24(9):99–100, 1991.
4. H. Aiso. Discussion on achievements, challenges, solutions, and problems of the 5th generation PL project, Nov. 13 2016. Evening with B. Thalheim and colleagues from Keio University, Yokohama, Japan.
5. B. AlBdaiwi, R. Noack, and B. Thalheim. Database structure modelling by stereotypes, pattern and templates. In *Proc. EJC 2014*, volume 2014/4 of *KCSS*, pages 1–19, Kiel, Germany, June 2014. Department of Computer Science, Faculty of Engineering.
6. B. AlBdaiwi, R. Noack, and B. Thalheim. Pattern-based conceptual data modelling. In *Information Modelling and Knowledge Bases*, volume XXVI of *Frontiers in Artificial Intelligence and Applications, 272*, pages 1–20. IOS Press, 2014.
7. A. Albrecht, M. Altus, E. Buchholz, A. Düsterhöft, and B. Thalheim. The rapid application and database development (RADD) workbench - A comfortable database design tool. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *CAiSE*, volume 932 of *Lecture Notes in Computer Science*, pages 327–340. Springer, 1995.
8. L. Baresi, F. Garzotto, and P. Paolini. Extending UML for modeling web applications. In *HICSS*, 2001.
9. J. Becker and P Delfmann, editors. *Reference Modeling: Efficient Information Systems Design Through Reuse of Information Models*. Springer, 2007.
10. A. Berztiss and B. Thalheim. Exceptions in information systems. In *Digital Libaries: Advanced Methods and Technologies, RCDL 2007*, pages 284–295, 2007.
11. A. Bienemann, K.-D. Schewe, and B. Thalheim. Towards a theory of genericity based on government and binding. In *Proc. ER'06, LNCS 4215*, pages 311–324. Springer, 2006.
12. E. Börger. A practice-oriented course on principles of computation. programming and systems design and analysis. In *CoLogNet / Formal Methods europe Symposium TFM'04*, Gent, 2004.
13. E. Börger and A. Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018.
14. E. Börger and O. Sörensen. BPMN core modeling concepts: Inheritance-based execution semantics. In *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*, chapter 9, pages 287–334. Springer, Berlin, 2011.
15. E. Börger and R. Stärk. *Abstract state machines - A method for high-level system design and analysis*. Springer, Berlin, 2003.
16. E. Börger and B. Thalheim. A method for verifiable and validatable business process modeling. In *Software Engineering*, LNCS 5316, pages 59 – 115. Springer, 2008.

17. J. Bormann and J. Lötzsch. *Definition und Realisierung von Fachsprachen mit DEPOT*. PhD thesis, Technische Universität Dresden, Sektion Mathematik, 1974.
18. M. Brambilla, S. Comai, P. Fraternali, and M. Matera. Designing web applications with WebML and WebRatio. pages 221–261, 2008.
19. G. Brassard and P. Bratley. *Algorithmics - Theory and Practice*. Prentice Hall, London, 1988.
20. J.E. Brenner. The logical process of model-based reasoning. In L. Magnani, W. Carnielli, and C. Pizzi, editors, *Model-based reasoning in science and technology*, pages 333–358. Springer, Heidelberg, 2010.
21. J. Conallen. *Building Web Applications with UML*. Addison-Wesley, Boston, 2003.
22. M.T. Cox and A. Raja, editors. *Mwetareasoning - Thinking about Thinking*. MIT Press, Cambridge, 2011.
23. A. Dahanayake. *An environment to support flexible information modelling*. PhD thesis, Delft University of Technology, 1997.
24. A. Dahanayake and B. Thalheim. Co-evolution of (information) system models. In *EMMSAD 2010*, volume 50 of *LNBIP*, pages 314–326. Springer, 2010.
25. T. Daly. Axiom. the scientific computation system. http://axiom-developer.org/axiom-website/, 2018.
26. P Deransart, M Jourdan, and B Lorho. *Attribute Grammars-Definitions, Systems and Bibliography*. LNCS 323. Springer Verlag, 1988.
27. D. Draheim and G. Weber. *Form-Oriented Analysis*. Springer, Berlin, 2005.
28. Eclipse project web site. http://www.eclipse.org.
29. M. Edwards. A brief history of holons. *Unpublished essay. http://www. integral-world. net/edwards13. html. Published*, 2003.
30. H. Ehrig, C. Ermel, U. Golas, and F. Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015.
31. D. W. Embley, S. W. Liddle, and O. Pastor. Conceptual-model programming: A manifesto. In *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*, chapter 1, pages 1–15. Springer, Berlin, 2010.
32. D. W. Embley and W.Y. Mok. Mapping conceptual models to database schemas. In *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*, chapter 5, pages 123–164. Springer, Berlin, 2010.
33. A. P. Ershov. The transformational machine: Theme and variations. In *Proc. MFCS 1981*, volume 118 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 1981.
34. E. A. Feigenbaum and P. McCorduck. *The fifth generation - artificial intelligence and Japan's computer challenge to the world*. Addison-Wesley, 1983.
35. P. Fettke and P. Loos, editors. *Reference Modeling for Business Systems Analysis*. Hershey, 2007.
36. G. Fiedler, H. Jaakkola, T. Mäkinen, B. Thalheim, and T. Varkoi. Application domain engineering for web information systems supported by SPICE. In *Proc. SPICE'07*, Seoul, Korea, May 2007. IOS Press.
37. R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, 39(2):59–66, 2006.
38. U. Frank. Multilevel modeling - toward a new paradigm of conceptual modeling and information systems design. *Business & Information Systems Engineering*, 6(6):319–337, 2014.
39. F. Garzotto, P. Paolini, and D. Schwabe. Hdm - a model-based approach to hypertext application design. *ACM ToIS*, 11(1):1–26, 1993.

40. D. Q. Goldin, S. Srinivasa, and B. Thalheim. Is = dbs + interaction: Towards principles of information system design. In *Proc. ER 2000*, volume 1920 of *Lecture Notes in Computer Science*, pages 140–153. Springer, 2000.
41. 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/Informationsverarbeitung der TU Dresden, Dresden, 1985.
42. D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, Reading, Massachusetts, 1987.
43. D. Harel and R. Marelly. *Come, Let's play: Scenario-based programming using LSCs and the play-engine.* Springer, Berlin, 2003.
44. K. Hölldobler, N. Jansen, B. Rumpe, and A. Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In *Proc. Modellierung 2020*, volume P-302 of *LNI*, pages 189–190. Gesellschaft für Informatik e.V., 2020.
45. G.-J. Houben, K. van der Sluijs, P. Barna, J. Broekstra, S. Casteleyn, Z. Fiala, and F. Frasincar. HERA. pages 263–301, 2008.
46. J. Hutschenreiter. *Zur Pragmatik von Fachsprachen.* PhD thesis, Technische Universität Dresden, Sektion Mathematik, 1986.
47. T. Isakowitz, E.A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, 1995.
48. H. Jaakkola, T. Mäkinen, B. Thalheim, and T. Varkoi. Evolving the database co-design framework by SPICE. In *Informaton Modelling and Knowledge Bases Vol. XVII, Series Frontiers in Arificial Intelligence, volume 136*, pages 268–279. IOS Press, May 2006.
49. H. Jaakkola and B. Thalheim. Visual SQL - high-quality er-based query treatment. In *IWCMQ'2003*, LNCS 2814, pages 129–139. Springer, 2003.
50. H. Jaakkola and B. Thalheim. Model-based fifth generation programming. In *Information Modelling and Knowledge Bases Vol. XXXI*, Frontiers in Artificial Intelligence and Applications, 312, pages 381–400. IOS Press, 2020.
51. T. Jonsson and H. Enquist. Semantic consistency in enterprise models - through seamless modelling and execution support. In *Proc. ER Forum 2017 and ER 2017 Demo Track*, volume 1979 of *CEUR Workshop Proceedings*, pages 343–346. CEUR-WS.org, 2017.
52. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.
53. D. Karagiannis, H. C. Mayr, and J. Mylopoulos, editors. *Domain-Specific Conceptual Modeling, Concepts, Methods and Tools.* Springer, 2016.
54. S. Kelly, M. Rossi, and J.-P. Tolvanen. What is needed in a metacase environment? *Enterprise Modelling and Information Systems Architectures*, 1(1):25–35, 2005.
55. S. Kelly and K. Smolander. Evolution and issues in MetaCASE. *Information & Software Technology*, 38(4):261–266, 1996.
56. 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.
57. Y. Kiyoki and B. Thalheim. Analysis-driven data collection, integration and preparation for visualisation. In *Information Modelling and Knowledge Bases*, volume XXIV, pages 142–160. IOS Press, 2013.

58. M. Klettke and B. Thalheim. Evolution and migration of information systems. In *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*, chapter 12, pages 381–420. Springer, Berlin, 2011.
59. D. E. Knuth. *The METAFONTbook*. Addison-Wesley, 1986.
60. D. E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information at Stanford/ California, 1992.
61. N. Koch, A. Knapp, G. Zhang, and H. Baumeister. UML-based web engineering. pages 157–191, 2008.
62. D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software and System Modeling*, 16(1):229–255, 2017.
63. F. Kramer and B. Thalheim. Holistic conceptual and logical database structure modelling with adoxx. In D. Karagiannis, H.C. Mayr, and J. Mylopoulos, editors, *Domain-specific conceptual model*, pages 269–290, Cham, 2016. Springer.
64. F.F. Kramer. *Ein allgemeiner Ansatz zur Metadaten-Verwaltung*. PhD thesis, Christian-Albrechts University of Kiel, Technical Faculty, Kiel, 2018.
65. Y. Kropp and B. Thalheim. Model-based interface generation. In *Proc. 29'th EJC*, pages 70–87, Lappeenranta, Finland, 2019. LUT, Finland.
66. G. Lakoff. *Women, fire, and dangerous things - What categories reveal about the mind*. The University of Chicago Press, Chicago, 1987.
67. L. Lamport. *LaTeX: A document preparation system*. Addison-Wesley, 1994.
68. J. Lötzsch. *Metasprachlich gestützte Verarbeitung ebener Fachsprachen*. PhD thesis, Dresden University of Technology, Germany, 1982.
69. L. Magnani, W. Carnielli, and C. Pizzi, editors. *Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery*. Springer, 2010.
70. B. Mahr. Information science and the logic of models. *Software and Systems Modeling*, 8(3):365–383, 2009.
71. D. Marco and M. Jennings. *Universal meta data models*. Wiley Publ. Inc., 2004.
72. H. C. Mayr, J. Michael, S. Ranasinghe, V. A. Shekhovtsov, and C. Steinberger. Model centered architecture. In *Conceptual Modeling Perspectives.*, pages 85–104. Springer, 2017.
73. S. Meliá and J. Gómez. The WebSA approach: Applying model driven engineering to web applications. *Journal of Web Engineering*, 5(2):121–149, 2006.
74. A. Molnar and B. Thalheim. Usage models mapped to programs. In *Proc. M2P – New Trends in Database and Information Systems*, Bled, 2019. Springer, CCIS 1064.
75. T. Moto-oka, editor. *Fifth generation computer systems*. North-Holland, Amsterdam, 1982.
76. P.-A. Muller, P. Studer, F. Fondement, and J. Bézivin. Platform independent web application modeling and development with Netsilon. *Software and System Modeling*, 4(4):424–442, 2005.
77. N. J. Nersessian. *Creating Scientific Concepts*. MIT Press, 2008.
78. K. Noack. Technologische und methodische Grundlagen von SCOPELAND. White paper, www.scopeland.de, 2009.
79. S. E. Page. *The model thinker – Waht you need to know to make data work for you*. Basic Books, New York, 2018.
80. O. Pastor, S.M. Abrahão, and J. Fons. An object-oriented approach to automate web applications development. In *Electronic Commerce and Web Technologies, Second International Conference, EC-Web 2001 Munich, Germany, September 4-6, 2001, Proceedings*, volume 2115 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 2001.

81. Oscar Pastor and Juan Carlos Molina. *Model-driven architecture in practice - a software production environment based on conceptual modeling*. Springer, 2007.
82. T. Pittman and J. Peters. *The Art of Compiler Design: Theory and Practice*. Prentice Hall, Upper Saddle River, 1992.
83. A.S. Podkolsin. *Computer-based modelling of solution processes for mathematical tasks (in Russian)*. ZPI at Mech-Mat MGU, Moscov, 2001.
84. Website PtolemyII. Ptolemy project: heterogeneous modelling and design. http://ptolemy.berkeley.edu/ptolemyII/, 2018. Accessed July 29, 2018.
85. D. Di Ruscio, H. Muccini, and A. Pierantonio. A data-modelling approach to web application synthesis. *Int. J. Web Eng. Technol.*, 1(3):320–337, 2004.
86. K.-D. Schewe and B. Thalheim. Co-design of web information systems. Texts & Monographs in Symbolic Computation, pages 293–332, Wien, 2015. Springer.
87. K.-D. Schewe and B. Thalheim. *Design and development of web information systems*. Springer, Chur, 2019.
88. J. W. Schmidt and F. Matthes. The DBPL project: Advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.
89. D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Communications of the ACM*, 38(8):45–46, 1995.
90. D. E. Shasha and P. Bonnet. *Database Tuning - Principles, Experiments, and Troubleshooting Techniques*. Elsevier, 2002.
91. L. Silverston. *The data model resource book. Revised edition*, volume 2. Wiley, 2001.
92. M. Steeg. *RADD/raddstar - A rule-based database schema compiler, evaluator, and optimizer*. PhD thesis, BTU Cottbus, Computer Science Institute, Cottbus, October 2000.
93. S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.
94. B. Thalheim. *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin, 2000.
95. B. Thalheim. Codesign of structuring, functionality, distribution and interactivity. *Australian Computer Science Comm.*, 31(6):3–12, 2004. Proc. APCCM'2004.
96. B. Thalheim. The theory of conceptual models, the theory of conceptual modelling and foundations of conceptual modelling. In *The Handbook of Conceptual Modeling: Its Usage and Its Challenges*, chapter 17, pages 547–580. Springer, Berlin, 2011.
97. B. Thalheim and A. Dahanayake. Comprehending a service by informative models. *T. Large-Scale Data- and Knowledge-Centered Systems*, 30:87–108, 2016.
98. B. Thalheim and I. Nissen, editors. *Wissenschaft und Kunst der Modellierung: Modelle, Modellieren, Modellierung*. De Gruyter, Boston, 2015.
99. B. Thalheim, A. Sotnikov, and I. Fiodorov. Models: The main tool of true fifth generation programming. In *Proc. EEKM 2019 – Enterprise Engineering and Knowledge Management*, pages 161–170, Moscov, 2019. CEUR workshop poceedings, Vol. 2413.
100. B. Thalheim, M. Tropmann-Frick, and T. Ziebermayr. Application of generic workflows for disaster management. In *Information Modelling and Knowledge Bases*, volume XXV of *Frontiers in Artificial Intelligence and Applications, 260*, pages 64–81. IOS Press, 2014.

101. S. Torge, W. Esswein, S. Lehrmann, and B. Thalheim. Categories for description of reference models. In *Information Modelling and Knowledge Bases*, volume XXV of *Frontiers in Artificial Intelligence and Applications, 260*, pages 229–240. IOS Press, 2014.
102. M. Tropmann, B. Thalheim, and R. Korff. Performance forecasting. *21. GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken)*, pages 9–13, 2009.
103. M. Tropmann-Frick. *Genericity in Process-Aware Information Systems.* PhD thesis, Christian-Albrechts University of Kiel, Technical Faculty, Kiel, 2016.
104. O. De Troyer, S. Casteleyn, and P. Plessers. WSDM: Web semantics design method. pages 303–351, 2008.
105. K. Ueda. Logic/constraint programming and concurrency: The hard-won lessons of the fifth generation computer project. *Sci. Comput. Program.*, 164:3–17, 2018.
106. R. P. van de Riet. An overview and appraisal of the fifth generation computer system project. *Future Generation Comp. Syst.*, 9(2):83–103, 1993.
107. B. Vela, C.J. Acu na, and E. Marcos. A model driven approach to XML database development. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER2004)*, pages 273–285, Shanghai, China, November 2004.
108. B. F. Webster. *Pitfalls of object-oriented development: a guide for the wary and entusiastic.* M&T books, New York, 1995.
109. P. Wegner and D. Q. Goldin. Computation beyond Turing machines. *Commun. ACM*, 46(4):100–102, 2003.
110. R. Wilhelm, H. Seidl, and S. Hack. *Compiler Design - Syntactic and Semantic Analysis.* Springer, 2013.
111. Li Xu and D. W. Embley. Combining the best of global-as-view and local-as-view for data integration. In *Proc. ISTA'2004*, volume P-48 of *LNI*, pages 123–136. GI, 2004.