INVITED PAPER

Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments

Design of embedded software-intensive systems based on well-developed system models is feasible and desirable; this paper proposes an approach to overcome the obstacles to achieving this goal.

By Manfred Broy, *Member IEEE*, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu

ABSTRACT | More than 20 years of research has created a large body of ideas, concepts, and theories for model-based development of embedded software-intensive systems. These approaches have been implemented by several tools and successfully applied to various development projects. However, the everyday use of model-based approaches in the automotive and avionic industries is still limited. Most of the time, the engineers work with a predefined set of isolated tools, and therefore adapt their engineering methods and process to the available tools. Today, the industry achieves tool integration by demand-driven, pragmatic, and ad-hoc composed chains of a priori existent commercial tools. Nevertheless, these tool chains are not (and cannot be) seamless, since the integration that can be achieved is not deep enough. This hampers the reuse and refinement of models, which subsequently leads to problems like redundancy, inconsistency, and lack of automation. In the end, these deficiencies decrease both the productivity and quality that could be provided by model-based approaches. To overcome these problems, a deep, coherent, and comprehensive integration of models and tools is required. Such an integration can be achieved by the following three ingredients: 1) a comprehensive modeling theory that serves as a semantic domain for the models, 2) an integrated architectural model that holistically describes the product and process, and 3) a manner to build tools that conform to the modeling theory and allow the authoring of the product model. We show that from a scientific point of view, all ingredients are at our hands to do a substantial step into an integrated process and tool world. Further, we illustrate why such a solution has not been achieved so far, and discuss what is to be done to get a step closer to seamless model-based engineering.

KEYWORDS | Common model repository; comprehensive modeling theory; generic tooling platform; integrated engineering environments; iterative language engineering; seamless model-based development; workflow support

Manuscript received September 8, 2009. First published February 25, 2010; current version published March 31, 2010. This work was supported in part by the German Federal Ministry of Education and Research under Grant SPES2020, 01IS08045A.

The authors are with the Fakultät für Informatik, Technische Universität München, 85748 Garching, Germany (e-mail: broy@in.tum.de; feilkas@in.tum.de; herrmama@in.tum.de; merenda@in.tum.de; ratiu@in.tum.de).

Digital Object Identifier: 10.1109/JPROC.2009.2037771

I. INTRODUCTION

Model-based development is adopted more or less consequently in practical development of automotive and avionic systems today. The pervasive use of models allows the engineers to abstract from implementation details, raising the level of abstraction at which the systems are developed. As a consequence, model-based development promises to increase the productivity and quality of software development for embedded systems.

However, model-based development approaches often fall short due to the lack of integration at both the conceptual and the tooling level. Even if artifacts are modeled explicitly, they are based on separate and unrelated modeling theories (if foundations are given at all), which makes the transition

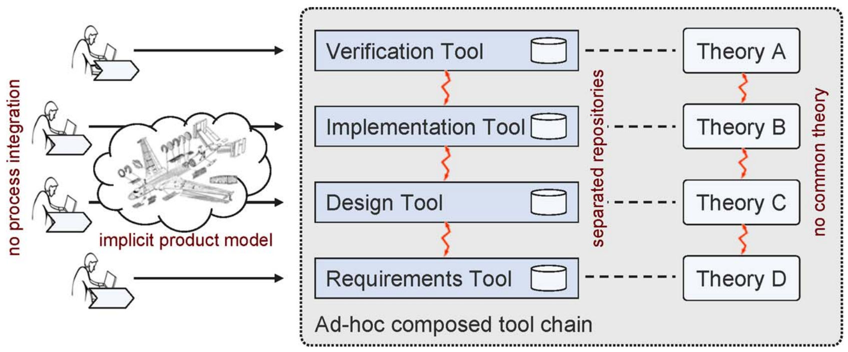

Fig. 1. ***Today's engineering environments: ad-hoc composed tool chains.***

from one artifact to another in the development process unclear and error-prone. Current tools usually focus on particular development steps and support single modeling paradigms (see Fig. 1). Although many of these tools do a good job in their limited domain, during the development of a system from initial requirements down to a running implementation in hard- and software, many models have to be constructed. In practice, several isolated tools are necessary to construct these models, and the transition between them is often far from clear. Consequently, the engineers adopt ad-hoc integration solutions that are far from a disciplined engineering. Both theories (whenever they are applied) and tools do not fit to each other, which hampers the reuse of models among different phases. Instead of refining and transforming the existent models, they are often rebuilt from scratch, which involves much effort and loss of information. The overall information about the developed product is available only implicitly in the engineers' minds.

The real benefits of the models take effect if they are used throughout the whole development process in a seamless way. For instance, requirements are the inputs for an initial system design and for test case generation. This workflow requires a deep integration of the requirements, the system design, and the tests in an integrated product model. Such integration can only be implemented in a model engineering environment supporting the reuse of the information that is captured within the models. To achieve the vision of seamless model-based development (illustrated in Fig. 2), we need the following three fundamental ingredients: 1) a comprehensive modeling theory that serves as a semantic domain for the formal definition of the models, 2) an integrated architectural model that describes the detailed structure of the product (product model) as well as the process to develop it (process model), and 3) an integrated model engineering environment that guarantees a seamless tool support for authoring and analyzing the product model according to the process defined in the process model. Instead of working with isolated models, engineers access via dedicated views a common model repository that explicitly stores the overall product model. All required views are formally defined and based on one comprehensive modeling theory, which enables the construction and unambiguous semantic interpretation of the product architecture. The compliance to the process model is assured by a common workflow engine.

Today one of the major impediments for the advent of seamless model-based development in the industry is the lack of tools. Ideally, based on modeling theories and a common product model, we should be able to *define* the requirements for tooling [see Fig. 3(a)]. In reality, however, a small set of tools (most of the time off-the-shelf) has to be used in a particular project. Thereby, these tools *impose* the modeling aspects that are treated and consequently the product model that is to be built [see Fig. 3(b)]. Due to the high costs of developing and maintaining tools, the development of in-house tools that are specific enough and tailored to a certain project is not an option. As a consequence, engineers need to adapt their process to the commercially available tools and many times to twist their wanted product design in a way enforced by the tools at hand. The top-down dependency between the modeling theory and their implementation in tools (as promoted by model-based development itself) is inverted in reality by the already available tools that dictate the modeling technique that is to be followed in a project.

In Section II, we describe the current situation of model-based development in practice—with focus on the

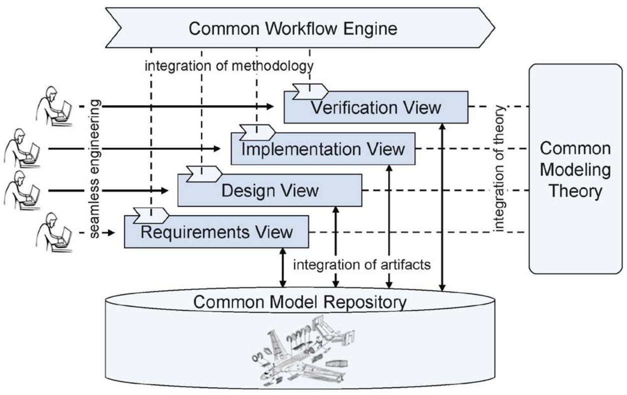

Fig. 2. ***Vision of integrated model-based engineering environment.***

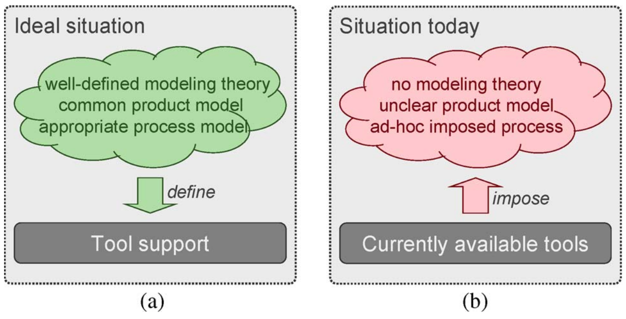

Fig. 3. *The tyranny of current tools. (a) Wanted and (b) unwanted.*

success that model-based development can currently claim and the pressing issues that prevent its pervasive usage in industry. To overcome these pressing issues, we outline in Section III our vision of a solution by an integrated engineering environment. Section IV explains from a technical point of view how such an integrated engineering environment could become reality and analyzes which technical ingredients are already available. Section V discusses the barriers toward developing such an integrated engineering environment due to the different interests of the involved stakeholders, and Section VI outlines a possible migration strategy. Section VII summarizes related work on integrated engineering environments before we conclude in Section VIII.

II. SITUATION IN PRACTICE TODAY: ISLANDS OF SUCCESS AND PRESSING ISSUES

In the following, we outline islands of success as well as pressing issues with respect to different aspects of model-based development. We ground this section both on a literature survey and on our experience gathered from working with industrial partners.

A. Islands of Success

Several advantages have been achieved in the development and application of model-based development techniques. Many academic as well as commercial tools and approaches are available that ease certain tasks in software engineering.

1) Formalized Modeling Languages: A well-defined semantic domain of the modeling techniques enables high automation, advanced tool support, and the use of verification techniques. For example, the formal method B has been successfully applied to develop the safety critical parts of several transportation systems. Verification was done using an interactive theorem prover. However, the authors state that "the practicality of their approach is entirely dependent on the quality and power of the tool support" [1], [2]. Another success story of modeling languages is Lustre [3]. It progressed from research to industrial use—in a commercial product—as the core language of the industrial development environment SCADE,[1] developed by Esterel Technologies. It is now used for critical control software in aircrafts, helicopters, and nuclear power plants (including the primary flight control system of the Airbus A380). Moreover, the language Esterel has been successfully applied to the development of avionic systems of medium size with a few hundred input and output signals. MATLAB Simulink and Stateflow[2] play an essential role in the automotive industry—e.g., some of the engine control units in current cars are completely modeled in MATLAB Simulink and Stateflow.

2) Verification: A considerable advantage of semantically well-founded modeling techniques is that once the models are built, powerful analysis and verification methods can be applied.

a) Testing: The most widely used method for verification in practice is testing. Model-based testing enables high automation and quality of the testing process, leading to savings of up to 50% for the testing tasks, which in the case of critical systems can take up to 50% of the development effort. The high quality is reached due to the generation of interesting test cases and the achievement of a very good coverage of the test space. Model-based testing proved good enough to discover a series of critical bugs such as the one from the Mars Lander [4].

b) Model Checking: Based on the advent of symbolic model checking in the early 1990s [5], the verification of industrial size systems became feasible. Model checking is on its way to a technique that is applied in the everyday systems development practice. Model checking was already used to verify a wide spectrum of safety critical avionic systems such as flight guidance [6], elimination of synchronization faults in air traffic control software [7], or verification of the control rules that support proper aviation traffic [8].

c) Theorem Proving: In comparison to model checking, theorem proving allows one to deal with specifications of more complex systems and proofs of more expressive properties. However, the verification of complex systems with theorem proving is most of the time semiautomatic. An example of success in using theorem proving is the Verisoft project.[3] In Verisoft, formal verification (using the Isabelle theorem prover[4]) has been applied to automotive systems in an industrial context by giving a pervasive correctness proof of the lower system layers: the hardware, the system software, the communication mechanism, and the programming model for the applications [9].

[1]http://www.esterel-technologies.com/index.html.
[2]http://www.mathworks.com/.
[3]http://www.verisoft.de/StartPage.html.
[4]http://isabelle.in.tum.de/.

3) Defining Comprehensive Domain Architectures: Whenever domain architectures are defined, they represent a backbone to which other artifacts are related. For example, domain architectures can be used to define a product model for the domain. Furthermore, by using a domain architecture as a common language for stakeholders, it facilitates the communication between client and supplier and the distributed development. For example, AUTOSAR[5] is one of the major approaches to create an integrated product model for the automotive domain. The AUTOSAR development partnership has defined a product model for the basic decomposition and interfaces in an automotive board net to ease the recombination and integration of software components. Due to its standardization and adoption by the car industry, we envision that AUTOSAR models will have a central role in the development of automotive software in the near future.

4) Automatic Synthesis of Models and Generation of Code: In most applications, the effort to develop a formalized model of an artifact is pretty high. However, once a model is built, it can be used as input for the refinement or generation of other models. The real benefit of formalized models occurs if they are used during the whole development process whenever necessary and in an automatic manner. In order to achieve automation, there are defined techniques for transformation between different kinds of models. For example, considerable work has been performed to generate state machines from scenarios of intended system behaviors [10]. Moreover, most of the widely used modeling tools allow the engineers to generate code from high-level models.

5) Achieving Certification: Safety-critical systems have to go through a rigorous certification process. Many certification conditions require high test coverage and integration. For example, the certification needs of the de-facto standard for certifying aerospace software (DO-178b) impose an explicit and well-defined assessment of the quality of software systems. The quality is certified according to the measure in which the implementation reflects the requirements and to the coverage degree between tests and requirements and between the tests and source code. A model-based approach concerning certification is the customization of general purpose modeling languages toward allowing the explicit implementation of the certification concerns. For example, [11] presents a Unified Modeling Language (UML) profile for enabling the precise modeling of the safety concepts from DO-178b. By using this profile, engineers can explicitly model their decisions related to safety; and based on this, reports containing certification-related information about the software can be generated.

[5]http://www.autosar.org.

B. Pressing Issues

Although many benefits have been achieved by adopting model-based development, there are several pressing issues that hamper an effective and seamless systems engineering.

1) Lack of Semantic Foundation: Many critical issues arise due to a missing, conflicting, or inappropriate semantic foundation of the modeling languages used today.

a) Weakly defined Semantic Domain: The semantics of modeling languages is often only specified as prose (if at all) and is thus insufficiently defined. This leads to inconsistencies between different implementations of the language in different tools and to ambiguities in the interpretation of the models. For example, a consequent usage of the UML [12] is difficult since a formal semantical interpretation of its models is missing. To countervail this weakness, much effort is currently spent to define precise semantics [13].

b) Multitude of Dialects of Modeling Languages: The existence of several dialects of the same modeling technique prevents a uniform treatment of the built models. For example, state machines exist in different dialects in StateMate, UML, and Rhapsody [14]. The models written in one dialect are ill-formed in another dialect or require a different semantical interpretation. These facts make the translation of models written in one formalism to another formalism difficult or even practically impossible. Furthermore, engineers that are experienced in one dialect might misinterpret models written in another dialect.

c) High Generality and Inappropriateness of Modeling Languages: One of the key challenges of modeling languages is the *abstraction challenge*—namely, how one can provide support for creating and manipulating problem-level abstractions as first-class modeling elements [15]. Engineers need several years to develop a specific product family (e.g,. the A350 family of Airbus aircrafts or the Z4 family of cars at BMW); and ideally, in order to be efficient, the modeling languages they use should directly support the development of that product and nothing more. Unfortunately, the employed modeling languages are highly general and, most of the time, domain independent and ontologically neutral [16]. This leads to a critical conceptual gap between the professional languages of engineers and the modeling languages. The same modeling techniques are used for describing a wide variety of situations, and a small number of predefined tools and modeling mechanisms are adapted to a wide variety of needs (e.g., these tools make no difference whether they build a model of a vehicle or an airplane). The domain inappropriateness and the generality of languages lead to abstraction loss: clearly defined concepts in the domain are not captured by the languages. This subsequently leads to weakly defined modeling languages that are too general and "not aware" of the specifics of a domain. For example, requirements structuring is an accepted best practice in requirements engineering. However, the structuring

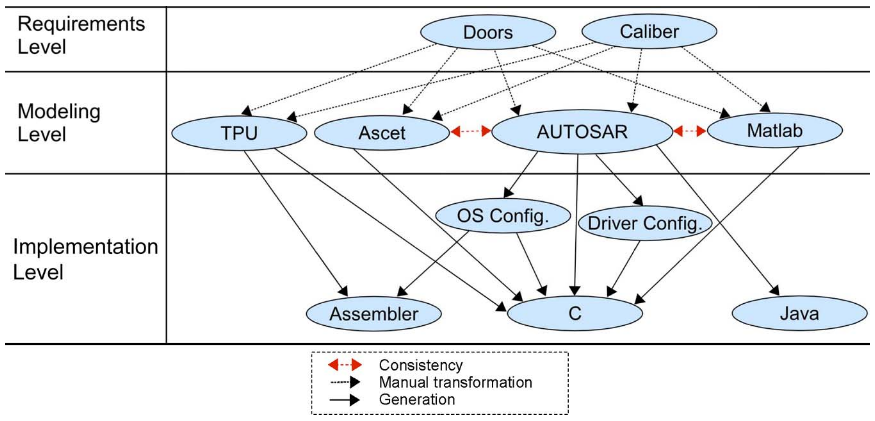

Fig. 4. ***An example of today's tooling situation in automotive software development.***

criteria vary among industries, companies, and even projects. In the avionics industry, the requirements of an airplane are ordered in a tree and classified according to a standard defined by the Air Transport Association of America (ATA). The ATA tree is only two levels deep, and different companies have the freedom to define additional levels. Unfortunately, the current commercial requirements engineering tools ignore the ATA chapters' way of structuring the requirements and provide only general structuring mechanisms such as generic requirements modules and objects (e.g., in Telelogic DOORS).

2) *No Integrated Architectural Model:* There is usually no integrated architectural model that defines how modeling starting from initial requirements down to a running system should be performed. Complementary modeling techniques provide better support for expressing different aspects of the system. In order to obtain a complete view of the system, the model views need to be integrated into a complete product model. Most of the time, however, the semantic integration of modeling techniques is not clear. For example, UML 2.0 [17] defines 13 types of diagrams for different development stages (e.g., use-case diagrams, activity diagram, component diagrams, and deployment diagrams). Even if these various views allow a better development of different aspects of the system, once different system views are constructed, it is not clear how they can be combined and integrated. The missing architectural model leads to the (logical) isolation of the developed models and subsequently to the inability to perform advanced model analyses such as feature interaction, impact analysis between different models, checking quality aspects that cross-cut models, etc.

a) Managing the Intent: Due to the isolation of models, the intent and rationale behind component designs are lost in the process [18]. The loss of intent and the impossibility to trace artifacts at different abstraction levels are effects of abstraction loss and of the lack of comprehensive architectural model. In order to document the intent more explicitly, trace links between different model elements in different tools are required. However, the trace links are only weak associations among model elements, and many times we need more advanced information about these links and their special meaning (e.g., that they should express consistency conditions). A typical example for missing trace links is requirements tracing information that is lost during the transition from a requirements engineering to a design modeling tool.

b) Managing Consistency: Consistency problems are twofold: vertical consistency and horizontal consistency. In the vertical case, we need to make sure that the models at two different phases of the development process are consistent with each other (e.g., specification with tests). The horizontal consistency is between the models created in the same phase–e.g., two views showing a perspective of the design. A well-known problem with the consistency of models (in this case belonging to two versions of CATIA) caused the cable crisis of A380 [19]. Different parts of the airplane were developed with incompatible versions of the program, and thereby the models (painfully) proved to be inconsistent at the end.

3) *Insufficiently Integrated Engineering Environments:* Many problems arise due to tooling issues such as missing tool integration and cumbersome handling of models. For example, in Fig. 4, we illustrate the flow of information between different tools employed in a typical development process in the automotive domain. The information is passed from one tool to another mainly by manual transformation. On the requirements level, tools like Telelogic Doors, Caliber, or even simple word-processing tools are used to describe the system functionality in semistructured natural language. After that, this information is used to build more structured models by design modeling tools. Beside AUTOSAR, also very special languages that are used to configure special hardware

controllers are employed. For example, ordinary von-Neumann processors are often not suitable to fulfill the very special requirements (e.g., timing) of engine control units, so special peripheral devices are used. These devices (e.g., Infineon's PCP/GPTA or Freescale's TPU) must be programmed in special (configuration) languages. These languages are then used to generate assembly code or C code to be deployed onto the Electronic Control Unit (ECU). Altogether, the information about the software-based functionality that runs within the electrical system of a vehicle is distributed in many different artifacts. For every artifact there is a special tool that offers special features to edit and analyze it. However, the information stored in these single artifacts corresponds to each other and is to be kept consistent manually.

a) Lack of suitable Tools: Only very few model-based technologies are backed up by tools that are robust and powerful enough for industrial use. For example, despite its shortcomings, the success of UML is (arguably) based also on the fact that it is well supported by commercial tools. Due to the high costs of tool development, only a few companies can afford to build their own solutions. Therefore, there is a general tendency to work with commercial-off-the-shelf (COTS) tools in industrial practice. On the one hand, COTS tools are too powerful and provide functionality that their users do not need in their daily work in a specific project. On the other hand, these tools are most of the time not aware of the specifics of the system being developed and thus do not support their users effectively. Beside "trivial" customization of layout, the advanced customization of the current tools is practically never done.

b) Isolation of Tools: Today many different kinds of tools are used to develop automotive and avionic systems. As soon as the system is modeled in different isolated tools, many problems arise since many tools are rather opaque and do not allow direct access to the models that they develop. Furthermore, if the access is possible, it is based on different heterogeneous technologies. Often the tools are standalone solutions: Tools are designed to be used as simple standalone programs that are executed by one developer on one machine. In practice, modeling is usually done in a collaborative way with a number of people involved. The fact that parts of the product model are implemented in different tools leads to redundancy. Then, in most cases, this redundancy is not modeled in exactly the same way, which makes it even more complicated to check the consistency. The integration of tools is done today in a peer-to-peer manner using tool couplers. This approach does not scale, since the number of couplers increases exponentially in the number of tools. Moreover, the coupler-based integration of tools is done in an ad-hoc manner in order to solve specific pragmatic problems.

c) Weak Support of collaborative Work: The relation between supplier and customer brings specific challenges such as specification of interfaces or integration of processes of customer and supplier. Furthermore, distributing and synchronizing global development requires a high degree of composability in the modeling techniques. In the automotive industry, the supplier usually has the so-called integration responsibility. This means that he is responsible to deploy several applications onto an ECU, which is the deliverable to the original equipment manufacturer (OEM). The OEM is responsible for the integration of the ECUs into the bordnet. The hardware-based composition of the total system out of several ECUs is subject to interface errors due to the lack of type checking. So in future automotive development, ECUs will no longer be the right concept to decompose the system and to break it into parts to be developed independently. A more software- rather than hardware-oriented decomposition and integration will be needed. So the OEMs will gradually gain more and more integration responsibility, as is already practiced in the avionic industry.

The collaborative development is very complicated since even application functions that are independent from each other from a functional point of view interact implicitly if they are deployed on the same hardware infrastructure in the car (e.g., they share the bus systems or run on the same ECUs). Because of this, not only the functional interfaces between the applications have to be precisely specified but also many other interactions that must be taken into account—e.g., interactions that are mediated indirectly through shared resources such as memory, peripheral devices, processor time, as well as bus load. Furthermore, the collaborating parties, e.g., an OEM and its component suppliers have to agree on two levels before seamless collaboration is possible: The interface of the supplied component must be defined (model-level), and the modeling technique used by the supplier should fit into the tool chain of the OEM (metamodel-level). A deep integration of tools is of importance because of the high degree of implicit and explicit interaction between the applications. Static analysis ranging from simple type checking (e.g., the consistent encoding of signals into CAN messages) to highly sophisticated scheduling analysis techniques [20] becomes an important method for verification and quality assurance.

d) Securing intellectual Property Rights: Distributed development especially requires a sophisticated rights management. In [21], the difficulty of the supplier–OEM relationship is mentioned as an important issue in the automotive domain. OEMs need to be able to investigate and check the quality of the delivered artifacts to make sure that the subsystems are compatible with their environment. OEMs also want to make sure that the delivered systems are well crafted. The sole activity of OEMs today is testing (dynamic analysis) and process assessments. The adoption of static analysis techniques is often not applicable. Suppliers on the other side are interested in keeping their special know-how and intellectual properties undisclosed. In many cases, it seems to be a good idea to agree on object files as a deliverable. This has the big advantage that the intellectual

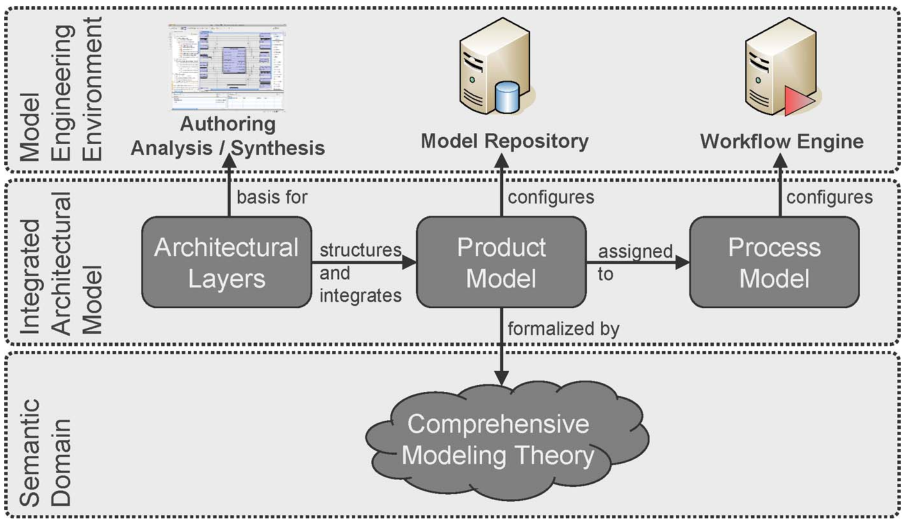

Fig. 5. ***Main ingredients for seamless model-based development.***

properties of the supplier remain undisclosed. However, the disadvantages are that static analysis that could ensure the correctness of the behavior of the software component in its environment (e.g., model checking) or the assessment of quality attributes is not possible.

e) Lacking Evolvability of Modeling Languages: Although often neglected, a modeling language is subject to change like any other software artifact [22]. This holds even for general-purpose modeling languages: e.g., UML, although relatively young, already has a rich evolution history. Domain-specific modeling languages are even more prone to change, as they have to be adapted whenever their domain changes due to technological progress or evolving requirements. Experiences from collaborations with our partners from the automotive industry show that languages are often not adapted to new requirements due to missing tool support for the resulting migration of models. However, modelers often find workarounds and encode additional information in a way not intended by the original language design. As the language editors cannot enforce the well-formedness of the introduced constructs, different modelers may choose different workarounds to encode the same additional information. Furthermore, it is difficult for language tools to process this information in a homogeneous way. Consequently, lack of evolvability can decrease the value of a modeling language in the long run.

III. VISION OF SEAMLESS SYSTEM DEVELOPMENT: CONTENT IN THE MIDDLE

Seamless model-based development promises to lift software development to higher levels of abstraction by providing integrated chains of models covering all phases from requirements to system design and verification. In seamless model-based development, modeling is not just an implementation method but also a paradigm that provides support throughout the entire development and maintenance lifecycle. Modeling starts early in the development process with requirements engineering, where informal requirements are turned into models step by step. At the end of the requirements engineering phase, we have a functional model capturing the requirements. In turn, the system architecture and, in sequence, the software architecture is described by various models that capture different aspects of the system. Provided these models are chosen carefully enough and based on a proper theory, the architecture model can be verified to guarantee that the functional requirements are fulfilled. Furthermore, a rigorous tracing is enabled between the functional requirements and the architecture model. To be able to do that, a carefully structured architecture model has to be worked out—not just describing a system at the technical implementation level but also describing carefully chosen useful abstractions such as function hierarchies and logical architectures [23].

Seamless and comprehensive model-based development is a key to a more systematic development process with higher potential for automation. To be able to work out such an approach, a number of ingredients are required, as illustrated in Fig. 5. These ingredients can be divided into three levels: the *semantic domain* forms the basis of an *integrated architecture model,* which is operationalized by a *model engineering environment*. In the following, we detail on these levels and their corresponding ingredients.

A. Semantic Domain

Seamless model-based development requires a *comprehensive modeling theory* as a theoretical basis to ensure a thorough formalization of all artifacts produced during the development of a system. Firstly an appropriate modeling theory provides the appropriate modeling concepts such as the *concept of a system* and that of a *user function*, with:

1) a concept for structuring the functionality by *function hierarchies;*
2) concepts to establish *dependency relationships* between these functions;
3) techniques to *model the functions* with their *behavior* in isolation including *time* behavior and to connect them—according to their dependability relations—into a comprehensive functional model for the system.

Secondly it provides a concept of *composition* and *architecture* to capture:

1) the *decomposition* of the system into *components* that cooperate and interact to achieve the system functionalities;
2) the interfaces of the components including not only the *syntactic interfaces* but also the *behavior interfaces;*
3) a notion of *modular composition* which allows us to define the interface behavior of a composed system from the interface behaviors of its components.

The modeling theory must be strong and expressive enough to model all relevant aspects of hardware and software architectures of a system such as structuring software deployment, description of tasks and threads, and modeling behavior aspects of hardware. These aspects and properties of architecture should be represented in a very direct and explicit way. Our approach to such a modeling theory is given by the *FOCUS* theory [24] and its various extensions.

B. Integrated Architectural Model

A comprehensive architecture model of an embedded system and its functionality is a basis for a product model that comprises all the content needed to specify a distributed embedded system in terms of its comprehensive architecture. A first version of an integrated architectural model for the automotive domain is described in [25].

1) Architectural Layers: An architectural model describes all model views that define a system at different abstraction levels and the relations among them. It enables a systematic and domain-appropriate development process and represents the starting point for tool support. All views on a system are part of the product model.

2) Product Model: In order to describe all the interesting aspects of a system, we need a domain-appropriate system architecture that contains all modeling artifacts in a product model. Its structure is described by a metamodel that is the basis for a data model that allows one to capture all the contents. This product model describes an embedded system inside a computer and can be subsequently used as a data backbone for development. In the product model, the dependencies and relationships between the modeling artifacts should be made explicit, since they are a key to extensive tool support. In the end, all artifacts produced throughout the development process should be part of the product model and related in a semantic way such that important issues such as tracing, impact analysis, and consistency checks are supported.

3) Process Model: A comprehensive process model is mandatory that relates the modeling artifacts to the activities that are needed to construct the architecture model step by step. According to the consistency and quality notions of the product model, the process model defines the sequence of steps that need to be performed at a certain development phase.

C. Integrated Model Engineering Environment

A central characteristic of model-based development is a high degree of automation by extensive tool support. The level of automation that can be achieved strongly depends on the used models and the associated theory. In fact, the support for automation has to address the capturing and elaboration of models, the analysis of models with respect to their consistency and important properties, and techniques for generating further development artifacts from models. The tooling should be based exclusively on the product model. Then all tools that carry out the steps of capturing models and creating models, analyzing models, and generating new artifacts from existing ones basically only manipulate and enhance the product model. The whole development should be regarded as an incremental and iterative process with the goal to work out the contents of a comprehensive product model.

In order to turn the vision of high automation into reality, we need an integrated engineering environment that offers support for creating and managing models within well-defined process steps. The integrated development environment should comprise the following four blocks: 1) a *model repository* that maintains the different artifacts including their dependencies, 2) advanced tools for *editing* models that directly support their users to build up models, 3) tools for *analyzing* the product model and *synthesizing* new artifacts out of the product model, and 4) a *workflow engine* to guide the engineers through the steps defined by the development process.

IV. REALIZATION OF AN INTEGRATED ENGINEERING ENVIRONMENT: TOOLING ISSUES

For all intents and purposes, the integrated modeling language should be operationalized by a tooling environment that supports the creation, transformation, analysis, and subsequent processing of all the artifacts that are needed. Due to the fact that tool development is extremely

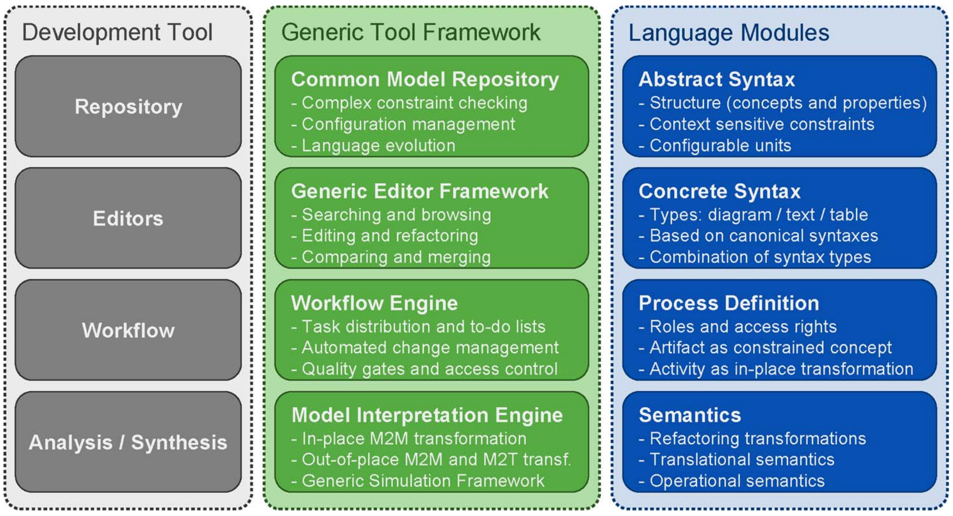
Development Tool
Repository
Editors
Workflow
Analysis / Synthesis
Generic Tool Framework
Common Model Repository
- Complex constraint checking
- Configuration management
- Language evolution
Generic Editor Framework
- Searching and browsing
- Editing and refactoring
- Comparing and merging
Workflow Engine
- Task distribution and to-do lists
- Automated change management
- Quality gates and access control
Model Interpretation Engine
- In-place M2M transformation
- Out-of-place M2M and M2T transf.
- Generic Simulation Framework
Language Modules
Abstract Syntax
- Structure (concepts and properties)
- Context sensitive constraints
- Configurable units
Concrete Syntax
- Types: diagram / text / table
- Based on canonical syntaxes
- Combination of syntax types
Process Definition
- Roles and access rights
- Artifact as constrained concept
- Activity as in-place transformation
Semantics
- Refactoring transformations
- Translational semantics
- Operational semantics

Fig. 6. ***Decomposing development tools.***

expensive, the industry sees no alternative to existing commercial tools. These tools many times are of general nature and not tailored to the specific needs of the engineers from a specific industry. Many efforts trying to develop their own tailored integrated engineering environment fail because of huge development efforts but even more substantial maintenance costs. This is due to the fact that besides the core business functionality, tools need a lot of infrastructure for the management of models. Fig. 6 shows that a development tool can be decomposed into the four parts *Editors*, *Workflow*, and *Analysis and Synthesis*. Each of the four parts can again be divided into a generic and language-specific part. The generic part can be provided by a *Generic Tool Framework*. The language-specific parts are named *Language Modules*.

To assess the ratio of the business functionality and the management infrastructure, we performed an empirical study on the functionality provided by engineering tools. Our assumption is that by investigating the menus or toolbars of a tool and by classifying the functionality that can be accessed from there, we can estimate the ratio between the functionality of tools that is related to their business domain and the functionality that is independent of the business domain. We performed our empirical study based on the description command buttons in menus and toolbars as given in the user documentation of the following tools: Esterel Scade v6.0, Telelogic Rhapsody v7.4, and Telelogic Doors v9.1. In Table 1, we present an overview of the functionality of these COTS tools as resulted from our classification. We notice in this figure that most of the functionality accessible to tool users is related to the editing of the models (e.g., creation, modification, or deletion of model parts), navigation (e.g., searching), and layout (e.g., colors, fonts, zooming). In fact, the functionality that is strongly related to a specific language (e.g., different analyses of models, synthesis of other information from models) is rather small. Thus, a big part of the front-end functionality of tools is unspecific, can be seen as commodity, and mostly could be provided by a generic tooling platform. We also can see that none of the tools provides an integrated workflow support.

However, it is in particular the heterogeneous implementation of this infrastructure in different tools that hinders their seamless integration. To countervail this situation, we propose a generic tooling platform that offers all the technical details that are independent of the concrete modeling language. In today's development tools, these so-called *horizontal tooling aspects* are interwoven with the implementation of the proper modeling languages, which we call *vertical tooling aspects*. However, an integrated modeling language is rather complex and thus difficult to develop in one step. In order to ease language development, an integrated modeling language should result from the composition of reusable, modular modeling languages that can be customized to the specific needs of the engineers. Furthermore, appropriate tool support is

Table 1 Quantitative Overview Over the Functionality of COTS

	Scade	Rhapsody	Doors
Repository			
- Persistency	7	3	2
- Configuration Management	-	-	6
Editors			
- Editing	107	95	40
- Navigation	24	5	14
- Layout	15	43	12
Analysis/Synthesis			
- Analysis	29	12	1
- Synthesis	14	7	-
Total	**196**	**165**	**75**

required for model migration in order to be able to improve a modeling language that is already under use.

A. Separation of Horizontal and Vertical Tooling Aspects

In order to reduce development costs, the tooling platform has to factor out the functionality that is independent of the specific product model. The tooling platform can then be parametrized by a modeling language, which operationalizes a certain product model. By means of our tooling platform, we thus want to achieve a strict separation of horizontal and vertical tooling aspects. Tooling aspects like the central model repository that are independent of a specific modeling language are termed *horizontal*. Tooling aspects like the syntax of a certain modeling language that are specific to a certain modeling language are called *vertical*. In today's development tools, these horizontal tooling aspects are interwoven with the implementation of the vertical tooling aspects. The missing separation of horizontal and vertical tooling aspects hampers the implementation of a central model repository, which is crucial for the introduction of an integrated engineering environment. Fig. 7 depicts the different tooling aspects together with their classification. In the following, we discuss the ingredients that are necessary to implement both vertical and horizontal tooling aspects.

1) Vertical Aspects: Tooling aspects are called vertical if they are specific to a certain modeling language. The tooling platform must support tool builders to easily implement vertical aspects. It should be easily possible for a company to adapt or develop a modeling language appropriate to its needs. In order to enable the cost-effective development of such modeling languages, we need so-called meta languages to describe the different elements of a modeling language. The tooling aspects related to supporting modeling languages are partitioned into the following elements:

- abstract syntax;
- concrete syntax;
- process definition;
- semantics.

In the following, we inspect the different elements of modeling languages and their requirements in more detail.

a) Abstract Syntax: The *abstract syntax* defines the concepts of a modeling language and their relationships. When a modeling language is appropriate to a domain, it enables the engineers to directly reflect the domain concepts and relations in their models. By using domain-appropriate languages, the engineers can work at a higher abstraction level and in direct analogy to the domain knowledge.

The abstract syntax determines the validity of models and can therefore be used to enforce the construction of valid models. The domain semantics of languages can be encoded in an abstract syntax by restricting syntactic correct models to those that are meaningful in the domain [26]. The abstract syntax usually consists of constructive and descriptive parts: constructive parts describe how to build valid models, and descriptive parts further restrict the number of valid models by constraints. Since an integrated modeling theory needs to describe the relationship between different models, a model is required to have a graph-like structure.

We advocate to put the abstract syntax in the center of modeling language definition. Other elements of a modeling language definition are then specified in relation to the abstract syntax. This enables the rapid development of modeling languages. Furthermore, different modeling languages are best integrated in terms of their abstract syntax.

The literature provides a large number of examples for languages to define the abstract syntax of a modeling language. The Object Management Group (OMG) even standardized languages to define the abstract syntax of object-oriented modeling languages: the Meta Object Facility (MOF) [27] for the constructive part and the Object Constraint Language (OCL) [28] for the descriptive part. However, MOF provides too many constructs to be completely understood and implemented. Consequently, one of the most widely used implementations of MOF, the Eclipse Modeling Framework (EMF) [29], implements only a rather small subset of MOF.

b) Concrete Syntax: The *concrete syntax* defines the representation of a model in a human-readable manner. There are different forms of concrete syntax: diagrammatic, textual, and tabular. The diagrammatic syntax shows the model in diagrams with layout information, the textual syntax visualizes the model as linear texts, and the tabular syntax visualizes the model in two-dimensional tables.

As real-world models can become quite large, the concrete representation of a whole model becomes incomprehensible. As a consequence, we have to be able to define a concrete syntax only for a view onto the model. For example, only the direct subcomponents of a

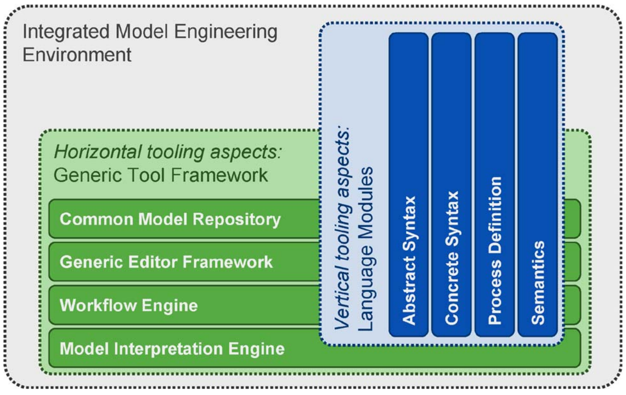
Integrated Model Engineering Environment
Horizontal tooling aspects: Generic Tool Framework
Common Model Repository
Generic Editor Framework
Workflow Engine
Model Interpretation Engine
Vertical tooling aspects: Language Modules
Abstract Syntax
Concrete Syntax
Process Definition
Semantics

Fig. 7. ***Horizontal and vertical tooling aspects of an integrated model engineering environment.***

component are visualized in a diagram. Furthermore, the representations of the different views have to be related with each other. The black-box of a component is depicted in the diagram for its parent component, whereas the white-box is shown in a different diagram.

Some modelers may prefer the diagrammatic concrete syntax, while others prefer the textual one. As a consequence, there might be several representations of the same view in different variations of concrete syntax. The consistency between the different representations has to be guaranteed by means of abstract syntax. Furthermore, it should be possible to combine several variations of concrete syntax for a view. A diagrammatic representation of a state machine, for example, may contain textual representations of the transition guards.

As we put the abstract syntax in the center of language definition, the concrete syntax has to be defined as a function that maps an abstract representation of a model into a concrete representation. If this function is bidirectional, it can be employed to provide authoring for the model. Otherwise, it provides only a read-only representation of the model. In order to enable rapid prototyping of modeling languages, there can be a mapping into a concrete canonical representation. One then starts from that mapping and subsequently refines it to get the desired concrete syntax.

There are already some approaches to define the concrete syntax on top of an abstract syntax. Textual Concrete Syntax (TCS) provides a template language to define a bidirectional function that maps EMF models into textual representations [30]. The Graphical Modeling Framework provides a language to specify a diagrammatic syntax for EMF models and allows one to generate an authoring tool from that specification.[6] Diagram Interchange Mapping Language provides a language to define a mapping from the abstract syntax to a diagrammatic syntax and a tool architecture to reconcile the diagrams based on model transformations [31]. Most of the approaches towards concrete syntax definition do not provide a clear separation between abstract and concrete syntax. This makes it difficult to define alternative concrete syntaxes for the same abstract syntax.

c) Process Definition: Part of the language definition is also the methodical way of modeling, defining at what time which parts of the model have to be developed. For each development phase it defines both which operations are available and what properties have to be fulfilled at the end of the phase. The process definition is interpreted by (and thus parametrizes) a workflow engine.

A process definition consists of the activities that have to be performed, the roles that are responsible for certain activities, and the artifacts that are produced in the course of certain activities. The abstract syntax defines the possible structure of the artifacts, and the concrete syntax the different views onto the model. The roles come with access rights that regulate the access to certain views onto the model. Activities may be performed sequentially, in parallel as well as iteratively. For a better overview, activities should be structured hierarchically. A basic activity may be fully automated such as code generation and can then be specified by an interpreter of the modeling language. On the other hand, a basic activity may have to be performed manually like, e.g., requirements elicitation, and can then be supported by the operations defined by the modeling language. Furthermore, the transition from one activity to the next may be protected by quality gates, which guarantee the quality of the activity's result. This can be achieved by integrity constraints or by the execution of complex analysis by interpreters. Integrity constraints actually depend not only on the modeling language but also on the progress of the process. For example, every requirement has to be implemented at the end of the process but is of course not implemented after requirements elicitation.

d) Semantics: Generally, there are three ways of specifying semantics. The first one is to describe the semantics of the modeling language by a calculus, the second one is to define the relationship to another formalism (denotational and translational semantics), and the third one is to specify a model interpreter (operational semantics).

The first one results in syntactical transformation rules preserving the semantics. It is possible to provide these rules with regard to tool support in the form of refactoring functionality, which is being realized via an in-place transformation engine (the original model is thus altered directly). In general, it must be differentiated between postulated rules (axioms) and deducible rules (theorems). In the scope of a language definition, however, axioms would in principle be sufficient. As theorems are, however, generally not deducible in an automated way but are particularly relevant in practice for the refactoring, they should nonetheless be formulated explicitly in the language definition. From a formal point of view, the syntactic transformation rules complete the syntax definition to form a calculus.

The second way maps each model according to the syntax definition to a model of another formalism (referred to as semantic domain). This may be a mathematical formalism like logic or set theory (denotational semantics) but also a programming language like C or Java (translational semantics). Note that this kind of semantical definition always depends on another formalism, which needs to be formalized itself. In total, this results in a system of modeling languages that are correlated to each other by the semantical mapping. According to our integrated tooling framework, the specified transformation rules are performed by an out-place transformation engine, i.e., the original model is not altered.

The third way describes how a valid model is interpreted as sequences of computational steps. These

[6]http://www.eclipse.org/modeling/gmf/.

sequences then make up the meaning of the model. In the context of generic tooling environments, it is therefore possible to use operational semantics to parameterize a generic simulation framework. Kermeta [32] is aiming at such a solution.

2) Horizontal Aspects: Tooling aspects are called horizontal if they are independent of a certain modeling language. Horizontal tooling aspects like a model repository are often reimplemented by each isolated tool. However, the use of different technologies for a model repository complicates seamless tool integration, as models have to be transformed to enable data exchange between the tools. Therefore, we propose a tooling platform that factors out horizontal tooling aspects. We identified the following horizontal tooling aspects that are required for seamless system development in the large:

- common model repository;
- generic editor framework;
- workflow engine;
- model interpretation engine;

In the following, we deal with the different horizontal tooling aspects and their requirements in more detail.

a) Common Model Repository: A central model repository is crucial for maintaining the dependencies between the different models produced during the development process. As the models of industrial systems become quite large, a database system is required to store all the models and their dependencies. The central model repository is also responsible to ensure the overall consistency of the models. A model is consistent if it fulfills the constraints defined by the modeling language.

In order to efficiently handle distributed development of systems, the database system has to be distributed. The models may be partitioned according to the different companies that participate in the development of a system, since each company needs to have sovereignty over its own models. Furthermore, as some companies may not be permitted to access or modify the models of other companies, the model repository has to provide individual rights by access control.

When distributed parties simultaneously work on the same models, conflicts arise that lead to inconsistencies. In order to prevent or repair conflicts, configuration management is to keep track of the different versions of the model. Furthermore, configuration management is to define which version of different models fit together.

Object-oriented database systems are best suited for implementing common model repositories, as they can efficiently handle graph-like model structures. Traditional file-based configuration management systems like CVS and SVN do not fit the needs of model-based development. Current configuration management systems for models like Odyssey-VCS mainly support a certain modeling language like UML [33]. However, there is also research on configuration management systems that can be parametrized by a modeling language (e.g., ModelCVS[7]).

b) Generic Editor Framework: A *front-end* provides a user interface for authoring models in the repository. The front-end should constitute a generic framework that can be parametrized by the applied modeling languages. The front-end provides editors to author a model in its concrete representation by using the concrete syntax of the modeling language. Furthermore, the front-end offers the operations to the modeler, which are defined by the modeling language (vertical) and operations that are common to all languages (horizontal).

These operations have to be intuitive to support the engineers in working with the models in an efficient manner. For example, the operations that support configuration management should allow the engineers to commit the changes on models and to update parts of the models. In case of a conflict, we need a merge operation that allows the visualization of the differences between models in their concrete syntax. The Eclipse platform is a perfect candidate for a front-end, as its service-oriented architecture makes it highly extensible.[8]

c) Workflow Engine: Our experiences show that a defined process is often not followed by its participants, as long as it is not supported by the modeling tool. To prevent deviation from the process, the developers should be guided through the defined process by the tooling platform. In order to operationalize the process, the workflow engine interprets the process definition of the modeling language. When interpreting a process model, the progress and current activities that need to be performed are always available through the workflow engine. To force a modeler to perform the current activities, all operations and interpreters not required for the activity have to be suppressed. The rights management of the tooling platform has to ensure that certain activities are only performed by certain roles. When modelers log on to the tooling platform, they can only perform activities that are currently available based on the process definition and that correspond to one of the roles they own.

d) Model Interpretation Engine: It provides the facilities to perform complex tasks such as analysis and synthesis based on the semantical definition of the language. To perform complex editing and refactoring facilities, an in-place model-to-model transformation engine is necessarily integrated in the front-end. For an automated generation of code and other process artifacts, an out-of-place model-to-model as well as model-to-text transformation engine is needed. Since such generation tasks might need a lot of time and computing power, they should be located at a different machine in the back-end. In order to be able to execute an operational semantics, a generic simulation framework is necessary, which also should be located in the back-end because of resource consumption issues.

[7]http://www.modelcvs.org/.
[8]http://www.eclipse.org/.

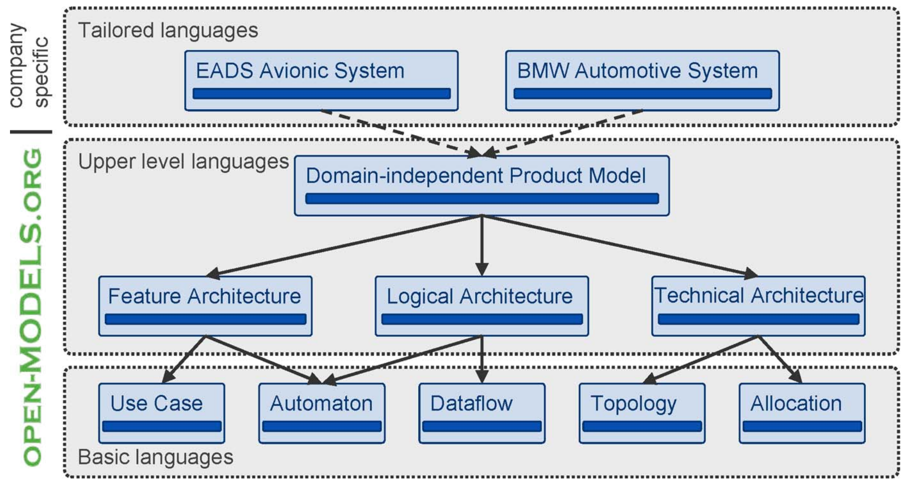

Fig. 8. ***Modularity in tooling: Dependencies between language modules.***

B. Building-Block Principle

To operationalize an integrated model theory for practice, a company may aim at defining an integrated modeling language that covers the whole development process. As a consequence, such an integrated modeling language is quite extensive and thus difficult to develop. The development costs are reduced by developing an integrated modeling language that is reused for several companies. However, this approach is usually not feasible, as a company may request a modeling language tailored to its specific needs.

Nevertheless, integrated modeling languages of different companies will be identical in some parts or similar in others. For example, many automotive companies prefer to use dataflow networks to model embedded systems. Reuse of these parts can be achieved by modularizing modeling languages. An integrated modeling language is then built by a number of predefined modeling language modules. As shown in Fig. 8, an organization composes existing modules (which may be provided by an open platform as planned for *open-MODELS.org*) for modeling requirements, software design, and deployment on hardware to form their own integrated modeling language.

A modeling language module consists of the elements that we have already described: abstract syntax, concrete syntax, process definition, and semantics. Additionally, a modeling language module needs to provide an interface so that it can be connected to other modules. A module for modeling software design, e.g., provides a connector for deployable units that can be connected to an appropriate connector in a module for modeling deployment on hardware. As we put the abstract syntax in the center of language development, the interface of a module is defined in terms of the abstract syntax.

Furthermore, companies might want to adapt a modeling language module to fit their specific requirements. Because of associated costs, companies do not want to rebuild the adapted modeling language from scratch. For this reason, a means has to be provided that allows for the customization of modeling language modules. There are several possibilities to do so: a language can be customized by constraints (lightweight extensions), subconcepts (heavyweight extensions), or parameters of the module. Customizing a modeling language results in a new module that depends on the module of the customized modeling language.

MOF provides basic coarse-grained operators for the composition of modeling languages like importing, merging, or combining packages [27]. Blanc *et al.* motivate the need for a new operator that allows one to reuse and generalize concepts when combining packages [34]. Clark *et al.* provide a new composition operator that allows one to equate concepts before merging the packages [35]. Karsai *et al.* propose more fine-grained operators that allow for the composition of modeling languages like the union of two concepts or finer control over inheritance relationships between two concepts [36]. Balasubramanian *et al.* show how to apply these operators to the integration of existing model-based development tools [37]. Estublier *et al.* provide similar constructs, but allow not only for the composition of the generated editors but also consider composition of corresponding model interpreters [38]. Emerson and Sztipanovits envision metamodel templates that enable a more flexible generalization and customization of modeling languages [39].

C. Managing Change for Modeling Languages

In order to be prepared for the inevitable evolution of modeling languages, appropriate tool support is required to safely change or extend a modeling language when already deployed [40]. In our tool architecture, the abstract syntax is first modified to fulfill the new requirements. As the other elements of a modeling language all depend on the abstract syntax, they have to be adapted to the modified abstract syntax and maybe extended with respect to the new requirements. Most importantly, however, existing models have to be migrated so that they can be used with the evolved modeling language.

Appropriate tool support is required for the migration of models in response to an evolving modeling language. As there may be a large number of models, model migration has to be automated. Further automation can be provided by reusing recurring migration scenarios. However, model migration becomes quite complex when motived by changes in the semantics of the modeling language. For this reason, appropriate tool support also needs to account for manual, expressive migrations.

With appropriate tool support for language maintenance, modeling languages can be even developed in an evolutionary way. A version of a modeling language is created and deployed to be assessed by the modelers. The feedback of the modelers is then easily incorporated into a new version of the modeling language, which is again deployed for further assessment. Elements of a modeling language other than the abstract syntax do not have to be defined in a first version of the modeling language but are completed in later versions. By evolutionary development of modeling languages, domain appropriateness can thus be reached iteratively.

When a specification changes, potentially all existing instances have to be reconciled in order to conform to the updated version of the specification. Since this problem of *coupled evolution* affects all specification formalisms (e.g., database or document schemata, types, or grammars) alike, numerous approaches for *coupled transformation* [41] of a specification and its instances have been proposed. The problem of schema evolution that has been a field of study for several decades has probably received the closest investigation [42]. Recently, the literature provides some work that transfers ideas from other areas to the problem of metamodel evolution. In order to reduce the effort for model migration, Sprinkle proposes a visual, graph-transformation based language for the specification of model migration [43]. Gruschko *et al.* envision to automatically derive a model migration from the difference between two metamodel versions [44], [45]. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [46].

V. POLITICAL BARRIERS

As described before, to achieve a seamless and integrated development environment, various integration activities have to be mastered. Up to now, this vision is not implemented in the industrial practice, although its realization is feasible from a scientific point of view. The reason for this lack of realization seems to be political. Those who require an integrated development environment—mainly the tool users—do not feel responsible for driving the integration. Those who seem to be responsible about tooling—mainly the tool vendors—do not profit.

A. The Different Stakeholders

For a better understanding, we distinguish the following stakeholders with their own interests.

1) Tool Vendors: In general, tool vendors want to sell their software tool products as often as possible with a minimum amount of (unpaid) customization activities. This results from the fact that software product companies have to deal with huge fixed costs, which makes it difficult to gain the return on investment. Every customization and every product redesign increases the development costs. Since a deep integration of tools affects most parts of their implementation, and in particular established companies have to handle a lot of legacy code, tool vendors in general are unreceptive for integration activities. In addition, tool integration activities come along with standardization, which makes the different tool vendors replaceable and thus decreases their customers' dependence. On closer examination, we can distinguish two types of tool vendors: vertical and horizontal vendors.

Vertical Tool Vendors: Horizontal tool vendors provide a concrete development tool for a dedicated purpose. There are many examples like Doors or Matlab-Simulink/Stateflow. At first, the benefit they provide depends on an integrated and seamless development environment. Even though integration will increase the benefit because of reuse aspects, vertical tool vendors fear the increased compatibility to their competitors. Since the lack of integration exists for all of the tool alternatives, vertical tool vendors are not forced to act. The Eclipse Platform may be one example for such an enforcement—once a critical mass of engineering tools is created using Eclipse, the other vendors will be forced to migrate to more open platforms. Another example is AUTOSAR, which is about to become the de-facto standard for automotive software engineering platforms. In both cases, the communities of potential tool users might be large enough. One of the few reasons a vertical tool vendor uses standardized platforms without pressure is the ability to replace proprietary in-house implementations by general-purpose software: Often the tool vendor is forced to implement a special solution for horizontal tooling aspects like configuration management, although it is out of the vendor's scope, because no appropriate solution is available on the market. After time, a proper solution will come up, and the tool vendor is willing to integrate the solution in order to minimize maintenance costs.

Horizontal Tool Vendors: Vertical tool vendors focus exactly on the general-purpose solutions that are independent of a dedicated modeling language. The above-mentioned configuration management is an example for this. Another example out of practice is database management systems.

Most development tools that provide a central database are using off-the-shelf databases. On the other hand, there are companies like MetaCase [47] that are still working with their own (object-oriented) database implementation because an appropriate solution is still not available. The difficulty horizontal tool vendors have to fight with is the fact that a horizontal tool itself does not provide a value to its users until the vertical solutions utilize them. To convince a vertical tool vendor to use a new horizontal solution in turn takes much effort. Often the issues are not technologically motivated but politically again: the vertical tool vendor becomes dependent on the horizontal tool vendor. Especially new and technically thrilling solutions are mainly introduced by young and small companies. The risk that such a company disappears from the market is critical for the utilizing vertical tool vendor. These issues make it necessary that a horizontal tool vendor is replaceable. Then, the risk for companies that integrate a horizontal solution is much lower. Because of that, horizontal tool vendors are much more interested in supporting a standardized tooling architecture than vertical tool vendors.

2) Tool Users: Generally speaking, the core business of tool users is not creating tools but using them in order to improve their system engineering process (decrease development costs and/or time, increase quality, etc.). Since developing their own development tools always leads to huge development and maintenance costs, most of the tool users try to strictly avoid proprietary tool solutions. The reason why tool users think about their own tool solutions, anyway, is given by the mismatch of required tools and provided tools. Also the integrative aspect of tooling is one of the most desired tool requirements that are not satisfied yet. Anyway, tool users do not feel in charge for establishing an integrated and generic tooling platform, although they are the first and only beneficiaries. The justification is that they are not a tool vendor, which is obviously correct but does not bring them closer to a solution. A solution might be that tool users only specify (and do not implement) a common tooling platform in order to be able to communicate their detailed requirements to the tool vendors. In the context of the automotive and avionic domain, we have to distinguish two kinds of tool users: OEM (integrators) and suppliers. From a technical point of view, *OEMs* focus on defining and integrating all supplied components for the final product (e.g., an airplane or a vehicle). Thus, they mainly concentrate on the early (requirements management) as well as late development phases (integration and system testing). *Suppliers* focus on developing a dedicated subsystem based on a given set of requirements. Consequently, according to the OEM, they work on the middle development phases (system design, implementation, and unit testing). Generally speaking, both OEMs and suppliers are interested in homogeneous tool environments. Nevertheless, the following facts have to be taken into account:

First, since systems engineering asks for an iterative development process including both OEM and suppliers, the interface between OEM and suppliers should be taken into account explicitly when talking about seamless development environments. Hereby, an important issue is the tradeoff between a seamless reuse of process artifacts and the protection of respective intellectual properties. For instance, one central database for both OEM and suppliers would be unacceptable. Instead, a distributed solution with a mature access control system may be a solution.

Secondly, every OEM collaborates with multiple suppliers, and every supplier collaborates with multiple OEMs. Thus, already as a matter of principle, neither the supplier can adopt the (potentially differing) tooling environments from their OEMs, nor can the OEM adopt the (potentially differing) tooling environments from their suppliers. Hence, the benefit of establishing a new tooling environment should not be a seamless tool integration between OEM and supplier in the first step. Instead, the primary goal should be an internal tool homogenization. Later on, other suppliers and OEMs, respectively, can be involved.

Finally, seamless tool environments are often seen as a major competitive advantage. Consequently, establishing cooperation between competitive OEMs and suppliers, respectively, is often difficult or even impossible. In this situation, tool users overlook the fact that integrations always have a standardization aspect, which inevitably requires cooperation even between the competitors in the end.

B. Levels of Integration

After our discussion about the involved stakeholders for a seamless tool environment, we outline the three possible and also necessary levels of integration on which the different stakeholders have to agree.

1) Common Tooling Platform: On this level, the stakeholders have to agree on all the horizontal tooling aspects independently of a concrete modeling language like persistency, access control, or configuration management. In particular, on this level, the meta languages for language construction must be defined. This is one of the most essential issues: without this special language, modeling languages are specified in different language specification formalisms. Imagine that one language is specified by an EBNF, another by a MOF Diagram. and a third by an XML Schema. A deep integration of such languages fails already because of the different formalisms. Tool users should be aware of the fact that informal or even semiformal language definitions are not sufficient: Tool vendors use their own interpretations of the language based on their needs. As a consequence, tools cannot interchange informations with each other, even though the language is standardized. That is why standards must provide formalizations of languages, e.g., reference implementations of languages. A prominent example for that problem is the Unified Modeling Language (UML). Even though the metamodel and the exchange format was standardized in an informal document, UML tools cannot exchange data correctly because each tool producer interprets the standard in a slightly different way. That is caused also by the complexity of the UML language itself. That is why the Eclipse community started to provide a reference implementation of the UML metamodel that enables the tools to exchange data correctly.

2) Common Modeling Languages: Once it is clear how to define modeling languages, the next level is to define modeling languages that various stakeholders have in common. In the automotive domain, AUTOSAR is a good example for such a standardization process. But due to the missing common tooling platform, the resulting AUTOSAR tools from different tool vendors again are not fully compatible. When discussing common modeling languages, the question arises: who of the stakeholders should be involved? Even though the tool users know best what is needed within a modeling language, the languages are defined mainly by the tool vendors. Tool vendors and tool users are almost decoupled and—with few exceptions generated by ad-hoc needs—there is no feedback between vendors and users. On this level, language customization should also be taken into account: Most of the tools today are general-purpose and off-the-shelf that can be used in a wide spectrum of domains (e.g., to develop vehicles, airplanes, or even medical instruments). On the other hand, today's tools usually cannot be customized. In addition, companies often try to protect their own languages from competitors, which makes it impossible to create seamless modeling environments. Instead of being isolated, languages should be extensible. The competitive advantage is the methodological knowledge of the staff about the language: It is much more important that a company *know* how to use a modeling technique correctly than just *have* one. The final consequence might be an open online platform for collaborative development of common modeling languages.

3) Common Artifacts: At last, concrete artifacts (modeled with the already defined and common modeling language) are used in common. In this context, data mining over divisions and company barriers becomes crucial: even though artifacts are used in common, the intellectual property must be protected further on. On this level, model reuse and product lines play also an important role. Although at this most concrete integration level the benefit is maximal, the above-mentioned more abstract integration layers must be handled before a mature solution is obtained.

VI. MIGRATION TO A BRAVE NEW TOOLING WORLD

A comprehensive approach for seamless model-based development cannot be introduced into practice in only one step. Instead of aiming at big-bang changes that would need to overcome enormous political and technical barriers, we need well-planed and incremental migration scenarios.

A. Incremental Rather Than Big Bang

We are aware that a "Big Bang"-like approach for changing the current practice with a pure seamless model-based approach as we described earlier is not feasible. Below we enumerate the most important factors that hinder the one-step transition:

1) Weak Semantic Domain: Many of the current commercial modeling tools are only partially formalized (if at all). Without a well-defined theory for the underlying languages, any model-based approach can be only partially successful and only implemented in an ad-hoc manner. The lack of formalization hampers automation of the workflow and the integration of models.

2) Conceptual Mismatch: The existing modeling techniques do not allow for a lean integration with each other due to their impedance mismatch at the conceptual level. Even if a specific modeling theory fits well for a particular process phase, it is not clear how it can be integrated into the engineering process. The lack of integration between modeling theories behind the tools turns the tools into isolated islands of automation.

3) Lack of Technical Agreement: The current architectures of the widely used system engineering tools are opaque towards the integration. Most of the time, the models are saved in proprietary formats and the tools are closed towards extensions or have undocumented APIs. Even the implementation of the most simple tool extensions requires digging into undocumented code and the usage of different scripting languages.

4) Huge Tool Development Costs: Building the tools that are mature enough to sustain the system engineering process is an expensive business. Many of the implementation costs are related to the infrastructure of the tools, and that is not directly reflected in the user functionality. Consequently, with few exceptions, the tool builders are interested to address a wide category of users (possibly from different domains) and thus build rather general tools.

B. Migration Scenario

We advocate that the most important mind setting is to plan and introduce model-based development incrementally. The incremental concepts should occur on distinct dimensions: from bridging the gap between the generality and domain appropriateness of tools, from peer-to-peer data integration to a central repository, from isolation of tools to an integrated environment, and from tool users to tool providers.

a) From general-purpose to domain-specific Tools: Ideally, engineers should work with tools that are tailored to their specific needs. Until we achieve this level of tailoring, the general-purpose tools should allow more enhanced customizations. The users of tools (or the IT departments from their companies) should be enabled to customize the COTS tools and thereby to bridge the gap between the general facilities offered by the tools and the very specific needs of the engineers. By allowing advanced customization, the gap between tool users and builders can be bridged—tool users become active contributors to the tools themselves. By taking customization to extreme, domain-specific tools can be obtained by parameterizing the general-purpose tools with different workflows and language definitions (or language profiles à la UML).

b) From peer-to-peer Data Integration to an Integrated Repository: Part of the current difficulties in integrating tools are due to the lack of standard interfaces that the tools define to access their data. The data integration approaches range between loose, peer-to-peer integration with the help of tool couplers to an integrated repository that contain all the product data. As an intermediate step, we envision the agreement to use public formats for exporting models and the publishing of open APIs by tool builders.

c) From Federations of Tools to Integrated Environments: In an ideal case, all system engineering tools would be integrated in an system engineering environment that would offer different views onto the product model. In the current state of practice, engineers make use of tool chains that only partially satisfy their needs. These tools do not know of each other, and the borders of tools are very sharp (e.g., when they are working with more tools, engineers need to change completely their working context by switching between different programs). As an intermediate step, we envision that the presentation layers of tools are coupled through plug-ins and in this way the boundaries between tools will disappear—a tool A can be a prolongation of a tool B, and vice versa. However, in order to support this, the tool builders should change the design and architecture of tools towards more open systems. Each tool will be a contributor to a repository-based system engineering environment rather than standalone.

d) From building Tools from scratch to a Common Tooling Infrastructure: The vertical aspects of tools are the most interesting ones for their clients and can make the difference when judging the capabilities of a certain tool. The horizontal aspects represent a great deal of effort and do not differ dramatically from domain to domain. Having a standard infrastructure enables the basic management of models, and thereby the efforts for implementing new tools can concentrate on the user functionality. This will subsequently cause a reduction of development costs for tools. As a side effect, this leads to a standard manner to build and manage the models. All the horizontal operations are done in the same manner (e.g., similar to the current de-facto standard content of the "File," "Edit," or "Help" menus of every tool). Beside the front-end, the back-end of tools is standardized, and thereby all the system engineering tools are built on a common tooling infrastructure.

e) From individual Companies to an Industry Consortium: Defining clear minimal requirements about the tools that are industry-specific could represent a big step forward and could establish a common vocabulary between the tool vendors and customers on the one hand and between the integrators and suppliers on the other hand. Such standards can be in the form of profiles (e.g., an ATA profile that leverages the general requirements-management capabilities of Telelogic DOORS to be aware of the avionics way of organizing the requirements).

VII. RELATED WORK

A. Tool Integration Approaches

In the literature, we can already find some approaches for tool integration. In [48], the authors propose a model-based approach to integrate tools working on interdependent documents. Wrappers for each tool allow us to abstract from technical details and provide homogenized access to documents through graph models. The different documents are kept consistent by graph transformations rules, which allow us to propagate changes in an incremental development process. In [49], the authors give a more detailed description of their algorithm for incremental and interactive consistency management. The authors of [50] explain and compare two architectural design patterns that allow for tool integration. The first architecture is based on an integrated model and adapters for each tool, which translate the data to the integrated model. The second architecture is based on a messaging system, which routes data according to a workflow specification, and implements a pairwise integration among tools. [51] presents the extension of the Electronic Tool Integration platform with Web service technology. The integrated tools interact with each other by use of Web services, which allow one to decouple the different tools from each other and therefore ease integration and maintenance activities. In [52], the authors present their rule-based approach Multi Document Integration for data integration of multiple data repositories. Metamodels are used to provide an abstract specification of the different models, a separate model is used to specify correspondence links between the models, and rules are used to specify consistency between the models. The declarative rules that are specified in the form of triple graph grammars are used to derive code for creating and consistency checking of correspondence links as well as for forward and backward propagation of changes. TOPCASED [53] is an open-source CASE environment for model-based development of critical applications and systems. Their ambition is to build an extensible and evolutive CASE tool that allows its users to access various models and associated tools.

Besides these academic approaches, there are already some tool developers offering integrated tool support. For the automotive domain, Vector has developed the tool eASEE,[9] which is intended to be a data backbone that stores the product data in a central repository. This tool is not designed as a generic tool integration platform but focuses on supporting predefined modeling functionalities. The tool PREEVision from Aquintos[10] follows a similar direction.

B. Language Workbenches

There are many standards from the OMGlike the MOF for the definition of metamodels, OCL [28] for defining constraints on MOF-based metamodels, and Query/Views/ Transformations specification [54]. Only partially based on these standards, many so-called Language Workbenches have been developed to enable the development of mainly diagrammatic domain-specific languages (DSLs). The most prominent ones are the Generic Modeling Environment [55], MetaEdit+ [47], Microsoft's Domain-Specific Language Tools, and EMF. All of these tools offer support for building modeling languages. However, these languages are often trapped inside of these tool environments because they all implement different metamodeling techniques. The composition of languages to an integrated modeling chain is still only supported in a very limited way. These tools also lack support for the development workflow. But nevertheless, they represent tool architectures that separate the modeling language from the tool infrastructure.

C. Research Road Maps

Many authors that wrote software engineering research road maps have already stated the problem of getting to a seamless tool environment. In [15], the authors describe the use of domain abstractions as a key to future model-based development. These can be made usable by the implementation of DSLs. To prevent a DSL-Babel, the authors name the ability of relating elements in different languages with each other to allow the flexible composition of languages. The authors also describe evolution as one of the big future problems in model-based development. In [56], the problem of distributed development in the automotive industry is explained. A deeper integration of models and tools is claimed as a key issue in future automotive software engineering research. [57] outlines a research roadmap that should lead to easier software verification in the future. They also claim for semantically richer specification and modeling techniques. The need for model engineering is also claimed in [58] and [59]. The AMMA platform is presented as an Eclipse-based integration environment that allows basic (meta)model management support. This approach has the potential to fulfill some of the requirements for an integrated tool support presented in this paper. In [60], Zeller describes tool integration as one of the challenging tasks for software engineering. He argues that the extensibility of tools will become a crucial issue.

VIII. CONCLUSION

The full promise of model-based development will only be achieved by using models throughout the development process. Requirements models have to be refined to design models from which implementation models are generated. To reuse the model information from one process step within other process steps, a seamless integration of the different models is required. Seamless model-based development can only be achieved with three main ingredients: 1) a comprehensive modeling theory, 2) an integrated architectural model, and 3) a seamless model engineering environment.

The large body of research in the last 20 years has led to a wide body of knowledge about modeling theories and architectures. In current practice, however, model-based development finds its way into the industry with difficulties mostly because of the lack of adequate tool support. Working with models requires the tools to be aware of the semantics of models and that the tools can exchange the models; these requirements increase substantially the difficulty of developing tools. In order to tackle these problems and to enable a seamless integration of methods, models, processes, and tools, we proposed a new tooling platform. We advocate that a tooling platform should separate between common functionality related to the infrastructure (horizontal aspects) from the functionality that is specific to a modeling language (vertical aspects). The horizontal aspects enable an common model repository with explicit dependencies between the different models. The generic tooling platform can be parametrized by a modeling language that defines the product model of a company. The clear separation of vertical and horizontal tooling aspects forms the foundation to reduce costs for both development and maintenance of an integrated engineering environment. Currently we contribute to the open-source project *OOMEGA*[11] in order to demonstrate such a generic tooling platform. In a project, we use OOMEGA to implement seamless model-based development for the automotive domain. Based on OOMEGA, we set up the *openMODELS*[12] project, which enables a collaborative development of commonly used language modules. ■

[9]http://www.vector.com/vi_easee_en,,223.html.
[10]http://www.aquintos.info.
[11]http://www.oomega.net/.
[12]http://www.open-models.org/.

REFERENCES

[1] J.-R. Abrial, "Formal methods: Theory becoming practice," *J. Univ. Comput. Sci.*, vol. 13, no. 5, pp. 619–628, May 2007.

[2] J.-R. Abrial, "Formal methods in industry: Achievements, problems, future," in *Proc. ACM 28th Int. Conf. Software Eng. (ICSE '06)*, New York, 2006, pp. 761–768.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proc. 14th ACM SIGACT-SIGPLAN Symp. Principles Program. Lang. (POPL '87)*, New York, 1987, pp. 178–188.

[4] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars polar lander fault identification using model-based testing," in *Proc. 8th IEEE Int. Conf. Eng. Complex Comput. Syst. (ICECCS '02)*, Washington, DC, 2002, p. 163.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10 states and beyond," in *Proc. 5th Annu. IEEE Symp. Logic Comput. Sci. (LICS 1990)*, J. Mitchell, Ed., Jun. 1990, pp. 428–439.

[6] Y. Choi, "From NuSMV to SPIN: Experiences with model checking flight guidance systems," *Formal Methods Syst. Design*, vol. 30, no. 3, pp. 199–216, Apr. 2007.

[7] A. Betin Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp, "Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers," *Autom. Software Eng.*, vol. 14, no. 2, pp. 129–178, Jul. 2007.

[8] C. A. Muñoz, G. Dowek, and V. Carreño, "Modeling and verification of an air traffic concept of operations," in *Proc. 2004 ACM SIGSOFT Int. Symp. Software Test. Anal. (ISSTA '04)*, New York, 2004, pp. 175–182.

[9] S. Beyer, P. Bohm, M. Gerke, M. Hillebrand, T. I. D. Rieden, S. Knapp, D. Leinenbach, and W. J. Paul, "Towards the formal verification of lower system layers in automotive systems," in *Proc. 2005 IEEE Int. Conf. Comput. Design (ICCD '05)*, Washington, DC, 2005, pp. 317–326.

[10] J. Whittle, R. Kwan, and J. Saboo, "From scenarios to code: An air traffic control case study," *Software Syst. Model.*, vol. 4, no. 1, pp. 71–93, Feb. 2005.

[11] G. Zoughbi, L. Briand, and Y. Labiche, "A UML profile for developing airworthiness-compliant (RTCA DO-178B), safety-critical software," in *Model Driven Engineering Languages and Systems*, vol. 4735. Berlin, Germany: Springe, 2007, ser. Lecture Notes in Computer Science, pp. 574–588.

[12] OMG, *Unified Modeling Language: Superstructure*. OMG Doc. formal/07-02-05.pdf, 2 2007. [Online]. Available: http://www.omg.org/docs/formal/07-02-05.pdf

[13] M. Broy, M. L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic, "2nd UML 2 semantics symposium: Formal semantics for UML," in *Models in Software Engineering*, vol. 4364. Berlin, Germany: Springer, 2007, ser. Lecture Notes in Computer Science, pp. 318–323.

[14] M. L. Crane and J. Dingel, "UML vs. Classical vs. Rhapsody Statecharts: Not all models are created equal," *Software Syst. Model.*, vol. 6, no. 4, pp. 415–435, Nov. 2007.

[15] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Proc. 2007 IEEE Future Software Eng. (FOSE '07)*, Washington, DC, May 2007, pp. 37–54.

[16] A. v. Lamsweerde, "Formal specification: A roadmap," in *Proc. ACM Conf. Future Software Eng. (ICSE '00)*, New York, 2000, pp. 147–159.

[17] OMG, *Unified Modeling Language (UML) spec. 2.1.2*, 2006.

[18] N. G. Leveson, "Intent specifications: An approach to building human-centered specifications," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 15–35, 2000.

[19] "A380 cable problems threaten Airbus," *FlugRevue*. [Online]. Available: http://www.flug-revue.rotor.com/FRHeft/FRHeft06/FRH0612/FR0612b.htm

[20] T. Pop, P. Eles, and Z. Peng, "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems," in *Proc. 10th ACM Int. Symp. Hardware/Software Codesign (CODES '02)*, New York, 2002, pp. 187–192.

[21] P. Wallin and J. Axelsson, "A case study of issues related to automotive E/E system architecture development," in *Proc. 15th Annu. IEEE Int. Conf. Eng. Comput. Based Syst. (ECBS '08)*, Washington, DC, Mar. 2008, pp. 87–95.

[22] J.-M. Favre, "Languages evolve too! Changing the software time scale," in *Proc. 8th IEEE Int. Workshop Principles Software Evol. (IWPSE '05)*, Washington, DC, 2005, pp. 33–44.

[23] M. Broy, "Model-driven architecture-centric engineering of (embedded) software intensive systems: Modeling theories and architectural milestones," *Innov. Syst. Software Eng.*, vol. 3, no. 1, pp. 75–102, Feb. 2007.

[24] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Secaucus, NJ: Springer-Verlag, 2001.

[25] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild, "Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme," Technische Univ. München, Tech. Rep. TUM-I0816, Jun. 2008.

[26] J. Evermann and Y. Wand, "Toward formalizing domain modeling semantics in language syntax," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 21–37, Jan. 2005.

[27] OMG, *Meta Object Facility (MOF) spec. 2.0*, 2006.

[28] OMG, *Object Constraint Language (OCL) spec. 2.0*, 2006.

[29] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Upper Saddle River, NJ: Pearson Education, 2003.

[30] F. Jouault, J. Bézivin, and I. Kurtev, "TCS: A DSL for the specification of textual concrete syntaxes in model engineering," in *Proc. 5th ACM Int. Conf. Generative Program. Compon. Eng. (GPCE '06)*, New York, 2006, pp. 249–254.

[31] M. Alanen, T. Lundkvist, and I. Porres, "Creating and reconciling diagrams after executing model transformations," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 128–151, Oct. 2007.

[32] Z. Drey, C. Faucher, F. Fleurey, V. Mahé, and D. Vojtisek, *Kermeta Language—Reference Manual*. Rennes, France: IRISA, Jan. 2008. [Online]. Available: http://www.kermeta.org/docs/KerMeta-Manual.pdf

[33] H. Oliveira, L. Murta, and C. Werner, "Odyssey-VCS: A flexible version control system for UML model elements," in *Proc. 12th ACM Int. Workshop Software Config. Manage. (SCM '05)*, New York, 2005, pp. 1–16.

[34] X. Blanc, F. Ramalho, and J. Robin, "Metamodel reuse with MOF," in *Model Driven Engineering Languages and Systems*, vol. 3713. Berlin, Germany: Springer, 2005, ser. Lecture Notes in Computer Science, pp. 661–675.

[35] T. Clark, A. Evans, and S. Kent, "A metamodel for package extension with renaming," in *UML 2002-The Unified Modeling Language*, vol. 2460. Berlin, Germany: Springer, 2002, ser. Lecture Notes in Computer Science, pp. 305–320.

[36] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, "Composition and cloning in modeling and meta-modeling," *IEEE Trans. Contr. Syst. Technol. (Special Issue on Computer Automated Multi-Paradigm Modeling*, vol. 12, pp. 263–278, 2004.

[37] K. Balasubramanian, D. C. Schmidt, Z. Molnar, and A. Ledeczi, "Component-based system integration via (meta)model composition," in *Proc. 14th Annu. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst. (ECBS '07)*, Washington, DC, Mar. 2007, pp. 93–102.

[38] J. Estublier, G. Vega, and A. D. Ionita, "Composing domain-specific languages for wide-scope software engineering applications," in *Model Driven Engineering Languages and Systems*, vol. 3713. Berlin, Germany: Springer, 2005, ser. Lecture Notes in Computer Science, pp. 69–83.

[39] M. Emerson and J. Sztipanovits, "Techniques for metamodel composition," in *Proc. 6th Workshop Domain Specific Model. (DSM'06)*, Oct. 2006, pp. 123–139. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/289.html

[40] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Automatability of coupled evolution of metamodels and models in practice," in *Model Driven Engineering Languages and Systems*, vol. 5301. Berlin, Germany: Springer, 2008, ser. Lecture Notes in Computer Science, pp. 645–659.

[41] R. Lämmel, "Coupled software transformations (extended abstract)," in *Proc. 1st Int. Workshop Software Evol. Transform.*, Nov. 2004.

[42] E. Rahm and P. A. Bernstein, "An online bibliography on schema evolution," *SIGMOD Rec.*, vol. 35, no. 4, pp. 30–31, 2006.

[43] J. Sprinkle and G. Karsai, "A domain-specific visual language for domain model evolution," *J. Vis. Lang. Comput.*, vol. 15, no. 3–4, pp. 291–307, Jun. 2004.

[44] S. Becker, T. Goldschmidt, B. Gruschko, and H. Koziolek, "A process model and classification scheme for semi-automatic meta-model evolution," in *Proc. Workshop 'MDD, SOA IT-Manage. 2007*, 2007.

[45] B. Gruschko, D. Kolovos, and R. Paige, "Towards synchronizing models with evolving metamodels," in *Proc. Int. Workshop Model-Driven Software Evol.*, 2007.

[46] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *Proc. 21st Eur. Conf. Object-Oriented Program. (ECOOP'07)*, 2007, vol. 4609, pp. 600–624.

[47] J.-P. Tolvanen, "MetaEdit+: Domain-specific modeling for full code generation demonstrated [GPCE]," in *Companion 19th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Applicat. (OOPSLA '04)*, New York, 2004, pp. 39–40.

[48] S. M. Becker, T. Haase, and B. Westfechtel, "Model-based a-posteriori integration of engineering tools for incremental development processes," *Software Syst. Model.*, vol. 4, no. 2, pp. 123–140, May 2005.

[49] S. M. Becker, S. Herold, S. Lohmann, and B. Westfechtel, "A graph-based algorithm for consistency maintenance in incremental and interactive integration tools," *Software Syst. Model.*, vol. 6, no. 3, pp. 287–315, Aug. 2007.

[50] G. Karsai, A. Lang, and S. Neema, "Design patterns for open tool integration," *Software Syst. Model.*, vol. 4, no. 2, pp. 157–170, May 2005.

[51] T. Margaria, "Web services-based tool-integration in the ETI platform," *Software Syst. Model.*, vol. 4, no. 2, pp. 141–156, May 2005.

[52] A. Königs and A. Schürr, "MDI: A rule-based multi-document and tool integration approach," *Software Syst. Model.*, vol. 5, no. 4, pp. 349–368, Nov. 2006.

[53] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel, "The TOPCASED project: A toolkit in open source for critical aeronautic systems design," in *Embedded Real Time Software (ERTS)*, 2006.

[54] OMG, *Query/View/Transformation (QVT) spec. 1.0*, 2008.

[55] J. Davis, "GME, the generic modeling environment," in *Companion of the 18th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Applicat. (OOPSLA '03)*, New York, 2003, pp. 82–83.

[56] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *Proc. 2007 IEEE Future Software Eng. (FOSE '07)*, Washington, DC, May 2007, pp. 55–71.

[57] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump, "Roadmap for enhanced languages and methods to aid verification," in *Proc. 5th ACM Int. Conf. Generative Program. Compon. Eng. (GPCE '06)*, New York, 2006, pp. 221–236.

[58] J. Bezivin, F. Jouault, and D. Touzet, "Principles, standards and tools for model engineering," in *Proc. 10th IEEE Int. Conf. Eng. Complex Comput. Syst. (ICECCS '05)*, Washington, DC, 2005, pp. 28–29.

[59] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based DSL frameworks," in *Companion to the 21st ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Applicat. (OOPSLA '06)*, New York, 2006, pp. 602–616.

[60] A. Zeller, "The future of programming environments: Integration, synergy, and assistance," in *Proc. IEEE Future Software Eng. (FOSE '07)*, Washington, DC, May 2007, pp. 316–325.

ABOUT THE AUTHORS

Manfred Broy (Member, IEEE) received the Ph.D. and Habilitation degrees in mathematics and computer science from Technische Universität München, Germany, in 1980 and 1982, respectively.

From 1983 to 1989, he was a full Professor of computer science and Founding Dean in the Faculty of Mathematics and Computer Science, University of Passau. He then became a full Professor of computer science in the Faculty of Computer Science, Technische Universität München. His research interests are software and systems engineering comprising both theoretical and applied aspects including system models, specification and refinement of system components, specification techniques, development methods, and verification.

Prof. Broy is a member of the European Academy of Sciences and the Deutsche Akademie der Naturforscher "Leopoldina." In 1994, he received the Leibniz Award from Deutsche Forschungsgemeinschaft and in 2007 the Konrad Zuse Medal from Gesellschaft für Informatik.

Martin Feilkas is pursuing the Ph.D. degree at Technische Universität München, Germany.

He is a Research Assistant with the Software and Systems Engineering group, Technische Universität München, where he has been involved in several research projects. His research interests include model-driven engineering of embedded systems and language engineering.

Markus Herrmannsdoerfer is pursuing the Ph.D. degree at Technische Universität München, Germany.

He is a Research Assistant with the Software and Systems Engineering group, Technische Universität München. His academic interests include model-driven engineering, language engineering, and language evolution.

Stefano Merenda is pursuing the Ph.D. degree at Technische Universität München, Germany.

His previous studies focused on formal languages and data modeling. Since 1993, he has been working on several software and hardware engineering projects. Since 2003, he has been concentrating on *OOMEGA*, which provides a framework for model-based software engineering tools. Since 2005, he has been a Research Assistant at Technische Universität München. He focuses on formalized metamodeling approaches, generic engineering frameworks, and the design of product models. Since 2008, he has been Head of the competence center Product Data Modeling and currently establishes the platform *open-MODELS.org*.

Daniel Ratiu received the Ph.D. degree from Technische Universität München, Germany.

He is a Researcher with the Software and Systems Engineering group, Technische Universität München. His research interests include domain modeling, domain-specific languages, and language engineering.