SPES 2020 Deliverable 1.4.B-3

Concepts for an Integrated Tool Architecture

Author: Martin Feilkas
        Markus Herrmannsdoerfer
        Florian Hoelzl
        Stefano Merenda
        Daniel Ratiu
        Wolfgang Schwitzer

Version: 1.0
Date: October 29, 2009
Status: Released
About this Document

More than twenty years of research have 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 industry is still limited. Most of the times, the engineers work with a pre-defined set of isolated tools, and therefore adapt their engineering and process to the available tools. Today, the industry achieves tool integration by demand-driven, pragmatic and ad-hoc composed chains of a priori existing commercial tools. Nevertheless, these tool chains are not (and cannot be) seamless, since the achieved integration 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, allow the authoring of the product model and guide the engineering process. 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 our first attempt to build such an integrated engineering environment.

Outline. In Section 1, we lay out the current bottom-up approach to integrate existing tools in industry, and contrast it with a top-down approach. Due to the current practice, a number of pressing issues arise which we outline in Section 2. To overcome these pressing issues, we introduce in Section 3 our vision of a solution by an integrated engineering environment. Section 4 explains from a technical point of view how such an integrated engineering environment could become reality and analyzes which technical ingredients are already available. The architecture of AutoFOCUS 3 which is our prototypical implementation of such an integrated engineering environment is outlined in Section 5. Finally, Section 6 summarizes related work on integrated engineering environments before we conclude in Section 7.
Contents

1 Introduction 4

2 Pressing Issues of current Tool Chains 7
  2.1 High generality and inappropriateness of modeling languages 8
  2.2 No integrated architectural Model 9
  2.3 Insufficiently integrated Engineering Environments 10

3 Vision of Seamless Model-based Development 13
  3.1 Semantic Domain 14
  3.2 Integrated architectural Model 14
  3.3 Integrated Model Engineering Environment 15

4 General Concepts for an Integrated Tool Architecture 16
  4.1 Separation of horizontal and vertical Tooling Aspects 17
    4.1.1 Vertical Aspects 18
    4.1.2 Horizontal Aspects 21
  4.2 Building-Block Principle 22
  4.3 Managing Change for Modeling Languages 24

5 AutoFOCUS 3: Prototypical Implementation of an Integrated Tool Architecture 25
  5.1 Example: Describing heterogeneous Hardware by Language Modules 27
  5.2 Case Studies and Future Work on AutoFOCUS 28

6 Related Work 29
  6.1 Tool Integration Approaches 29
  6.2 Language Workbenches 29

7 Conclusion 31

References 32
1 Introduction

Today, model-based development is adopted more or less consequently for practical development of systems in various domains (e.g., automotive, avionic, and automation systems). 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 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 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 Figure 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 a lot of 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 which supports the reuse of the information that is captured within the models. To achieve the vision of seamless model-based development (as illustrated in Figure 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 which 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 which 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.

![Diagram of integrated model-based engineering environment]

**Figure 2: Vision of integrated model-based engineering environment.**

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 Figure 3(a)). In reality, however, a small set of tools (most of the times 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 Figure 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, the 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 the 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.
Situation today

Ideal situation
-well-defined modeling theory
-common product model
-appropriate process model

Tool support

Situation today
-no modeling theory
-unclear product model
-ad-hoc imposed process

Currently available tools

(a) wanted
(b) unwanted

Figure 3: The tyranny of current tools.
2 Pressing Issues of current Tool Chains

We developed a questionnaire to get an overview over the situation and the needs of tools for model-based development in industry today. We distributed this questionnaire to all SPES industry partners, and received 24 filled out questionnaires. The results [HM09] indicate the following current problems which hamper the seamless integration of tools:

- **Heterogeneity of target platform.** The target platform onto which the embedded system is deployed is affected by a considerable heterogeneity. The target platform varies a lot from project to project in terms of both programming language, bus system, operating system, and middleware. As a consequence, the development tools are also affected by heterogeneity, as each of them is usually specialized for a certain target platform. Therefore, the heterogeneity of target platform heavily hampers the seamless integration of the tools.

- **Heterogeneity of tools.** There is also an enormous heterogeneity in the tools that are used for the model-based development of embedded systems. More specifically, the tools vary a lot for both the development phases like requirements management, modeling, and quality assurance, and for the crosscutting functionality like tool platform, version and change management. The heterogeneity hampers the seamless integration of the tools, as the different tools are usually based on different technologies.

- **Weakly defined models.** Tools which are based on models without a precise meaning are still widely used in industry. For requirements management, the companies still rely a lot on Microsoft Word and Excel which are not completely adequate for modeling requirements. The situation is much better for system modeling with a high usage of modeling tools such as Matlab Simulink and Scade. For process support, a lot of tool support is based on Microsoft Project and Excel which do not allow for the rigorous definition of a process. Models with a precise meaning can be better analyzed, operationalized, and integrated with models from different development phases or different subsystems.

- **Sporadic usage of model-based development.** The tool chains used in practice do not yet provide the full benefits of model-based development. Even though the companies use a lot of advanced modeling tools like Matlab Simulink and Scade, a lot of code is still handwritten instead of generating it from the models. For quality assurance, the companies still use a lot of manual techniques like manual testing and code review, and do not rely on automatic techniques like model checking and theorem proving. These results show that current tool chains are not yet seamless enough to realize the promises of model-based development.

- **High rate of proprietary tools.** The companies make use of a lot of proprietary tools which are especially developed for them. The results show us that proprietary tools are developed for all process phases. Although a huge number of development tools are available, the market does not seem to provide solutions satisfying the requirements. This may be also due to the missing customizability of existing tools to the individual requirements. In particular, most effort is spent on the integration of existing tools which is very important for seamless model-based development.
The results show that current tool chains are and cannot be seamless enough to benefit from the promises of model-based development. When looking at the needs for future tools, the following features are considered the most important:

- **Importance of tool integration.** A deep integration of the development tools is seen as the major issue. Most effort for proprietary tool development is spent on coupling of existing tools. Moreover, the missing integration of the tools heavily restricts the developers in their daily work. Currently, only very few tool chains are fully integrated on different levels of integration. However, tool integration is considered as important on nearly all levels of integration.

- **Importance of process support.** Support of the development process is seen as an important feature which is not fully satisfied by current tools. Currently, the companies mainly use Microsoft Process and Excel for process support which do not allow them to operationalize the process. The missing process support restricts a lot of the developers in their daily work. Today, only very few tool chains are fully integrated by means of process support, but however more integration is desired. This is probably due to the fact that integrated process support also requires tool integration which is also missing today.

In the following, we characterize and analyze pressing issues concerning current tool chains, which lead to the problems mentioned above. We ground this section both on the literature, and on our experience gathered from working with industrial partners.

### 2.1 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 [FR07]. 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 that they use should directly support the development of that product and nothing more. Unfortunately, the modeling languages are highly general and most of the times domain independent and ontologically neutral [vL00]. 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 pre-defined 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 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).
Lacking evolvability of modeling languages. Although often neglected, a modeling language is subject to change like any other software artifact [Fav05]. 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.

2.2 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. Complementing 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 times, however, the semantic integration of modeling techniques is not clear. For example, UML 2.0 [OMG06c] defines 13 types of diagrams for different development stages (e.g., use-case diagrams, activity diagram, component diagrams, deployment diagrams). Even if these many views allow a better development of different aspects of the system, once different system views are constructed, it is not clear how they are to 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, formal verification, checking quality aspects that cross-cut models, etc.

Managing the intent. Due to the isolation of models, the intent and rationale behind component designs is lost in the process [Lev00]. 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.

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 to 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 [A38]. 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.

2.3 Insufficiently integrated Engineering Environments

Many problems arise due to tooling issues such as missing tool integration and cumbersome handling of models. For example, in Figure 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 semi-structured 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 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.

![Figure 4: An example of today's tooling situation in automotive software development.](image)

**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 times 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.

**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.

**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, for instance, the supplier usually has the so called integration responsibility. This means that he is responsible to deploy several applications onto an Electronic Control Unit (ECU), which is the deliverable to the 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 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 like it 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) as well as 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 [PEP02] becomes an important method for verification and quality assurance.

**Securing intellectual property rights.** Distributed development especially requires a sophisticated rights management. In [WA08] 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 only thing that OEMs do 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 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.
3 Vision of Seamless Model-based Development

Seamless model-based development promises to lift software development onto 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 it is a paradigm that provides support throughout the entire development and maintenance life cycle. 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 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 [Bro07].

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 Figure 5. These ingredients can be divided into three levels: the semantic domain forms the basis of an integrated architectural model which is operationalized by a model engineering environment. In the following, we detail on these levels and their corresponding ingredients.

![Figure 5: Main ingredients for seamless model-based development.](image-url)
3.1 Semantic Domain

Seamless model-based development requires a comprehensive modeling theory as theoretical basis to ensure a thorough formalization of all artifacts produced during the development of a system. An appropriate modeling theory provides firstly 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, and
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.

and secondly 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, and
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, as well as 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 [BS01] and its various extensions.

3.2 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 [BFG+08].

**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.
Product model. In order to describe all the relevant 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 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.

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.

3.3 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 as well as techniques for generating further development artifacts from models. 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.
4 General Concepts for an Integrated Tool Architecture

To 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 expensive, the industry sees no alternative to existing commercial tools. These tools are many times 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 beside the core business functionality, tools need a lot of infrastructure for the management of models. Figure 6 shows that a development tool can be decomposed into the four parts Repository, 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 encapsulated in Language Modules.

![Figure 6: Decomposing development tools.](image)

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 horizontal. We performed our empirical study based on the description of menus and toolbars command buttons 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 COTS 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 most of it could be provided by a generic tools platform. We also can see that none of the tools provides an integrated workflow support.

<table>
<thead>
<tr>
<th></th>
<th>Scade</th>
<th>Rhapsody</th>
<th>Doors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Persistency</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>- Configuration Management</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Editors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Editing</td>
<td>107</td>
<td>95</td>
<td>40</td>
</tr>
<tr>
<td>- Navigation</td>
<td>24</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>- Layout</td>
<td>15</td>
<td>43</td>
<td>12</td>
</tr>
<tr>
<td>Analysis/Synthesis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Analysis</td>
<td>29</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>- Synthesis</td>
<td>14</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>196</td>
<td>165</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 1: Quantitative overview over the functionality of COTS.

However, it is in particular the heterogeneous implementation of this infrastructure in different tools which hinders their seamless integration. To counteract this situation, we propose a generic tooling platform which offers all the technical details which 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 which can be customized to the specific needs of the engineers. Furthermore, appropriate tool support is required for model migration in order to be able to improve a modeling language that is already under use.

4.1 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 which are independent of a specific modeling language are termed horizontal. Tooling aspects like the syntax of a certain modeling language which 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. Figure 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.

Figure 7: Horizontal and vertical tooling aspects of an integrated engineering environment.

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

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 syntactically correct models to those that are meaningful in the domain [EW05]. 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) [OMG06a] for the constructive part and the Object Constraint Language (OCL) [OMG06b] for the descriptive part. However, MOF provides too many constructs to be completely understood and implemented. The most widely known implementation of a subset of MOF called EMOF (Essential MOF) is the Eclipse Modeling Framework (EMF) [BSM03].

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 sub components of a 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 the 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.

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 [JBK06]. The Graphical Modeling Framework (GMF) provides a language to specify a diagrammatic syntax for EMF models and allows to generate an authoring tool from that specification [Foua]. Diagram Interchange Mapping Language (DIML) 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 [ALP07]. 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.

**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 which 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 not only depend 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.

**Semantics.** Generally, there are three ways of specifying semantics: The first one is to describe the semantics of the modeling language by a calculus (axiomatic semantics), 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 which 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 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 [DFF+08] is aiming at such a solution.

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

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 which 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 [OMW05]. However, there is also research on configuration management systems which can be parametrized by a modeling language (e.g. ModelCVS [Mod]).
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 [Foub].

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 which are currently available based on the process definition and which correspond to one of the roles they own.

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 a 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 should be also located in the back-end because of resource consumption issues.

4.2 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, a lot of 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 Figure 8, an organization composes existing modules for modeling requirements, software design and deployment on hardware to form their own integrated modeling language.

![Figure 8: Modularity in tooling: Dependencies between language modules.](image)

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 which 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), sub concepts (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 [OMG06a]. Blanc et al. motivate the need for a new operator that allows to reuse and generalize concepts when combining packages [BRR05]. Clark et al. provide a new composition operator that allows to equate concepts before merging the packages [CEK02]. 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 [KML+03]. Balasubramanian et al. show how to apply these operators to the integration of existing model-based development tools [BSML07]. 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 [EVI05]. Emerson and Sztipanovits envision metamodel templates that enable a more flexible generalization and customization of modeling languages [ES06].

4.3 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 [HBJ08]. 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 motivated 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 [Läm04] of a specification and its instances have been proposed. The problem of schema evolution which has been a field of study for several decades has probably received the closest investigation [RB06]. 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 [SK04]. Gruschko et al. envision to automatically derive a model migration from the difference between two metamodel versions [BGGK07, GKP07]. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [Wac07].
5 AutoFOCUS 3: Prototypical Implementation of an Integrated Tool Architecture

History. AutoFOCUS is a CASE tool that is developed at the Technische Universität München since 1996. The tool was an implementation of tool support for the formal method FOCUS [BS01]. AutoFOCUS demonstrated that formal methods can be applied to practical engineering, if they are built into an easy to use development environment. The major capability of AutoFOCUS was the graphical modeling of networks of logical components (system structure), automatons for the specification of dynamic behavior and of course data types. As AutoFOCUS was based on a well defined semantic foundation, a simulator has been implemented which made it possible to do early validation and verification of AutoFOCUS systems. In 1999, AutoFOCUS has won the Tool Competition at the Formal Methods conference. Based on the success of the first AutoFOCUS implementation, AutoFOCUS 2 was developed based on more modern technologies. This new version also offered extended functionality which mainly addressed verification activities by supporting model checking and test-case generation. Also the generation of Java and C code from the models was supported. We used this version to conduct the CoCoME case study [BFH+08].

AutoFOCUS 3. In 2007, AutoFOCUS 2 and its SWING-based GUI were considered outdated and old-fashioned, and that it was no longer meeting the demands of nowadays CASE tools. Thus, it was decided to implement the next major evolution AutoFOCUS 3 on the basis of the well-established Eclipse technologies EMF and GEF [BSM+03]. So, the tool AutoFOCUS 3 itself is now widely developed by applying model-based software engineering techniques like meta-modeling and code generation. Although the semantic foundation behind AutoFOCUS 3 is still the FOCUS theory, its scope is beyond being just a demonstrator for formal methods. AutoFOCUS 3 was designed to be an extendable platform that supports systems modeling for different domains. As already stated in Section 2.1, a modeling language (e.g. state charts) does not perform equally well on each given domain. For some kinds of systems, a specific modeling technique is convenient for expressing the system’s functionality, for other ones it is very cumbersome [LKpT04]. The requirement of offering domain-appropriate modeling languages cannot be addressed by a single monolithic CASE tool that provides a fixed set of description techniques. Thus, the modeling workbench AutoFOCUS 3 should explicitly fulfill the requirement of being easily extendable for new language modules as proposed in Section 4.2. This approach is in contrast to informal modeling techniques in which the relationships between different views often remain unclear. Even completely new description techniques should be deeply integrated into the overall language supported by AutoFOCUS. This integration should not only provide consistency checks, but also simulation and code-generation of models containing different description techniques that have been composed with each other.

A building-block tool architecture. The main principle of the architecture of AutoFOCUS was the separation of horizontal and vertical tooling aspects as described in Section 4.1. As illustrated in Figure 9, generic functionalities that are not specific to a certain modeling technique (green boxes) have been factored out from the implementation of the language-specific building-blocks (blue boxes) that contain the functionality needed for only one single modeling technique. Thus, a layer providing very generic modeling functionality has been introduced.
above the Eclipse frameworks. Generic functionality has been implemented to support the easy and quick development of language modules. More specifically, libraries have been built that offer convenient support for developing new meta-models, editing functionality and code-generators that are deeply integrated with each other. Eclipse with its OSGi-based dynamic module system [OSG] and its robust descriptive extension mechanisms turned out to be perfectly suited for the implementation of the vision of an extendable modeling environment. Every language module is implemented as a set of Eclipse plugins. Before starting an engineering project, a composition of modeling languages that are most appropriate for a specific system can be combined. To achieve a deep integration the modeling languages are dependent of each other and use one another’s extension points.

Currently supported modeling techniques. At first, AutoFOCUS aimed at supporting the ‘classical’ description techniques from the previous versions. Thus, in the initial releases, networks of hierarchical logical components could be modeled that interact by using ports to send messages through channels. The correct composition of the components is ensured by the type system. The behavior of the system can be specified using automatons in leaf components. After this stage has been achieved, new views have been introduced to support a model-based deployment of the logical components onto a distributed hardware platform consisting of ECUs that are connected by different bus systems and provide different kinds of actuators and sensors [Sch08]. To support this functionality, a new language module has been developed that provides a graphical editor to model hardware topologies. Furthermore, another language was built to define the deployment mapping between the logical components to the ECUs as well as the mapping between the logical ports and the physical actuators and
5.1 Example: Describing heterogeneous Hardware by Language Modules

A language for describing hardware topologies. Especially for integrating the modeling of hardware topologies containing heterogeneous ECUs, buses, sensors and actuators the extendable architecture has proved to be an effective decision. There are a multitude of different ECUs and bus systems on the market. Conventional tools that allow model-based deployment often rely on specific (rapid prototyping) hardware (e.g., ASCET [ETA]). Thus, the actual hardware used for production and the hardware used during design or prototyping differ. The architecture of AutoFOCUS 3 allowed us to implement a metamodel of the hardware topology that is generic (in one plugin). This generic topology metamodel only knows about ECUs that are connected to buses, sensors and actuators without talking about which specific ECUs are given. The implementation of specific ECUs (e.g., an MPC5554, ARM, Coldfire), buses (e.g., a CAN-bus, Flexray, Ethernet) and peripherals (buttons, LEDs, displays etc.) is performed by extending the metamodel classes of the generic topology plugin. These specific ECUs are provided by the metamodel of separate plugins that contain the language of the concrete hardware components. The specific hardware components are represented directly in the metamodel of the language. The language module depends on the generic hardware topology language component by inheriting from the abstract ECU, bus, actuator or sensor entities of its metamodel. These kind of language modules might be provided as hardware support bundles by vendors of the specific hardware instead of providing ordinary libraries as they do today. This kind of approach allows for a deep integration of the specific hardware with the editors, constraint-checking, simulation and code-generators of a CASE-tool like AutoFOCUS. A similar way of integrating heterogeneous hardware into an integrated development environment is currently pursued by Microsoft’s NET-Micro-Framework [Mic] by so-called Board Support Packages.

Generating code. The big advantage of representing concrete hardware directly in the metamodel of the language is that very detailed and specific information about the hardware components can be provided. This information can be used, for example, in the code generator to be able to produce better (smaller, more efficient etc.) code. For example, the information about the architecture of a processor of an ECU can be used to generate code that directly accesses its IO modules for interacting with sensors and actuators (instead of writing glue code) or that actively uses co-processors like DSPs and specialized timing units to achieve better performance. Nowadays, generated code often fails short on runtime- and memory-performance and forces developers to implement such code manually. The language module that provides the model of a certain piece of hardware also comes up with an associated part of a code generator. This partial code generator implements an interface requested by the generic topology language generator. Consequently, the people that know the hardware best (i.e., the vendors) would be able to provide specific generator parts in order to achieve the best performance of the hardware. The customer would be able to quickly get a running system that already comes up with high runtime- and memory-performance without having to learn every detail of the hardware.
Beyond hardware. The extendable architecture of AutoFOCUS 3 was not only built to allow extending the hardware topology language. Also, on the logical architecture, extendability is a key issue. For example, the automata which were the only way of describing behavior in the previous AutoFOCUS versions are not always the most appropriate modeling technique. If the function that has to be computed by a logical component is just a simple mapping of its input to its output ports, a mapping table is a much more convenient way of describing the situation. To develop control applications, special languages that allow an efficient description of the control algorithms would be much more desirable than automata. Even some imperative language that enables the definition of variable assignments, loops (perhaps only statically bounded ones) and if-statements might be beneficial in several situations. Having the flexibility to tune the language to make it most appropriate for the specific task achieves a less cumbersome and more efficient development. Furthermore, smaller and more comprehensible models can be built that due not fall short a huge encoding gap [Rat09].

5.2 Case Studies and Future Work on AutoFOCUS

AutoFOCUS 3 has already been successfully applied to develop a realistic ACC/PCS system [FFH+08] and a Keyless Entry system [FHP+ar] based on real-world requirements. Furthermore, we also provided support for the generation of verified code [H09].

In the future, AutoFOCUS will be further extended with language components that address the early phases of a project. In addition, several behavior description techniques, such as mode diagrams, an imperative language etc. are planned.
6 Related Work

6.1 Tool Integration Approaches

In the literature, we can already find some approaches for tool integration. In [BHW05], 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 [BHLW07], the authors give a more detailed description of their algorithm for incremental and interactive consistency management. The authors of [KLN05] explain and compare two architectural design patterns which 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. [Mar05] presents the extension of the ETI (Electronic Tool Integration) platform with web service technology. The integrated tools interact with each other by use of web services, which allow to decouple the different tools from each other and which therefore ease integration and maintenance activities. In [KS06], the authors present their rule-based approach MDI (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 which 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 [FGC+06] 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.

Beside these academic approaches there are already some tool developers offering integrated tool support. For the automotive domain, Vector has developed the tool eASEE\(^1\) 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\(^2\) follows a similar direction.

6.2 Language Workbenches

There are many standards from the Object Management Group (OMG) like the Meta Object Facility (MOF) for the definition of metamodels, Object Constraint Language (OCL) [OMG06b] for defining constraints on MOF-based metamodels and the Query/Views/Transformations (QVT) specification [OMG08]. 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

---

\(^1\)http://www.vector-worldwide.com/vc_easee_de.htm,1296.html
\(^2\)http://www.aquintos.info
Modeling Environment (GME) [Dav03], MetaEdit+ [Tol04], Microsoft’s Domain-Specific Language Tools, and the Eclipse Modeling Framework (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 of support for the development workflow. But nevertheless they represent tool architectures that separate the modeling language from the tool infrastructure.

Ptolemy 2 [EJL+03] is a scientific tool based on the idea of combining heterogeneous models of computation. The basic architecture of Ptolemy 2 consists of a kernel and several shells around it. The first shell is a general graph implementation allowing entities to be connected. The actor shell then provides the semantic interpretation of entities and connections by means of different models of computation. These models of computation can be plugged into the tool. Compared to Ptolemy 2, AutoFOCUS 3 follows a different goal. While AutoFOCUS 3 targets the complete development cycle from requirements management to system deployment using a single model of computation (which we believe is in particular suitable to describe the software of embedded systems), Ptolemy aims at combining different models of computation at different levels of the design. AutoFOCUS 3 provides a coherent set of semantically founded description models that leads to a seamless, pervasive engineering process. In contrast, Ptolemy 2 aims at studying the correlation between heterogeneous models of computation defined by computer science during the last decades.
The full promises 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 twenty years 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, and 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 aspect (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 which 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.
References


[ETA] ETAS. Etas ascet website.


[Mod] Model cvs webpage.


[OSG] OSGi. Osgi alliance website.


