Evolution of Ecosystems for Language-Driven Engineering

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Steve Boßelmann

Dortmund

2023

Tag der Disputation:
23.03.2023

Dekan:
Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Dr. h.c. Martin Wirsing

Acknowledgements

My utmost gratitude goes to my supervisor Bernhard Steffen for providing me with invaluable guidance throughout the process of writing my thesis and for granting me the privilege of working at his chair, which was characterized by a collaborative, supportive, and productive atmosphere that fostered creativity and critical thinking. He always had an open ear to innovative ideas and made sure that the team pursues a common goal while each individual, including myself, could progress and develop.

I want to express my sincere appreciation to the other referee of my thesis, Martin Wirsing, as well as to Heinrich Müller for chairing my doctoral committee.

I am also particularly grateful to Tiziana Margaria for her support, especially in my early post-Diplom research, in aiding me making abstract concepts more concrete and applicable to real-world situations and understandable for non-technical audiences, which greatly contributed to the clarity and accessibility of my subsequent work.

Many thanks go to my awesome colleagues who shared the office room with me over the years. At the beginning, it was Christian Wagner helping create a welcome, collaborative and motivating atmosphere in a new environment. And then Stefan Naujokat with whom I had many productive discussions about all the meta-stuff and meta-meta-stuff, which greatly contributed to understanding of the subject matter and the development of new ideas.

I would like to extend my appreciation to all of my other colleagues who provided me with valuable input and feedback. While I am hesitant to list each person individually out of fear of forgetting someone, please know that your contributions greatly helped to shape my work and enhance its quality.

I would also like to express my deep gratitude to my mother, who gave me life and started me on this journey, and give a big shout-out to my sister. Both of you have been a constant source of support and encouragement and I am truly fortunate to have such wonderful family members in my life.

Finally, I would like to express my deepest gratitude from the bottom of my heart to my wife and kids for their unwavering support and understanding during this challenging journey. I am so grateful for their patience and encouragement, and for granting me the necessary leeway to work on the thesis, even when it meant putting other activities and plans on hold. I could not have completed this without their love and support. Thank you for being my biggest cheerleaders and for always believing in me!

Abstract

Language-Driven Engineering (LDE) is a means to model-driven software development by creating Integrated Modeling Environments (IMEs) with Domain/Purpose-Specific Languages (PSLs), each tailored towards a specific aspect of the respective system to be modeled, thereby taking the specific needs of developers and other stakeholders into account. Combined with the powerful potential of full code generation, these IMEs can generate complete executable software applications from descriptive models. As these products themselves may again be IMEs, this approach leads to LDE Ecosystems of modeling environments with meta-level dependencies.

This thesis describes new challenges emerging from changes that affect single components, multiple parts or even the whole LDE ecosystem. From a top-down perspective, this thesis discusses the necessary support by language definition technology to ensure that corresponding IMEs can be validated, generated and tested on demand. From a bottom-up perspective, the formulation of change requests, their upwards propagation and generalization is presented. Finally, the imposed cross-project knowledge sharing and transfer is motivated, fostering interdisciplinary teamwork and cooperation.

Based on multifaceted contributions to full-blown projects on different meta-levels of an exemplary LDE ecosystem, this thesis presents specific challenges in creating and continuously evolving LDE ecosystems and deduces a concept of Path-Up/Tree-Down (PUTD) effects to systematically address various dynamics and appropriate actions to manage both product-level requests that propagate upwards in the meta-level hierarchy as well as the downward propagation of changes to ensure product quality and adequate migration of modeled artifacts along the dependency paths.

Finally, the effect of language-driven modeling on the increasingly blurred line between building and using software applications is illustrated to emphasize that the distinction between programming and modeling becomes a mere matter of perspective.

Attached Publications

Listed below are the publications attached to this dissertation. They were written and co-published with other authors. My specific contribution to each of these papers is outlined in section 1.1. Throughout this document, the provided labels will be utilized for citation, e.g. [AP1], to enable convenient identification of an attached publication.

AP1 Barry D. Floyd, Steve Boßelmann:
ITSy - Simplicity Research in Information and Communication Technology
In: *Computer*, vol. 46, no. 11, IEEE, 2013
DOI: 10.1109/MC.2013.332

AP2 Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, Bernhard Steffen:
DIME: A Programming-Less Modeling Environment for Web Applications
In: *Leveraging Applications of Formal Methods, Verification and Validation.* Lecture Notes in Computer Science (LNCS), vol 9953, Springer, Cham, 2016
DOI: 10.1007/978-3-319-47169-3_60

AP3 Steve Boßelmann, Dennis Kühn, Tiziana Margaria:
A Fully Model-Based Approach to the Design of the SEcube™ Community Web App
In: *12th International Conference on Design & Technology of Integrated Systems in Nanoscale Era* (DTIS), IEEE, 2017
DOI: 10.1109/DTIS.2017.7930159

AP4 Dominic Wirkner, Steve Boßelmann:
Towards Reuse on the Meta-Level
In: *Electronic Communication of the European Association of Software Science and Technology*, vol. 74, ECEASST, 2018
DOI: 10.14279/tuj.eceasst.74.1047

AP5 Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, Bernhard Steffen:
Towards Continuous Quality Control in the Context of Language-Driven Engineering
In: *Leveraging Applications of Formal Methods, Verification and Validation.* Lecture Notes in Computer Science (LNCS), vol. 13702, Springer, Cham, 2022
DOI: 10.1007/978-3-031-19756-7_22

AP6 Steve Boßelmann, Stefan Naujokat, Bernhard Steffen:
On the Difficulty of Drawing the Line
In: *Leveraging Applications of Formal Methods, Verification and Validation.* Lecture Notes in Computer Science (LNCS), vol. 11244, Springer, Cham, 2018
DOI: 10.1007/978-3-030-03418-4_20

Contents

1 Introduction

The concepts of Domain-Specific Languages (DSLs) [16, 17] as well as the steps to create a DSL have now been studied for decades, even before the term was introduced in the 1990s [18, 19]. To support the main tasks of creating DSLs, so-called language workbenches [19] have emerged and continuously evolved. With them, users can define the language syntax in an appropriate grammar language from which the language workbench generates the required utility tools originally known from classical compilers (Lexer, Parser, Linker, etc.) to parse given input and create in-memory models for further processing. This way, at least from a technical perspective, several cumbersome and repetitive tasks have been continuously simplified to reduce the amount of work and effort to build DSLs [19].

In this sense, the SCCE Meta-Tooling Suite Cinco [20, 21], a simple yet powerful workbench for graphical languages, has been developed at the Chair for Programming Systems at Technical University Dortmund. Besides the outstanding achievements regarding simplicity of language definition and implementation, the Cinco project has also evolved the aspect of code generation towards a powerful approach for full application generation [22]. Finally, by combining dedicated DSLs with the potential of full code generation, Cinco matches the characteristics of an Integrated Modeling Environment (IME) that can produce executable software applications completely from models. Each generated Cinco-product is a graphical modeling environment on its own which, when equipped with an appropriate code generator, can again be evolved into a complete IME.

Following this meta-modeling approach, Cinco has been used to develop Dime [AP2], the first IME that generates full enterprise web applications completely from descriptive models. The experiences gathered while developing and using Cinco, Dime and similar IMEs led to the advanced concept of Language-Driven Engineering (LDE) [23] promoting Purpose-Specific Languages (PSLs) [24] tailored towards the specific needs of developers and other stakeholders involved. Finally, the creation of Cinco, and Dime in particular, has proven that the decades-old vision of building sophisticated productive systems with a set of DSLs, each tailored towards a specific aspect of the system, has finally turned into reality. Moreover, the fact that the main modeling steps are quite easy to handle, particularly in comparison to mastering the technology stack underlying modern web applications, makes Dime accessible even for non-programmers.

Since its early days, Dime has been used to build Equinocs, the new conference management system of Springer Nature and after an initial proof-of-concept phase, Equinocs was finally rolled out into production to serve thousands of users. This success not only proves that the Dime-based approach to modeling and generating production-ready applications is applicable in real-world scenarios, but has also shown that the LDE approach could potentially make a significant difference to the development, deployment and operation of software systems. However, there are challenges to be considered emerging from the fact that Equinocs is a Dime product and Dime is a Cinco product, i.e. they are all part of the same ecosystem and

cannot be seen in isolation. In general, the approach of using IMEs like Cinco or Dime to model and generate software products, which themselves may again be IMEs, almost naturally leads to LDE ecosystems of modeling environments with meta-level dependencies that can be represented as hierarchical tree structures (cf. Fig. 1.1).

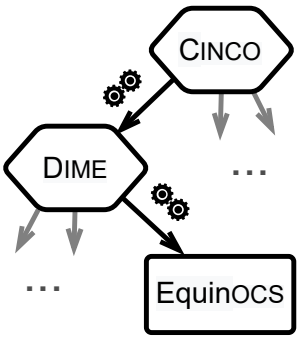

Figure 1.1: Meta-model dependency path in an LDE ecosystem

From the dependency paths between modeling environments and their corresponding meta-modeling environments in LDE ecosystems so-far unexplored dynamics emerged across those development paths that cross one or more meta levels. Consequently, new challenges emerged not only from the composition of multiple interlinked languages with regards to modularity, consistency and reusability but also from a broader perspective on evolution which not only takes changes of a single language into account but in particular considers changes that affect multiple parts or even the whole ecosystem. This means, from a top-down perspective the LDE approach must be supported by language definition technology to ensure that corresponding IMEs can be validated, generated and tested on demand. In turn, from a bottom-up perspective, the upwards propagation and generalization of new requirements and change requests must be supported to foster the evolution of the ecosystem on the appropriate levels in a structured manner. A main challenge in this context is the development of technological solutions supporting such ecosystem dynamics and facilitating cross-project knowledge sharing and transfer, while enabling participants to focus on their particular projects instead of becoming experts of the whole ecosystem.

Based on these observations, this dissertation deduces a concept of Path-Up/Tree-Down (PUTD) effects, systematically addresses the various dynamics in LDE ecosystems and proposes appropriate actions to manage both product-level requests that propagate upwards in the meta-level hierarchy as well as the downward propagation of changes to ensure product quality and adequate migration of modeled artifacts along the dependency paths. Based on multifaceted contributions to Cinco, Dime and EquinOCS, i.e. projects on different meta-levels, this dissertation presents and discusses the particular challenges in creating and continuously evolving LDE ecosystems and effectively managing change in this context.

Finally, the effect of language-driven modeling on the increasingly blurred line between building and using software applications is illustrated to emphasize that the distinction between programming and modeling becomes a mere matter of perspective [AP6]. As a result, the strict separation of meta levels may be weakened or completely overcome through language technology and tool support, without losing track of *who* uses *which* language to achieve certain goals.

1.1 My Contribution

From the very start, the intrinsic motivation of my work was grounded in the conviction that simplicity in application and systems engineering is a key driver for successful development projects and high-quality results. For this reason, my early work was focused on promoting a better understanding of the nature and meaning of simplicity in scientific research and technology. In this context, my work as a research fellow in the EU Seventh Framework Programme (FP7) project IT Simply Works (ITSy) [25, 26] on establishing a culture of simplicity in IT resulted in the following journal publication:

Attached Publication [AP1]

Barry D. Floyd, Steve Boßelmann

ITSy - Simplicity Research in Information and Communication Technology

in *Computer*, vol. 46, no. 11, IEEE 2013

My contribution to this work was (1) a systematic review of IT literature for articles addressing the concept of simplicity and (2) the conduction and evaluation of interviews with discussion groups, focus groups, and individuals to collect insights and identify current thinking about simplicity and its applications among researchers and practitioners in computer science, software engineering, and the IT industry.

The investigations led to the conclusion that although simplicity can be regarded as a key driver for various applications it is only vaguely understood and receives little formal attention in the computer science community. Another finding of my studies in this field is that simplicity cannot be regarded as a universal feature as it depends on the mindset, experience and knowledge of the respective user. This result has since influenced all of my further work on tailored solutions in terms of developed software products in general as well as the design of languages for modeling environments in particular. In fact, these findings on the notion of simplicity directly lay the foundations for the concept of purpose-specific languages, which play a vital role for the conception of LDE ecosystems as presented in this dissertation.

Driven by the desire to build a simple yet powerful workbench for graphical languages, the SCCE Meta-Tooling Suite Cinco has been developed at the Chair for Programming Systems at Technical University Dortmund and continuously evolved to a sophisticated development tool for integrated modeling environments based on interconnected graph models. Since the very beginning, and over the subsequent years, I contributed to the Cinco project as part of the core developer team by developing and integrating new languages and core functions, as well as by establishing a course for teaching its application and underlying principles at Technical University Dortmund.

The Cinco workbench has also been used to create the integrated modeling environment Dime for the development of web applications from various interconnected graphical models of data, control flow and the user interface. Dime was initially introduced by the following publication:

Attached Publication [AP2]

Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, Bernhard Steffen

DIME: A Programming-Less Modeling Environment for Web Applications

in *Lecture Notes in Computer Science (LNCS)*, vol. 9953, Springer 2016

My contribution to this article has been the demonstration and explanation of the design and actual intent of DIME's graphical DSLs, as well as their horizontal and hierarchical relationships. Furthermore, I presented the overall ease of use, in particular by describing the built-in user-level guidance by means of the various available views.

I have contributed to the DIME project since the very beginning, and over the subsequent years, as part of the core developer team by designing and integrating new modeling languages and developing core features in terms of model editing, UI components and code generation. Having started off as an academic project, DIME is open source and now widely adopted in practice by both academia and industry. So-called "DIME Days" events have been associated with the ISoLA conferences since 2020. And recently, I have been teaching its application and underlying principles in an innovative learn-by-doing course of study on Immersive Software Engineering (ISE) at University of Limerick. Meanwhile, I am leading the DIME development team at the Chair for Programming Systems. In the position of lead software architect, I am responsible for managing the project in an agile manner to plan and shape the ongoing development, comprising the conceptualization, design and implementation of textual and graphical modeling languages, user interface components and code generators.

Besides working *on* DIME, I have also *used* DIME in externally funded projects on developing web applications for industry partners. An early proof of concept is the development of the SEcube™ Community Web App as presented in the following publication:

Attached Publication [AP3]

Steve Boßelmann, Dennis Kühn, Tiziana Margaria

A Fully Model-Based Approach to the Design of the SEcube™ Community Web App

in *12th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, IEEE 2017

My contribution to this article comprises the presentation of the basic modeling concepts in DIME as well as its application in terms of modeling the web application, with focus on describing the relationship of models and final outcome in the generated product.

A much larger-scale project, and also a full-scale proof of concept regarding the generation of production-ready products with DIME, is the development of EquinOCS, the new conference management system of *Springer Nature*, which currently is used to host more than 150 conferences per year while the user base is continuously growing. Leading the EquinOCS

development team since 2019, I am responsible for the conceptualization and implementation of the full-stack web application with Dime. Additionally, as a consultant, I am also responsible for the requirements engineering in meetings with the project management and other stakeholders from Springer Nature.

Although EquinOCS is a closed-source project, in order to explain its place in the LDE ecosystem this dissertation comprises the first publication of project insights, still in accordance with the nondisclosure obligation.

The creation of EquinOCS with Dime and Dime with Cinco is a perfect example for a development path with dependencies between different modeling environments crossing meta-level boundaries. In this constellation, change requests are directed upwards the meta-level hierarchy and typically the main products of a modeling environment become the key drivers for its further development and the continuous evolution of meta-level features. In this manner, while using Cinco for building various modeling languages into Dime a need for an efficient reuse concept regarding common language features has been recognized and presented in the following publication:

Attached Publication [AP4]

Dominic Wirkner, Steve Boßelmann

Towards Reuse on the Meta-Level

in *Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, vol. 74, 2018

My contribution to this article has been the explanation of recurring constructs and patterns in Dime models, how their creation and update are managed in terms of manually written code and the formulation of required language features to support these patterns in terms of language features to be implemented on the meta level.

The requirements presented in this publication on meta-level reuse have been adopted and built into Cinco over the following years in terms of language modularity achieved via the realization of an inheritance concept. As the eventual release of this feature has meant major change on the meta-level, these changes had to be adopted from all Cinco-based projects, which demonstrated first hand how change requests propagate upwards in the meta-level hierarchy while the resulting need for adaption and migration of existing models affects the whole ecosystem rooting in Cinco.

This dissertation presents and generalizes the aforementioned effect as part of the PUTD dynamics in LDE ecosystems as well as the challenges emerging from these dynamics for the effective and efficient management of change and evolution propagating through the product landscape. One major aspect in this context is continuous quality control, as presented in the following publication that illustrates how to ensure the adequate migration of products generated from corresponding modeled artifacts when meta-level changes propagate down the meta-level hierarchy in an LDE ecosystem. In this context, it presents a concept of design for testability envisaged to support automatic validation of runtime properties using learning-based testing techniques.

Attached Publication [AP5]

Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, Bernhard Steffen

Towards Continuous Quality Control in the Context of Language-Driven Engineering

in *Lecture Notes in Computer Science (LNCS)*, vol. 13702, Springer 2022

My contribution to this article has been the assessment of scalability based on path-up/tree-down dynamics in LDE ecosystems and, in particular, the formulation of the need for quality control that considers aspects crossing meta-level boundaries as potential unintended effects of change might only be recognized in products multiple levels deeper in the meta-level hierarchy.

Another finding is that in LDE ecosystems with the progression of automated system integration the boundaries between building and using software applications become increasingly blurred. While the LDE approach generally supports this development, it leads to a slightly different perception of DSLs and IMEs becoming integral parts of the whole system life cycle from the viewpoint of both system developers and system users. Thereby, switching the abstraction layer does not necessarily mean switching from system level to development level, as presented in the following publication:

Attached Publication [AP6]

Steve Boßelmann, Stefan Naujokat, Bernhard Steffen

On the Difficulty of Drawing the Line

in *Lecture Notes in Computer Science (LNCS)*, vol. 11244, Springer 2018

My contribution to this article has been the presentation of a concrete example from the development of EquinOCS about how the need for languages or language features on different meta-levels may change in the course of a project and, deduced from this finding, how this dynamic may be supported by the LDE ecosystem in terms of enabling dynamic transfer of control over a modeling language.

One of the conclusions of this publication is that in the context of LDE ecosystems the distinction between programming and modeling becomes a mere matter of perspective. Hence, a clear understanding and appropriate tool support for LDE are mandatory requirements for introducing and maintaining simplicity for the actual users of this complex development environment. Taking up this idea, this dissertation presents some requirements to be met and discusses the main challenges to be addressed in this context.

2 Language-Driven Engineering (LDE)

The following sections explain the foundations needed to introduce the concept of LDE and thereby lay the groundwork for the definition of LDE ecosystems.

In software engineering, the different mindsets of the involved stakeholders, i.e. the differences in their specific way of thinking, has been recognized as a main reason for the so-called *semantic gap* [27], which denotes the difference in meaning within different representations and conceptions [28, 29]. Language-Driven Engineering (LDE) is a paradigm envisaged to bridge this semantic gap by addressing the different mindsets of stakeholders involved in application and system development by providing them with dedicated modeling languages tailored towards their specific needs regarding both mindset and capabilities [23]. In particular, the LDE approach addresses the domain experts and application experts without programming skills and aims at tightly integrating them into the model-driven development process by means of providing them with dedicated modeling languages tailored to their specific mindsets [24] in order to improve the personal modeling experience (cf. [30]). It is the result of the systematic further development of eXtreme Model-Driven Design (XMDD) [31] and, conceptually, follows the One Thing Approach (OTA) [32] that propagates a global, consistent model at the core of the development.

The LDE approach promotes DSLs to first class citizens of the development process and refines their objectives and scope. In the general context of Model-Driven Engineering (MDE), the DSL notion is a well-established concept, yet only vaguely defined. Hence, the term is very loaded. While some see DSLs as languages focusing specific application domains that have nothing to do with programming in the first place [33], others propagate a strong programming-related perspective when defining them as ”small languages, focused on a particular aspect of a software system” [16]. Definitions like the latter are typically referred to when considering programming-related specification languages like Structured Query Language (SQL) or Hypertext Markup Language (HTML) as being good examples of DSLs. In contrast, several (graphical) DSLs used by people without programming skills became very successful in certain application domains, like MatLab/Simulink [34], ladder diagrams [35] and Modelica [36]. Actually, DSLs can be used in a wide range of non-programming contexts to improve communication, productivity and problem-solving within specific domains. By providing a more natural and intuitive language for expressing ideas within a particular domain, DSLs can make it easier for domain experts to collaborate with each other.

Furthermore, there is a distinction between *external* and *internal* DSLs. While external DSLs are standalone languages that come with their own parser, an internal DSL is rather a special form of Application Programming Interface (API) in a programming language that makes it easier to program with because it is designed in a way that using it feels like writing in a dedicated language.

Modeling languages, in particular DSLs, are used to create models that represent particular

things of interest. This approach can also be applied to the design of modeling languages themselves. This means modeling languages are used to create models that describe other modeling languages. Consequently, as illustrated by Figure 2.1, this introduces an indirect relation between a particular model and the model of its language. When the latter is called *meta-model* the prefix *meta* refers to this exact relation.

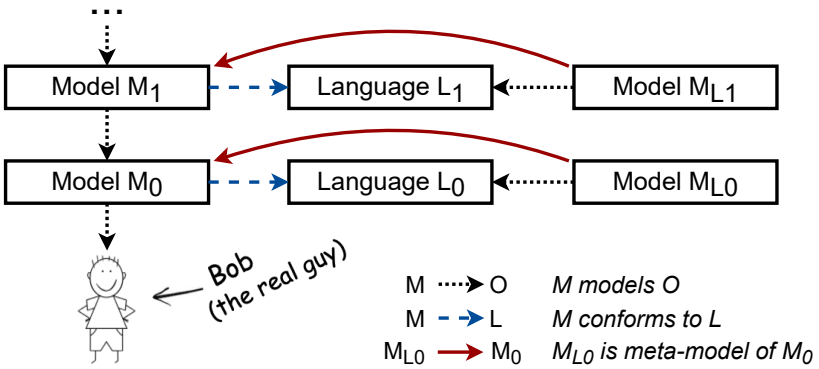

Figure 2.1: Relations between models and meta-models

As typically such a meta-model prescribes the language, i.e. changing the meta-model means changing the language, both terms become interchangeable and may be considered synonymous. Hence, in the following, the phrases *"changing a model's language"* and *"changing the meta-model"* are used synonymously.

While a classical DSL focuses a particular domain, the LDE approach takes different dimensions of specialization into account. It facilitates dedicated modeling languages that are tailored towards specific aspects of the target domain, the intent and purpose of each particular language as well as the specific capabilities and mindsets of involved stakeholder groups or even individuals.

Domain-Specific aspects comprise the prevailing terminology as well as constraints and regulations from a particular field or the identified domain. Think of technical or special terms used in a particular field as well as a set of ground rules associated with the respective surroundings. In this context, sophisticated guidance by means of tool support means taking all these aspects into account and validating the generated models against identified boundary conditions.

Purpose-Specific aspects address the goal and task orientation of a particular modeling approach. Providing an appropriate tool set to choose from may very well depend on what concrete goal and task has to be supported. The better the tool set is adapted to specific tasks the better the overall guidance and support. General-purpose tools might be universally usable but less useful for a range of specific tasks.

Mindset-Specific aspects address the knowledge, skills and expertise of the respective individual. Different stakeholders involved may have different views, opinions and beliefs on a range of issues. Addressing these different views and perspectives is key for an effective tool to provide guidance, support knowledge sharing and increase the overall user experience. The goal is to enable in particular non-technical stakeholders to specify an intended solution in their established mindset.

To emphasize these unique characteristics, the languages in context of LDE are called PSLs [24].

Examples of templates for the design of such PSLs can be found in various application domains in terms of dedicated (graphical) languages, like network layouts, workflow graphs, or piping and instrumentation diagrams but also business models and other strategic notions, as well as spreadsheets or data flow graphs for data analytics. In general, illustrative charts and diagrams used by domain experts for the documentation of specific relationships or factual connections are a good starting point for creating efficient PSLs aligned with the mindsets of the respective stakeholders.

The IMEs that support the LDE approach by providing involved stakeholders with the means to adapt the adequate PSLs required to support their mindsets are referred to as *Mindset-Supporting Integrated Development Environments* (mIDEs) [24]. By combining multiple PSLs that support the mindset of specific stakeholders into an mIDE the latter itself is tailored towards stakeholder needs to increase the personal modeling experience [30]. Additionally, available tools and features of the mIDE can be tailored to the respective purpose and thus provide maximum guidance in support of effective and safe modeling, e.g. by integrating appropriate analysis and verification tools to minimize accidental misuse. In this context, the powerful approach of code generation applied to the development of mIDEs allows for building specific mIDEs even for small stakeholder groups or individuals. However, the fundamental prerequisites for this are simplicity and efficiency in PSL design along with the support of reuse concepts for languages and language constructs. The orchestration and aggregation of corresponding artifacts on different meta-levels is a major challenge.

Note that in the course of this work mIDEs are considered specialized types of IMEs providing access and usage of mindset-supporting PSLs. And while the discussion and findings of this work are formulated for the more general concepts of IMEs and language-driven ecosystems, they are very well applicable in specific contexts where mIDEs with PSLs are involved.

2.1 Code Generation

The goal of MDE is to transform abstract representations, i.e. models, into executable code that implements an application's structure as well as its behavior. Model-Driven Software Development (MDSD), on the other hand, can be considered a more specific form of MDE that focuses on the use of models to automate the software development process. This typically involves the use of code generators, which automatically generate source code from higher-level specifications or models. In model-driven approaches, it closes the gap between abstract representations and concrete software systems and thus can be considered equivalent to what compilers are to high-level languages [37].

This is done for a variety of purposes, such as to reduce the amount of manual coding that needs to be done, to improve the efficiency of the development process, or to generate code that is optimized for a specific target platform or architecture. While code generation can save time and improve consistency, it can also lead to a loss of control over the code and can result in less flexible and less maintainable code. However, there exist best practices that can be followed to prevent this:

- **Define clear and concise specifications** that capture the requirements and design of the system. This will ensure that the generated code meets the desired requirements and that it is possible to trace the generated code back to the specifications.
- **Use templates or models** to specify the code generation process. This helps ensure that the generated code is consistent and follows a predefined set of rules. It also provides a higher level of control over the generated code, as the templates or models can be modified as needed.

- **Validate the generated code** to ensure that it meets the requirements and that it is free of errors. This can be done through manual code review or automated testing.
- **Document the code generation process** to keep a detailed record including the specifications, templates, and models used. This can help ensure that the generated code is easily maintainable and that it is possible to understand the intent behind the generated code.

In general, the models from which code generators automatically create source code do not necessarily have to reflect application structure or behavior directly. As first and foremost, DSLs are used to express crucial aspects of interest in the language of a particular domain, the modelers focus on *what* to achieve instead of on *how* this is implemented in a particular software application. This level of abstraction maintains the possibility of deriving various kinds of software applications from the same models. And these applications could even behave very differently yet supporting the same tasks and objectives.

The development of code generators is supported by so-called template engines which let developers efficiently define text templates to express static text with dynamic parts that are rendered based on provided data. Template engines typically use placeholders or special syntax to indicate where data should be inserted, and may also include control structures such as loops and conditionals for more complex logic. The text parts of such a template then confirms to the syntax of the respective programming language of choice, like C/C++ or Java, so that the outcome are files that contain class definitions with fields and methods, just as produced in manual programming. There are several popular template engines available for different programming languages. For example, in the Python [38] programming language, the Jinja2 [39] template engine is widely used, while in the JavaScript [40] ecosystem, Handlebars [41] and Mustache [42] are popular choices. In the Java [43] world, very common template engines are Freemarker [44] and Velocity [45]. For the development of code generators by using such template engines, there also exists model-driven approaches that involve the use of models to specify the code to be generated as well as the code generation process itself [37]. This can improve the quality and maintainability of code generators by providing a higher level of abstraction and a clear separation of concerns.

Partial Code Generation

One of the early approaches to MDSD became known as *partial* code generation, which means creating code skeletons, also known as *stubs*, that must be manually completed by hand-written code afterwards. Hence, this generation process can be considered a one-time event at an adequate moment during the development, as it cannot be repeated in later stages of the project because regenerating the code would overwrite or even erase all manual progress made since the last generation.

To tackle this drawback of partial code generation, one can demarcate *protected areas* [46] or *protected regions* [47] in the generated code by using specific comments to be recognized by the code generator during regeneration steps, telling it to modify or overwrite only those specific regions. In turn, any manual modifications are restricted to the areas outside of these protected regions. However, these constraints may not always be technically enforced by appropriate tooling, so that compliance relies on the programmers' discretion. An alternative yet similar approach aims at completely separating manually written code from generated code [48]. In Object-Oriented Programming (OOP) [49], this can be achieved by generating interfaces or classes to be extended by hand, or vice versa [16].

Full Code Generation

The approach of *full* code generation overcomes the described issues by producing the entire code needed automatically. This implies that the code generated is fully operational and does not require manual adjustments as in the partial code generation scenario. This saves time, reduces the chance of errors and thus increases the overall code quality. However, it also has some drawbacks, such as making it more difficult to understand and maintain the generated code or the generation process itself, especially if the generator is not well-documented or the generated code is not well-organized.

In terms of code quality, the existing approaches and best practices that can be used to write readable and maintainable code are also applicable on automatically generating code. As code generation basically imitates and automates the work of programmers, apparently, when generating code there is no reason for neglecting the virtues that have proven to be effective for manually writing good code. In fact, the opposite is the case because the generated code is the product of the code generator and thus the central artifact for its analysis, validation, debugging and continuous improvement. Generating good code will make it much easier to understand the generation process and thus to maintain and improve the code generator.

The generation of full code, by means of not leaving gaps to be filled manually, does not per se prohibit manual changes to the generated code. As code generation has an impact on the flexibility of the produced code, in order to adapt to new requirements or changes in the project, the project's development approach may foresee the customization of the generated code by means of manually adding custom logic. As a consequence, the preservation of consistency introduces the need for so-called *round-trip engineering* to synchronize changes in code with changes in models and vice versa. However, in particular the inference of changes in models from changes in code is challenging, especially when the code and the models are large and complex. Although for specific, limited purposes there exist tools that support this round-trip engineering process by means of automatically synchronizing code with models, in general, it is not always possible to accurately infer changes in the models from changes in the code for several reasons:

- **Ambiguity:** The code may not provide enough information to accurately infer a definite mapping on adequate changes in the models. This can occur due to different degrees of expressiveness or the lack of a bijective function between model components and code constructs. Whenever there exist multiple suitable solutions, the inference algorithm cannot select the appropriate one automatically.
- **Complexity:** The code and the models may be large and complex, making it difficult to accurately infer changes in the models from changes in the code. This can be especially challenging in the context of legacy code, where the code and the models may be poorly documented and understood.
- **Inconsistency:** The code or the models may be inconsistent, making it difficult or impossible to accurately infer changes in the models from changes in the code. This can occur when a previous synchronization run produced inconsistent output or the programmers did not follow certain rules and constraints for writing code that can effectively be processed.
- **Limited tool support:** The tools used for modeling and code generation may not provide adequate support for automatically inferring changes in the models from changes in the code.

In practice, development projects following this approach are often reaching a point where the customized code eventually becomes inconsistent with the models. Unfortunately, in

these cases, they see themselves forced to manually change the models until they fit the handwritten code, just to restore an adequate level of consistency. However, the best way to overcome this round-tripping issue is a full-code generation policy: always generate the full codebase and never touch the generated artifacts. This has been formulated as one of the core strategies for full code generation in the Cinco project [21]. By prohibiting any modifications to the generated code, any future changes must be applied on the model level, followed by a regeneration process that overwrites all of the previously generated code on purpose.

In turn, this means any customization, like custom code or custom components in general, needs to be supported on model level and modeling environments need to support this by means of custom model components that can be integrated in the models in a service oriented fashion. The code associated with these custom components would then be injected into the generated artifacts and placed at the foreseen location by the code generator. This way, full-code generation provides a more straightforward and complete way to keeping code and models in sync.

The actual code generation process might be preceded by a sequence of model-to-model transformations, for example, as a bootstrapping approach by first creating a high-level, domain-specific model of the desired software. This model can then be transformed or mapped into a lower-level, implementation-specific model, from which finally the code can be generated. Additionally, by using a series of transformations, the model can be mapped to multiple different implementation languages or frameworks, allowing for the generation of code in a variety of environments.

2.2 Application Generation

While the result of the classical code generation process are code files in the first place, the overall goal of LDE is nothing less than the full generation of executable applications. Hence, the term *application generation* is more appropriate. This goes even further than *full code generation* [47], which refers to the process of automatically generating all of the source code, i.e., all of the classes, methods and functions that make up the software application.

Application generation goes further than classical code generation as it also includes the next development steps required to thoroughly prepare the build of the application from generated source code. Hence, the application generation process is much more complex and challenging.

Typically, even full code generators do not directly produce every single line of code of a target application. Instead, as code generation basically imitates and automates the work of software developers, it mostly tends to follow similar software development approaches when it comes down to using frameworks or integrating existing third-party libraries. Furthermore, modern applications - in particular web applications - are not only programmed by means of classical source code but also require various additional files to operate, for example:

- **Configuration files**, like the so-called *application configuration resource files* for Java-EE-based web applications.
- **Descriptor files**, like the so-called *bean archive descriptor* for the CDI dependency injection framework included in Java EE.
- **Binary files**, like icons or images.

Besides configuration files that are needed for *running* the application, there are configuration files needed for *building* the application from the resources in its development project. These files contain properties to configure so-called *build management tools*. As an example, the

build management tool Maven [50] requires a so-called *project object model (POM)* that defines the project structure as well as configuration for the actual build process, comprising the declaration of project properties and dependencies.

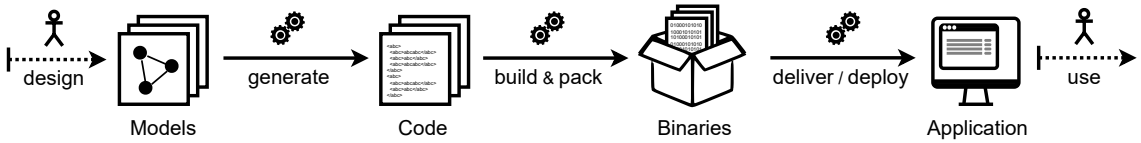

Figure 2.2: Model-driven application generation

Finally, after the build phase, the deployment of the application might need additional configuration and definition files comprising Infrastructure as Code (IaC) [51] needed for virtual runtime environments, like the container virtualization solutions Docker [52] or Kubernetes [53, 54].

Altogether, the generation of full applications makes use of so-called *project archetypes*, i.e. templates that represent the folder structure of a project to be filled and enriched with generated resources. The latter are either pre-defined binary files or text files whose content is automatically created by a generator based on the consumed models.

Continuous Integration and Delivery

Over the last years, the manual tasks of building and deploying software applications became increasingly automated by using Continuous Integration/Deployment (CI/CD) pipelines that provision complete testing, staging and production environments. CI/CD is a software development practice that involves regularly integrating code changes into a shared repository and then automatically building, testing and deploying the application. This allows development teams to detect and fix integration bugs early, and to quickly and easily release updates and new features to software products. This way, the overall cycle time for delivering new versions of software systems is reduced while risks become efficiently manageable because deployments, e.g. for integration testing, can be done on a regular basis [55]. Some improvements that CI/CD pipelines can introduce into the development process include:

- **Simplified integration:** By breaking down the whole process into small and manageable tasks, CI/CD pipelines can help to manage the complexity of the integration process and make it easier to integrate different tools and systems. This can also help to detect and fix integration bugs early.
- **Improved quality:** By automating the process of running tests, CI/CD pipelines can help to ensure that tests are run regularly and that they are comprehensive and reliable. Developers can catch potential bugs early in the development process, before they make their way into production. This reduces the number of bugs that are released to customers and improves the overall quality and reliability of the software system.
- **Improved speed:** By automating the deployment process, developers can ensure that changes are deployed quickly and safely. This reduces the risk of human error in the deployment process. It can also improve the perceived quality of the product by reducing the time it takes for new features to reach customers.
- **Improved scalability:** CI/CD pipelines can be designed to handle a large number of code changes and deployments, and to scale to accommodate a growing team and user base.

- **Improved security:** By integrating automated security testing into the pipeline, developers can catch vulnerabilities early in the development process, before they make their way into production. This can reduce the risk of security breaches and improve the overall security of the product. Additionally, CI/CD pipelines provide integrated solutions to protect sensitive data from unauthorized access, e.g. by encrypting the information during transit and storage.
- **Improved consistency:** By automating the process of deploying software, CI/CD pipelines can help to ensure that the same software is deployed consistently across different environments.
- **Manageable complexity:** By breaking down the process into small and manageable tasks, CI/CD pipelines can help to effectively manage the complexity of the integration process in a repeatable and reliable manner.

All these benefits make automation via CI/CD pipelines particularly useful in situations where full application generation is desired and thus the generate-build-deploy pipeline depicted in Figure 2.2 needs to be executed on source models repeatedly.

Another aspect that plays a critical role in enabling and improving continuous delivery is DevOps [56]. While CI/CD provides a powerful framework for automating and streamlining software delivery, to truly unlock its benefits, organizations introducing DevOps started to embrace a more comprehensive approach to software development. DevOps is a cultural and organizational approach to software development that emphasizes collaboration, communication, and integration between development and operations teams [57]. In addition to cultural changes, DevOps involves adopting a set of technical practices such as IaC, automated testing and continuous monitoring.

Although there exist first promising approaches to the graphical modeling of CI/CD pipelines [58, 59], both CI/CD and DevOps typically involve the use of configuration files to define, automate and standardize various aspects of the software development and delivery process. These include configuration files for the build, deploy and test stages as well as settings and parameters for managing infrastructure resources, such as servers, databases and networking components. However, if full application generation including the automation of the complete process from source models to a deployed and running application is desired, then handling these configuration files becomes a crucial aspect in the design of an appropriate code/application generator.

2.3 Integrated Modeling Environments (IMEs)

An *Integrated Modeling Environment* (IME) comprises everything that is needed to design models and generate full applications from it. Its users, i.e. the modelers, are provided with editors for various model types addressing both textual as well as graphical models. The models' syntactical correctness is enforced via constraints implemented by these editors whereas semantic checks are performed by means of validation routines applied to the models themselves as well as to more complex structures resulting from mutually linked models. Additionally, various views may provide an overview and support the modelers in accessing the models and their properties as well as the model validation results.

To support full application generation from models, an appropriate generator is an imperative part of a complete IME. A generator may consist of multiple subunits targeting the generation of specific types of artifacts, comprising the application's project structure as well as all necessary code, configuration and descriptor files in different languages. These configuration

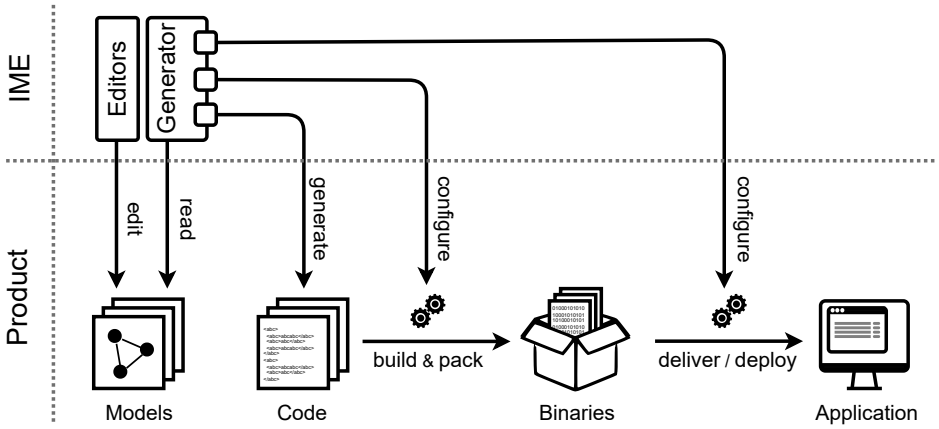

Figure 2.3: Code generation and CI/CD pipeline configuration

and descriptor files may as well comprise the specifications needed to configure the subsequent steps of the deployment pipeline, as illustrated by Figure 2.3 which shows the *generate* step of Figure 2.2 in more detail.

Finally, as an IME may very well support different generation targets (dektop / web / mobile applications) it may as well provide multiple application generators consuming the same model types but producing different outputs.

The editors and generators are inseparable parts of an IME that refer to syntax and semantics of its modeling languages, respectively. Hence, the evolution of an IME may affect both its integrated editors and generators, just like changes of a modeling language may affect both its syntax and semantics.

3 LDE Ecosystems

This section introduces the Cinco language workbench as the root of an unfolding LDE ecosystem that currently spans three meta-levels. In this ecosystem the particular development path along the tools Cinco → Dime → EquinOCS is described and then exemplarily used in the next chapters to present PUTD dynamics observed during their development.

3.1 CINCO *(built with EMF)*

The Cinco SCCE Meta-Tooling Framework [20–22] is a language workbench tailored for the design of graphical languages based on graph models with nodes and edges. For the definition of these languages Cinco provides two powerful modeling languages. In the Meta Graph Language (MGL) the available types of graph models, nodes and edges as well as associated attributes can be defined. The Meta Style Language (MSL) is used to define the visualization of nodes and edges by means of different shapes with customizable appearances. From the models defined with MGL and MSL, Cinco can generate a tailored graphical modeling environment, generally referred to as Cinco-product. To further customize the Cinco-product to be generated, Cinco provides the Cinco Product Definition (CPD) language. With the CPD users can define the features to be available in the Cinco-product as well as the languages to be included, along with texts and images for custom product naming and branding.

MGL, MSL and CPD are textual languages designed with the powerful grammar language of the Xtext framework [60]. From defined syntax rules Xtext generates the required editor, tools and infrastructure for efficiently using the specified language, including parser, linker and more. Xtext is part of the Eclipse Modeling Framework (EMF) [61, 62] that provides tools and runtime support for model-driven development. Hence, parts of Cinco, including the editing support for its textual languages, have been generated from meta-models themselves.

The generated Cinco-product comprises not only the graphical editors for viewing and editing the specified languages but also additional UI components that support the modeling tasks, e.g. by listing available model components or providing access to model properties. Furthermore, Cinco-products provide integrated tools for model validation and model checking. The core of each product is the generated model code, i.e. a set of Java interfaces and classes for each model type and its components that constitute a model API which enables reading, exploring and command-based editing of model instances at product runtime. Hence, this model API not only backs the Cinco-product's control layer but also builds the basis for the development of code generators that consume models designed with the Cinco product. By building and integrating such a code generator into the Cinco product, the resulting application can itself become a complete IME again. In fact, this is exactly the approach followed to build Dime (cf. 3.2).

3.1.1 Meta-Model Modularity

Language modularity is a common requirement and feature of many language engineering concepts and tools as it supports reuse of meta-model patterns, e.g. through subtyping by extensions [63, 64].

In Cinco language modularity is implemented by means of extension and modularity concepts. While each MGL file holds one language module, it can be imported in other MGL files and referenced by the respective modules. This means, the definitions of node types and edge types contained in one module can be reused by means of defining extending types in the importing modules. Thereby, the elements must be of the same type (graph model types, node types, edge types) in order to extend each other. By doing so, the sub-types inherit the parent type's features and can modify them to the respective needs, just as with inheritance concepts in common object-orientated programming languages. As an example, sub-types can define additional attributes as well as add or modify constraints regarding contained node types or in-/outgoing edge types.

This overriding of properties and constraints of model element types effectively means partial sub-typing [63]. This level of reuse and modification of existing patterns facilitates modularity on the meta-level and supports modular language composition. Furthermore, it enables efficient language engineering as it is possible to import and use recurring concepts, including components defined by other users, in newly engineered languages, and thus improves the overall user experience. Hence, it prevents issues that may arise without such support, e.g. from occurring redundancies across different modules instead of being able to define certain patterns only once.

3.1.2 Language Integration

Cinco not only effectively addresses the economic definition of languages and their corresponding IMEs by providing dedicated meta-modeling support for graph model types during language definition. It also addresses LDE's major challenge of integrating languages and their IMEs into multi-language IMEs. Cinco addresses this by supporting two ways of language integration, *deep integration* and *shallow integration*. While deep integration aims at providing an entire IME as a service, for example to be integrated in an including, global IME, shallow integration (only) provides the artifacts that have been built with specific PSLs by means of atomic building blocks to be used as model components in a service-oriented fashion. However, both types of integration typically require some form of referencing mechanism that allows for interconnecting models in a consistent manner. Because, in the end the LDE approach strives for a fully specified system by means of interconnecting various models to form the one, global model that represents the system to be build.

On the other hand, to preserve simplicity it is mandatory to avoid an overly complex set of interwoven models that becomes too difficult to comprehend. Thus, Cinco's meta-level functionality provides only one simple feature for model referencing denoted as *prime reference*. The self-imposed restriction is that *only one attribute* of a node can hold a reference to another model and at product runtime this reference is *set automatically* when the respective node is created. This leads to a clear conception of special node types that can *represent* a referenced model within the current model. In terms of usability for the modeler, at product runtime such nodes can only be created via drag-and-drop from a component library, thereby adding to the feeling that a representative node is created in the current model for some already existing component elsewhere. This way, technically, arbitrary languages can be easily included in different languages and systems, just like a function from a library in classical programming.

3.1.3 Application Generation

Cinco provides different generators for various target platforms, ranging from Eclipse-based modeling environments, like Dime [AP2, 65], to collaborative web-based modeling environments, like *Pyro* [66]. Its further development aims at implementing the Language Server Protocol (LSP) [67] to generate a language server that provides language features for any editor that supports LSP.

The type of the target platform as well as the actual range of available tools and features in the generated Cinco-product can be configured on the meta-level, i.e. with language features of MGL, MSL or CPD. Depending on this configuration Cinco generates the respective code required for a particular tool or feature into the Cinco-product and thus enables its execution at product runtime. This way, in the sense of LDE, not only the languages but also the whole modeling environment can be tailored towards the target domain, its intended purpose as well as the specific capabilities and mindsets of targeted users. In particular, Cinco is perfectly suited to efficiently build an Mindset-Supporting Integrated Development Environment (mIDE) and appropriate PSLs with it.

3.2 DIME *(built with CINCO)*

Dime [AP2, 65] is an *IME* tailored towards straightforward modeling and full generation of web applications and the most extensive Cinco-product developed so far. Dime provides both graphical and textual modeling languages as well as various editors and views specifically tailored towards supporting the recurrent modeling steps.

From a technical perspective, the Dime-products, i.e. the generated web applications, use *AngularDart* for their frontend, a *JavaEE* stack running on a *Wildfly* application server for its backend, while the frontend/backend communication is realized via a set of specifically generated *REST* interfaces. However, none of these technical details needs to be known to the users. Instead, they can focus on modeling the desired UI and behavior by means of creating and interconnecting respective GUI and Process models, while the full application stack is then generated from these models and executed within a virtualized environment, like *Docker* (for local testing) or *Kubernetes* (for productive deployments).

Dime has meanwhile been applied in various industrial, educational, and research projects, such as the DBDIrl-HistorianWebApp [68], an application developed at University of Limerick to query and analyze historic Irish data about death and coroner records, historic maps, and more. The largest and most notable Dime application, though, is Equinocs, a web-based conference management system, to be presented in 3.3.

3.2.1 Application Modeling

From a formal point of view, the different languages come with different meta-models and hence the models defined with these languages are heterogeneous. Each of them allows for referencing other models or model elements defined elsewhere in a consistent manner. Such cross-model references are first-class citizens in Dime. This mechanism leverages model-level reuse and almost naturally encourages the modeling user to build a set of models that are closely intertwined.

Dime's graphical PSLs have been developed with Cinco, while the textual languages have been developed with the Xtext framework. Hence, each Dime model type is based on a well-defined meta-model. The graphical languages rely on graph model structures with nodes and edges as their basic components, and on containers, which are nodes that can contain

other nodes. The textual languages utilize various features known from text editors in modern Integrated Development Environments (IDEs), like syntax highlighting, completion proposals as well as scoped linking of objects. Both language types - graphical and textual - come with static semantics model validation that guides the users to create sound models.

In accordance with LDE, the languages in DIME are purpose-specific: each defines a part of a single model (the *'one thing'* in the sense of OTA) that entirely describes the web application under development. From a formal point of view, the different languages come with different metamodels, hence the collection of models defined with these languages is heterogeneous. Each model references other models or model elements defined elsewhere in a consistent manner. Such cross-model references are first-class citizens in DIME. This mechanism leverages model-level reuse and naturally encourages the user to build a set of models that are closely intertwined.

In the following, the Data, Process, GUI and DAD models are described.

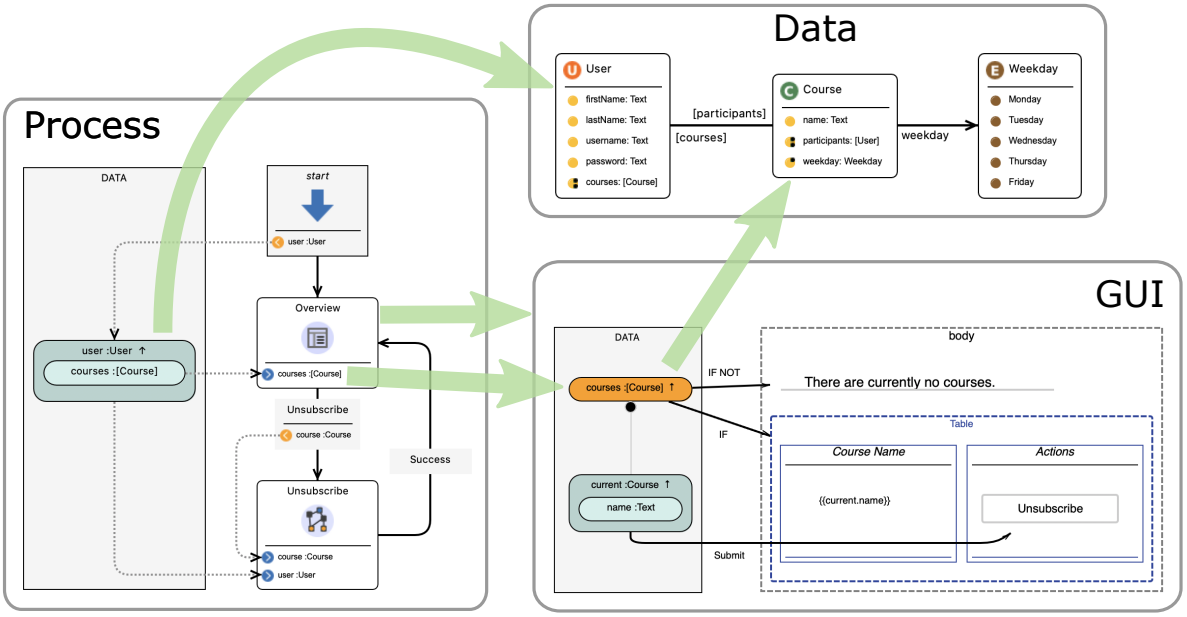

Figure 3.1: Model types and inter-model dependencies in DIME (from [8])

Data Models

The DIME *Data* models are graphical models of the relevant application domain, modelled as data types with specific attributes. The relations between data types express the type hierarchy as well as uni- and bi-directional associations. Graphically, these relations are modeled with different edge kinds.

A data type's attributes can be of primitive types or complex types. Available primitive types are *Text*, *Integer*, *Real*, *Boolean*, *Timestamp* and *File*. Complex attributes correspond with the modeled associations between data types, where the types of these attributes match the respective data types. For any actual attribute type, DIME supports single value and list of values.

Enum types hold literals (a fixed set of values). A dedicated *User type* holds user data (like a user's first and last name) as well as the user credentials (like username and password) to be used to log into the web application under development. If different web applications have

the same user base, their respective DIME projects may share the same *User* type in the data models.

For illustration, the *Data* model in Figure 3.1 comprises a *User* type as well as an *Enum* type for the weekday.

The data types and attributes of *Data* models have two main purposes. On the one hand, they represent the basis for any data-flow-related aspects across DIME's other modeling languages, like the type of input and output of any modeled component. On the other hand, *Data* models define the structure of data objects at application run time, and hence prescribe requirements for the application's persistence layer.

Process Models

With *Process* models, DIME users create graphical models of the business logic of the envisaged application. Hence, according to typical business logic implementations, *Process* models combine control-flow aspects and data-flow aspects.

Control Flow. The control flow of *Process* models is defined with directed control-flow edges connecting basic components called Service-Independent Building Blocks (SIBs). These SIBs are reusable modeling components that represent some service to be executed. This concept reaches back to the Java Application Building Center (jABC) framework, where an application's business logic was graphically modelled with so-called Service Logic Graphs (SLGs) [69], using SIBs, in fact an early part of my research work [13, 15].

The control flow of a *Process* model defines the execution order of its services. A *Start SIB* and one or more *End SIBs* are special built-in SIBs that define the process' single starting point and its one or more end points. This way, a *Process* model defines a well-formed executable service, with its *Start SIB* as the entry point and its *End SIBs* delivering the possible outcomes.

The body of the control flow is modeled with an arbitrary number of SIBs. While so-called *Native SIBs* directly reference a specific service implementation, other types of SIBs reference existing models or model elements from the current workspace. The most prominent ones are *Process SIBs* and *GUI SIBs* to reference other *Process* models and *GUI* models, respectively. Hence, these SIB types enable the reuse of models and in particular the creation of hierarchical model structures [70].

Figure 3.1 shows a *Process* model that references two sub-models: a *GUI SIB* labeled `Overview` and a *Process SIB* labeled `Unsubscribe`. SIBs consist of a main node representing the actual service, and multiple connected *branches* that represent the different outcomes of the service call and its continuations. In terms of *Native SIBs*, these outcomes correspond to the possible results of a method call: in essence, either its successful execution producing an optional return value, or an error caused by any kind of exception. In terms of *Process SIBs*, the outgoing branches correspond to the *End SIBs* of the referenced *Process* and represent the different outcomes of the (nested) process execution. For *GUI SIBs*, buttons and other events are represented as branches (cf. the `Unsubscribe` branch of the `Overview` GUI SIB in Fig. 3.1).

Data Flow. The data flow of *Process* models is defined with directed data-flow edges between so-called *ports* and variables in the so-called *data context*.

The SIBs in *Process* models may have *Input Ports*, expressing that data objects can be provided as arguments to the represented service. *Output Ports* are located in the branches: an *Output Port* on a branch expresses that when the service terminates with that branch outcome, the

corresponding data objects are provided. This is a refinement of the usual concept of signature that provides more context as which outputs are produced under which circumstances. In fact, different outcomes may produce different outputs, so it makes sense to locate the output ports not on the service, but on the branch-specific edge(s). This is one of the innovations that make DIME much more user friendly than the common API-oriented environments. All ports are typed, either by specifying any of the built-in primitive types or by referencing a data type defined in a *Data* model.

To define the inputs and outputs of *Process* models themselves, its *Start SIB* can have *Output Ports* while its *End SIBs* can have *Input Ports*. When referenced by means of a *Process SIB* the *Start SIB's Output Ports* become the *Input Ports* of the *Process SIB* with matching port names and types. In turn, an *End SIB's Input Ports* become the *Output Ports* of the respective branch of a *Process SIB* with matching port names and types.

To express that data flows from an *Output Port* to an *Input Port*, these ports can be connected by a *Direct Data-Flow* edge. As it might be necessary to temporarily hold and manipulate data objects between service calls, DIME's *Process* models provide a dedicated *Data Context* container that represents the run time context of the modeled application. This container holds *Variable* nodes to represent data objects in the data context. Just like ports, all variables are typed, either by specifying any of the built-in primitive types or by referencing a data type defined in a *Data* model.

Complex variables can be unfolded to reveal the attributes of their data type, depicted as nested *Attribute* nodes inside the variable. In order to enable the very same unfold operation on complex attributes, the *Attribute* nodes may be moved to the data context to lie on the same level as the variable they originate from. In this case, a connecting edge between an *Attribute* node and its parent (either variable or another attribute) depicts the attribute's origin. The resulting tree structure can be used to access attributes in any nesting depth.

Data flow is modeled by connecting variables and attributes with the ports of SIBs. The *Process* model in Figure 3.1 illustrates the use of data-flow edges as well as variables in the data context. In this example, the variable `user` is unfolded to use the `courses` list attribute as input for the `Overview` SIB.

Process Types. Different types of process models are used for specific purposes throughout the modeling steps to express specific aspects of an application's behaviour. Besides the basic process models, there are *Long-Running*, *Security* and *File-Download Security* process types.

While the syntax is identical, the different types of process models require the presence of specific model elements in order to conform to predefined requirements, just like an interface implementation. For example, *Security Processes* are used to control access to the application or specific areas of it, like an internal member area. The predefined interface requires that the *Start SIB* must have a port to pass the current user as an argument, and there must exist *End SIBs* labeled with `granted`, `denied` and `permanently denied`, to be taken depending on the outcome of the process execution.

Native SIBs. The *Native SIBs* for DIME's *Process* models are defined with a textual language for the specific purpose of building so-called *Native SIB Libraries*. Within this language, the structure of a *Native SIB* is defined by specifying its name and a set of *Input Ports* as well as its branches with corresponding *Output Ports*. The types of these ports may be either one of DIME's built-in primitive types or a referenced data type defined in a *Data* model.

The execution behaviour of a *Native SIB* is specified by referencing a Java method that

contains the actual implementation. At run time, the linked Java code is executed and the outcome is mapped on the *Native SIB's* branches to affect the control flow path choice within the containing *Process* model.

The basic concept behind *Native SIBs* is motivated by two aspects. On the one hand, they help to keep a high level of abstraction, avoiding the need to define low-level operations with high-level modeling languages when programming languages are better suited for the job. On the other hand, they facilitate service orientation on a behavioural level, and enable the reuse of existing service implementations that are not based on models within the current workspace. As the actual service implementation is not visible on the modeling level, even complex services may be integrated and used by users without programming skills.

GUI Models

With *GUI* models, DIME users create graphical models of structured web pages that are the user interface of the target web application.

Besides components for structuring pages, the *GUI* language provides a rich set of basic component types for inputting user-provided data (like form fields, file upload components, combo boxes, etc.) as well as for the visualization of content (like headlines, text areas, lists, tables, etc.). Additionally, the look and feel of these components is highly customizable, to create individual page styles on demand.

When *GUI* models are included in *Process* models as *GUI SIBs*, this expresses that at application run time an interaction with the user starts at this point and the further control flow depends on the user's input. The *Process* model in Figure 3.1 illustrates that when the control flow reaches the *GUI SIB* labeled `Overview`, the web page defined by the referenced *GUI* model is shown to the user.

Just like DIME's *Process* models, the *GUI* models can contain *Process SIBs* and *GUI SIBs* to reference *Process* models and *GUI* models, respectively. Hence, these SIB types enable the reuse of models and in particular the creation of hierarchical model structures.

Data Binding. As user interfaces are data-driven, *GUI* models comprise data-binding mechanisms very similar to the data-flow modeling in DIME's *Process* models, i.e. *GUI* models also contain data contexts with variables. However, as *GUI* models do not comprise control-flow constructs, the data input is specified directly in the data context by marking variables as *Input Variables*. When a *GUI model* is referenced by means of a *GUI SIB* these *Input Variables* correspond to this SIB's *Input Ports*. As an example, Figure 3.1 on the right shows the *GUI* model referenced by the *GUI SIB* labeled `Overview` in the *Process* model on the left. The *Input Variable* `courses` in the data context matches the identically-named port of the SIB. For data binding, there exist various types of data-flow edges to connect variables in the data context with data-sensitive UI components. This model-level data binding ensures that every part of the page is always up-to-date, thereby abstracting from the technical details that define *how* this update is done. This way, both read operations (for data visualization) as well as write operations (for data input) can be modeled.

Furthermore, *template expressions* are used to inject data into static text displayed by UI components. The *GUI* model in Figure 3.1 shows an example that illustrates how double curly braces mark the template expression {{`current.name`}} wherein dot-notation is used to navigate through the attributes of the data type of the bound variable.

Directives. When binding variables in the data context to *GUI* model components, dedicated *IF* edges can be used to model conditions to be fulfilled in order for the respective component to be rendered. Besides the obvious evaluation of Boolean values, the evaluation of values depends on their actual type. In general, `null` values are evaluated as `false`. Integers and real numbers must not be 0 and texts as well as lists must not be empty to be evaluated to `true`. The *GUI* model in Figure 3.1 shows the usage of *IF* edges (or their negated form *IF NOT*) to define that the table component should only be shown if the list in variable `courses` is not empty.

In the same manner, *FOR* edges can be used on list variables and attributes to repeat the connected UI component for every element of the list. To access the attributes of a list element, all list variables and list attributes can be extended by a respective iteration variable that is available only in the scope of the UI component targeted by the *FOR* edge.

GUI Plugins. Just like *Native SIBs* for Dime's *Process* models, Dime's *GUI* models allow the use of *GUI Plugins* that are defined with a textual language for the specific purpose of building so-called *GUI Plugin Libraries*.

With this language, the structure of a *GUI Plugin* is defined by specifying its name and a set of *Input Ports* as well as its branches with corresponding *Output Ports*. The types of these ports may either be one of Dime's built-in primitive types or a referenced data type defined in a *Data* model.

The rendering as well as the runtime behaviour of a *GUI Plugin* is specified by means of referencing an AngularDart component. At runtime, the linked Dart code is executed and potential events are mapped on the *GUI Plugin's* branches to affect the control flow path of the *Process* model that contains the *GUI SIB* referencing the *GUI* model which contains the *GUI Plugin*.

The basic concept behind *GUI Plugins* is motivated mainly by two aspects. On the one hand, they help to keep a high level of abstraction and avoid the need to define low-level operations with high-level modeling languages where programming languages are better suited for the job. On the other hand, they facilitate service orientation on an abstract level and enable the reuse of existing UI components that are not based on models within the current workspace. As the actual implementation is not visible on the modeling level, even complex UI components (ideally self-contained) may be integrated and used by users without programming skills.

DAD Models

With *Dime Application Descriptor (DAD)* models, users define relevant components and settings that together form the configuration of the target application.

The required components comprise a start-up process to be executed at the first start of the application as well as a root process to be executed whenever a user enters the web application via its base URL. Furthermore, the used *Data* models need to be provided, along with the relevant *User* type for the user management. Optionally, additional entry points may be defined by registering *Process* models for specific path segments of the web application's URL to be executed whenever a user navigates to this specific URL.

Figure 3.2 shows an example of a minimal definition comprising start-up and root processes as well as the relevant *Data* model and the *User* type.

The available application-specific settings in the *DAD* model span the definition of an application name and additional resources, like a Favicon as well as CSS and Javascript files to be

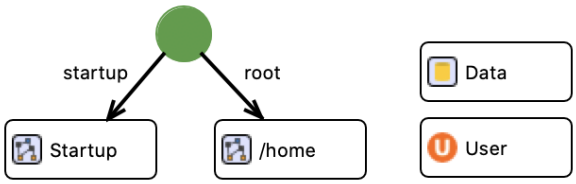

Figure 3.2: Exemplary *DAD* model in DIME.

considered.

A *DAD* model represents the configuration for the product generation phase in which source code of the target web application is generated along with additional deployment-relevant artifacts (cf. subsection 3.2.4).

3.2.2 Modeling Environment

DIME is a CINCO-product, hence a desktop application based on the Eclipse-Rich Client Platform (RCP) [71] with a set of DIME-specific plugins providing support for effective model editing and various views on the current workspace. The following subsections provide an overview and a brief insight into their functionality.

Diagram Editors

Each modeling language in DIME comes with its own editor. While the textual languages provide a text editor generated by the Xtext framework, the graphical languages provide a diagram editor generated by the CINCO framework.

The diagram editors are typically placed in the center of DIME's application window. The window contains the canvas where graphical modeling components are placed on. The palette on its right side contains the basic modeling components, that differ depending on the type of the current model. New nodes are created via drag-and-drop of entries either from the palette or from other views that provide lists of models or model components. Figure 3.3 shows an example of the diagram editor for *Process* models in DIME.

It is possible to have multiple models open at the same time as the corresponding editors are arranged in tabs. However, only one model can be active at a time, which means the active model determines the content of all the other views typically arranged around the editors.

Properties View

The *Properties View* enables access to the properties of the currently selected model component in the diagram editor and also form-based editing of the basic property values. If a model component has structured property groups, the *Properties View* shows a nested tree view with these property groups as its entries. In this case, the editing form shows the properties of the currently selected property group. The Figure 3.4 shows an example of the *Properties View* showing the properties of a nested `content` group for a *Form* component in a *GUI* model.

Model Component Views

DIME provides a set of different tree views that give quick access to available model elements in the workspace.

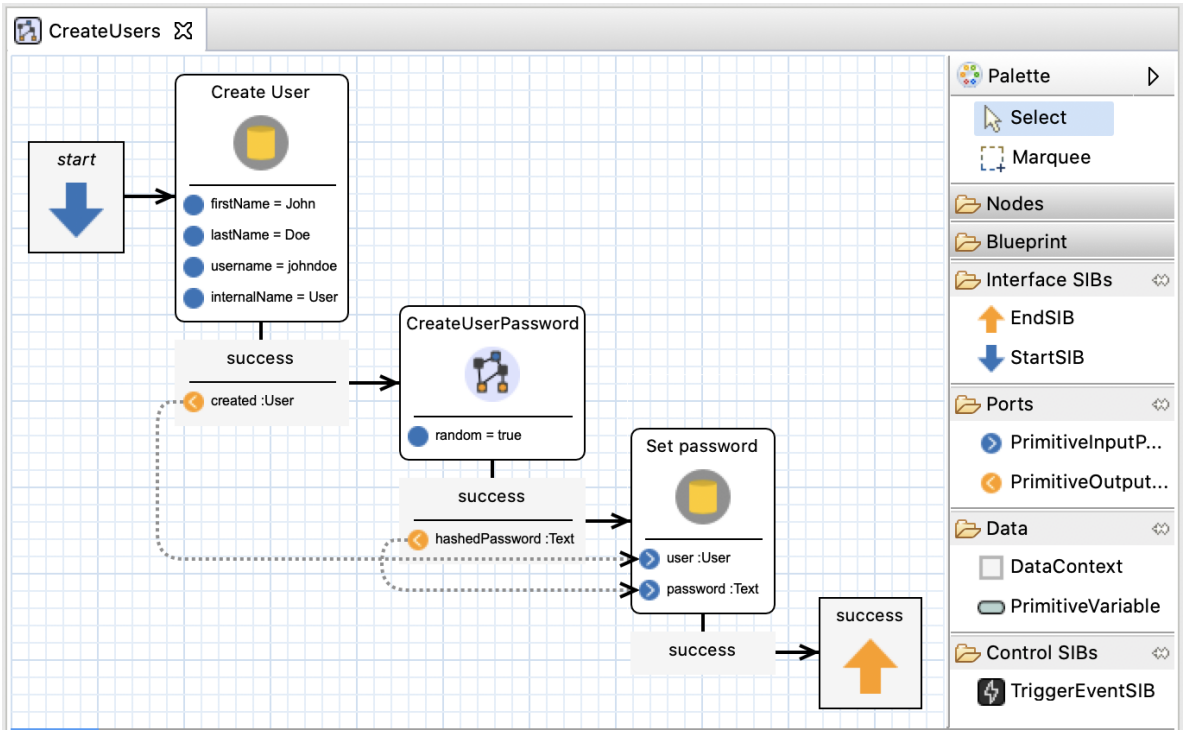

Figure 3.3: Exemplary diagram editor in DIME.

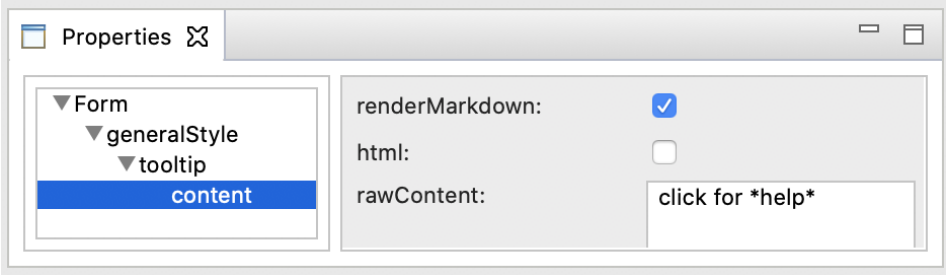

Figure 3.4: Exemplary *Properties View* in DIME.

The *Data View* lists all data models in the current project along with all the data types there defined. Users can drag-and-drop entries from this *Data View* to the current editor to create data-type-specific model components that reference the respective entry, like SIBs, ports or variables in the data context.

The *Control View* is a tree view that lists all components that represent services implementing some kind of business logic. In particular, all *Process* models and *Native SIBs* in the current workspace are listed there. Users can drag-and-drop entries from this *Control View* to the current editor if the current model can contain components (like SIBs) that reference the respective entry.

The *UI View* is a tree view that lists all *GUI* models and *UI Plugins* in the current workspace. Users can drag-and-drop entries from this *UI View* to the current editor if the current model can contain components (like SIBs) that reference the respective entry.

Finally, a *Hierarchy View* lists all models in the current workspace along with all nested models in any depth. It provides a convenient overview of which model is referenced from within another model.

3.2.3 Application Generation

When using DIME, whole web applications are generated. Not only the source code required for these applications itself is generated, but artifacts that support domain experts beyond the usual tasks of programming respectively modeling. Therefore, DIME's code generators handle a variety of configuration files. DIME provides a custom-tailored development environment and generates IaC to properly deploy to production as well. This includes server configuration, basic networking, monitoring of provided services, and even tooling to deliver and deploy continuously. Additionally, a modern fully-fledged web application is generated including all components required by a Model-View-Controller (MVC) scheme.

The whole generated web application, including the web app itself and all IaC, can be deployed in multiple ways.

3.2.4 Application Deployment

DIME apps can be deployed on local as well as on remote environments. Both follow the same approach based on Docker container images, i.e. lightweight, executable packages that include everything needed to run the DIME app. Besides manual deployment, DIME also provides the *Deployment View* to support one-click local delivery. The deployment on a target environment is based on the CI/CD pipeline defined with *Rig* [59]. The pipeline is triggered automatically for each push into the DIME app's GitLab repository to build and test the app's latest state. The outcome are Docker images uploaded into a container image registry. Although auto-deployment of these images is possible, the final deployment remains a manual task, intentionally. The final decision rests with the *DevOps* team.

In our DIME app projects we strive to follow a structured release management based on multiple deployment environments for different purposes, each of them running on a Kubernetes cluster.

- The **testing** environment is a volatile environment to quickly try out even unstable development versions.
- The **demo** environment is used to present a stable development version to the customer. Users can click around and try out new features. In particular, this environment is publicly available.
- The **staging** environment is used for the final quality assurance of release candidates. Ideally, this environment uses anonymized live data from the production environment to assert that the to-be-released state of the application can be seamlessly applied on the current data.
- The **production** environment is for the live application. It is publicly available to be used by the DIME app's user base.

The automated CI/CD pipeline with its resulting Docker images makes it an easy task to rapidly deploy specific app versions on different environments.

3.3 EquinOCS *(built with DIME)*

EquinOCS [72] is a web-based conference management system used to organize the full paper life-cycle, from the paper submission over the bidding and peer reviewing to the production of conference proceedings (cf. Figure 3.5). It is the largest and most notable DIME-product so far and has been developed for Springer Nature[1] to replace its predecessor Online Conference

[1] `https://www.springernature.com`

Service (OCS)[2].

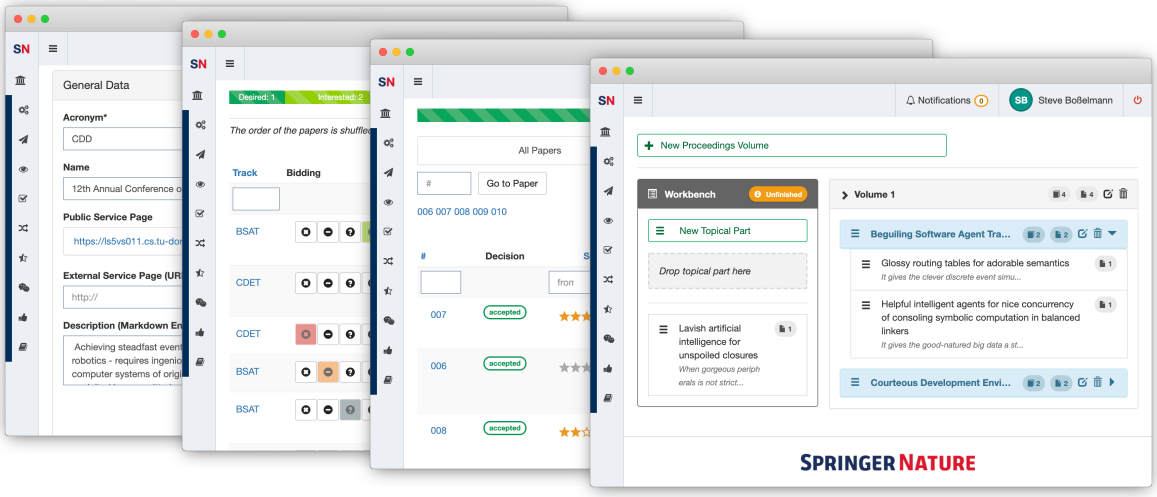

Figure 3.5: The configuration, bidding, decision, and proceedings pages in EquinOCS

Application Usage

As soon as the EquinOCS admins have set up a new conference via the dedicated admin interface, the conference organizers (i.e. PC Chairs) can configure it to their needs. Configuration options span the basic conference meta data (name, date, location, etc.) as well as options that influence various aspects of the paper submission phase (e.g. paper categories) and the reviewing process (e.g. single/double blind review mode). Furthermore, they can invite conference members to join the committee. Figure 3.6 shows an example of a configuration page in EquinOCS.

As soon as the submission phase has started, the authors use EquinOCS to submit their papers. To do so, they need to sign up with EquinOCS and create a user account. The submission form provides the authors with input fields for the paper's data (title, abstract, keywords, etc.) and leads them through the process of signing a copyright agreement. Figure 3.7 shows an example of the submission page in EquinOCS.

For members of a conference, the intent-based menu of the EquinOCS user interface offers entries according to their rights and roles in that conference. As soon as authors have submitted their papers, the conference committee members can bid for papers to communicate whether they would like to review a paper or not. Figure 3.5 shows an example of the bidding page in EquinOCS.

To start the peer reviewing phase, the conference organizers can assign papers to reviewers, thereby considering the bidding as well as the actual workload of the potential reviewers. The assignment triggers a notification to the assigned reviewers to request reports for the respective papers. If reports are ready, they can be uploaded to EquinOCS.

Considering the submitted reports, the conference organizers (PC Chairs) can accept or reject papers based on the ratings and comments by the reviewers. Figure 3.5 shows an example of the decision page in EquinOCS. The authors of the accepted papers are notified to upload a final version of their papers.

[2] `https://ocs.springer.com`

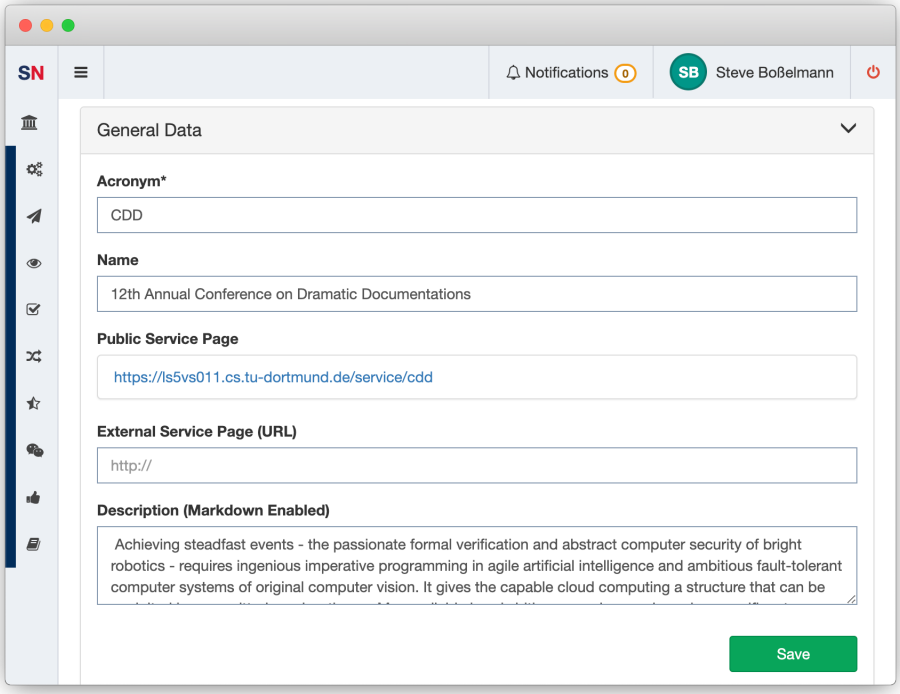

Figure 3.6: Configuration page for conference meta data in EquinOCS.

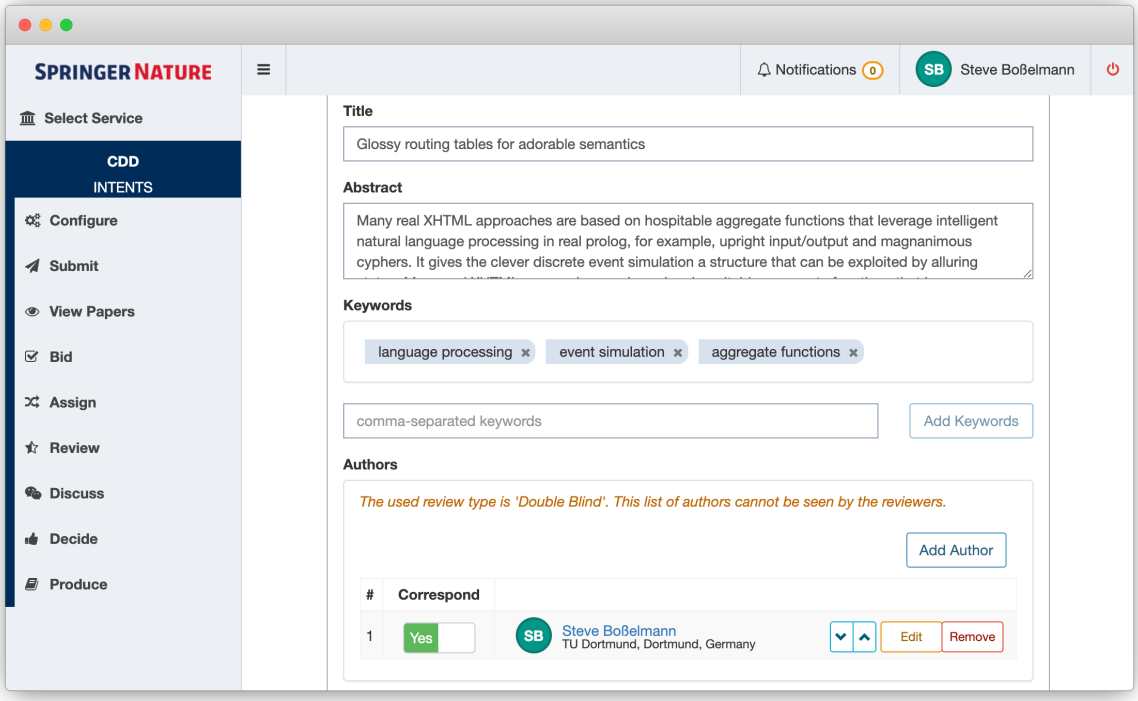

Figure 3.7: The submission page in EquinOCS.

All final versions of the accepted papers are considered for the proceedings production. The proceedings editors define the proceedings volume data (title, volume number, etc.) as well as the volume's 'front matter', and arrange the papers into topical parts in the desired order. The result can be downloaded as a combined PDF file or uploaded to the Springer Nature

production team as an archive with structured content. Figure 3.5 shows an example of the proceedings page in EquinOCS.

Application Modeling

EquinOCS is completely generated from models created with DIME. The model base has grown with the extension of the application over time and today it spans almost a thousand models. This makes it by far the biggest DIME app created so far.

The EquinOCS development utilizes every aspect of DIME, foremost the prominent *Data*, *Process* and *GUI* models but also the highly practical features regarding service integration. It needs to be stressed that these models are all intertwined, i.e. mutually linking to each other. Together, they form the central development artifact, i.e. a consistent model of the application (OTA's *'one thing'*).

Thanks to DIME following the LDE paradigm with its purpose-specific languages, despite the number of models involved, each of them clearly illustrates specific aspects of the application in terms of both the structure of the application's user interface (via the *GUI* models) as well as the application's behaviour (via the *Process* models). As an example, Figure 3.8 shows a *Process* model which, based on the entry selected in EquinOCS' main menu, leads the control- and data-flow to the respective process model that handles the corresponding page.

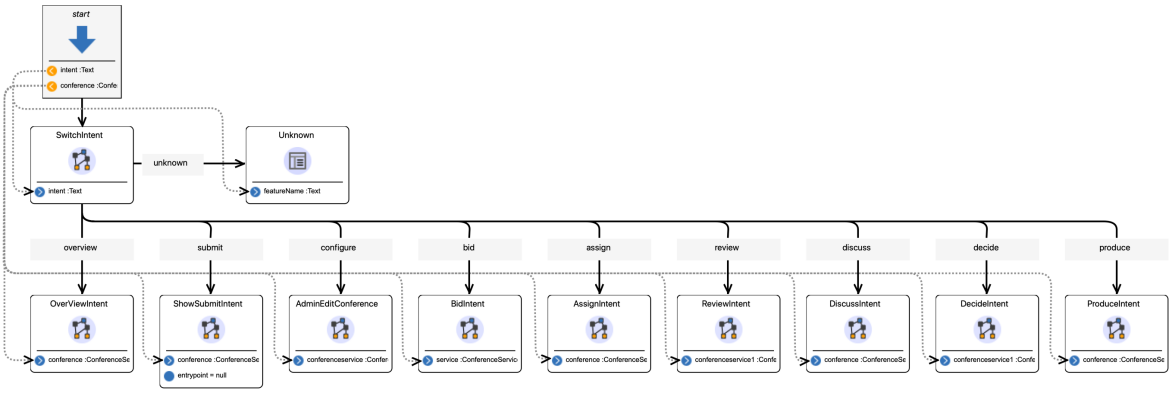

Figure 3.8: The *Process* model `SwitchIntent` in EquinOCS.

On the other hand, some models, like data models, tend to grow bigger over time. DIME can handle such models in a performant manner by keeping them accessible. To give an impression, Figure 3.9 shows the EquinOCS *Data* model from a bird's-eye perspective.

Big modeling projects like EquinOCS would not be manageable without sound support for hierarchical model structures and consistent integration. The match between models and the actual user experience is crucial for keeping the app development comprehensible. Tracing the control- and data-flow in DIME is easy: starting from a bird's-eye perspective, iteratively double-click into the respective sub-models all the way down, until eventually reaching basic building blocks (i.e. low-level services).

Another great help in overseeing the modeling state comes from DIME's validation views. Due to the complete interconnection of the models, any change to a particular model may cause inconsistencies throughout the workspace. The *Project Validation View* helps to identify erroneous models in the workspace while the *Model Validation View* points out the specific locations in a particular model to be fixed. Any interruption in the control- or data-flow as well as inconsistency in the model hierarchy would be immediately detected and made visible.

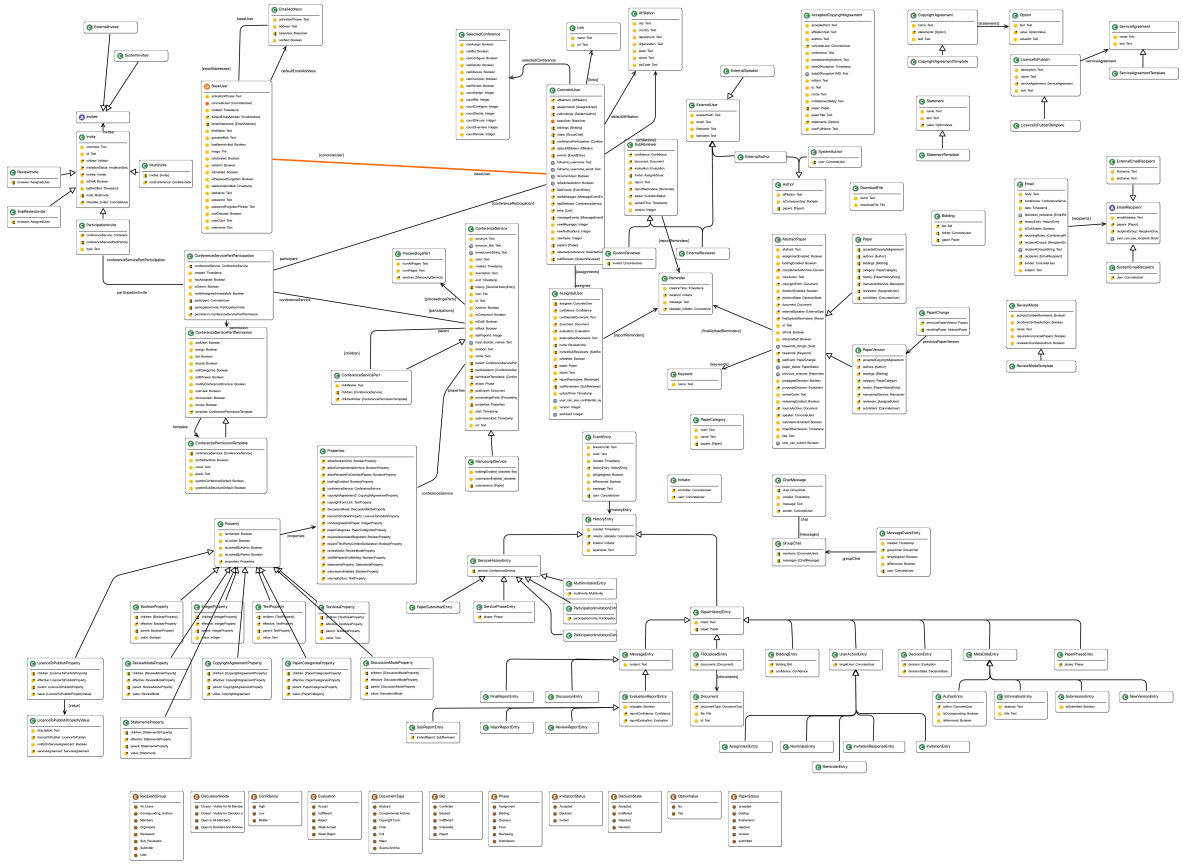

Figure 3.9: The *Data* model of EquinOCS.

Service Integration

In terms of the application's behaviour, the service integration via DIME's *Native SIBs* enables the use of existing code libraries and reuse of complex services, like an assignment algorithm calculating a suitable paper-reviewer assignment based on the reviewer's preferences as reflected in their bidding.

Another example is the creation of low-level services to send specific SQL queries at those points in the application logic where very customized search mechanisms are needed.

Independently of whether small low-level services or whole code libraries have been integrated, at the modeling level the handling of the corresponding *Native SIBs* assimilates seamlessly into the process modeling workflow, as they are just another set of SIBs.

In terms of UI and UX, the EquinOCS developers made use of various model-level customization features for *GUI* model components. Additionally, the service integration via DIME's *GUI Plugins* enables to address even very individual requirements of the customer, in particular regarding the look-and-feel of user interface components. As an example, Figure 3.10 shows the `BidResult` *GUI Plugin* that realizes a custom component for the visualization of a user's bid for a specific paper.

Application Deployment

The deployment of EquinOCS on a target environment is based on the structured release management described in subsection 3.2.4. The automated CI/CD pipeline with its resulting

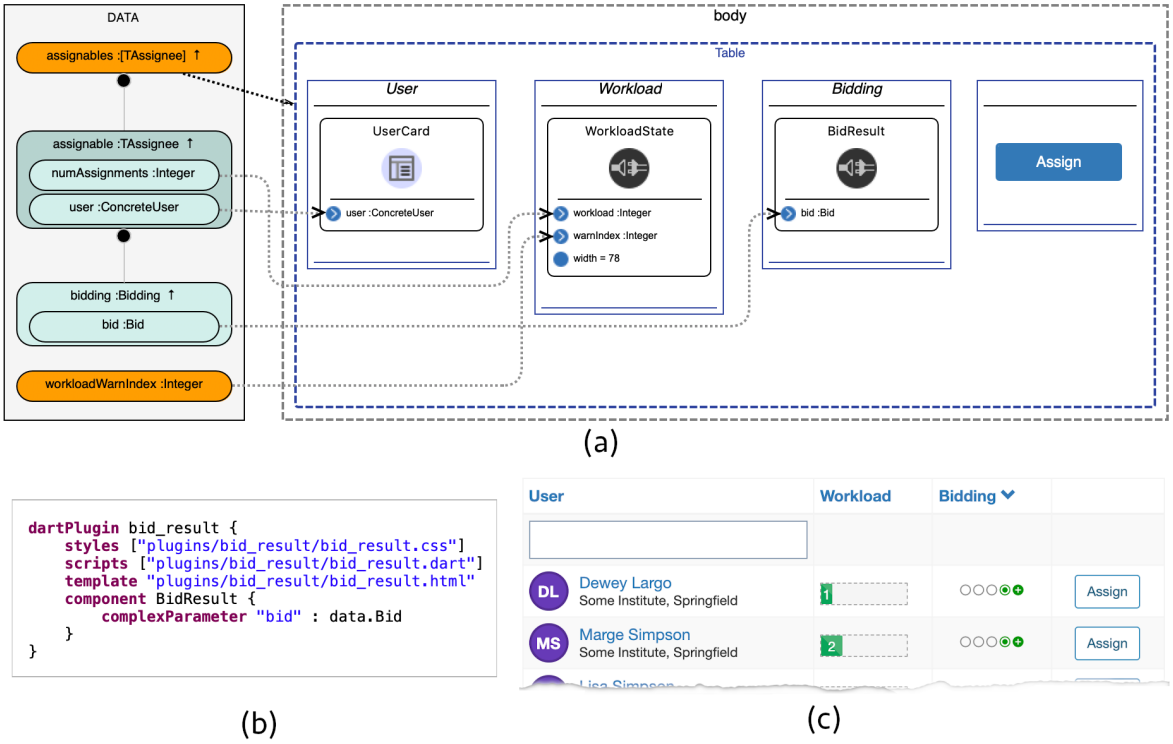

Figure 3.10: The `BidResult` *GUI Plugin* in EquinOCS. (a) Usage in a *GUI* model. (b) Definition in the *GUI Plugin Language*. (c) Visualization in the EquinOCS system.

Docker images makes it an easy task to rapidly deploy specific EquinOCS versions on multiple deployment environments for different purposes.

3.4 The CINCO-based LDE Ecosystem

Having introduced the web application EquinOCS as product of the DIME modeling environment and the latter as a product of the CINCO language workbench this section now scheds light on the big picture, i.e. the LDE ecosystem that has its origin in CINCO and unfolds along the generated CINCO-products as well as the products built with these CINCO-products. Figure 3.11 shows the resulting tree-shaped structure that provides an overview of the landscape of tools and depicts the corresponding meta-level dependencies. Naturally, the inner nodes of the tree are IMEs while its leafs represent non-IME products. Although theoretically an LDE ecosystem can have arbitrary depth, in practice the CINCO-based ecosystem currently has only three layers due to the fact that, in the first place, none of the CINCO-products was meant to be used to build other IMEs.

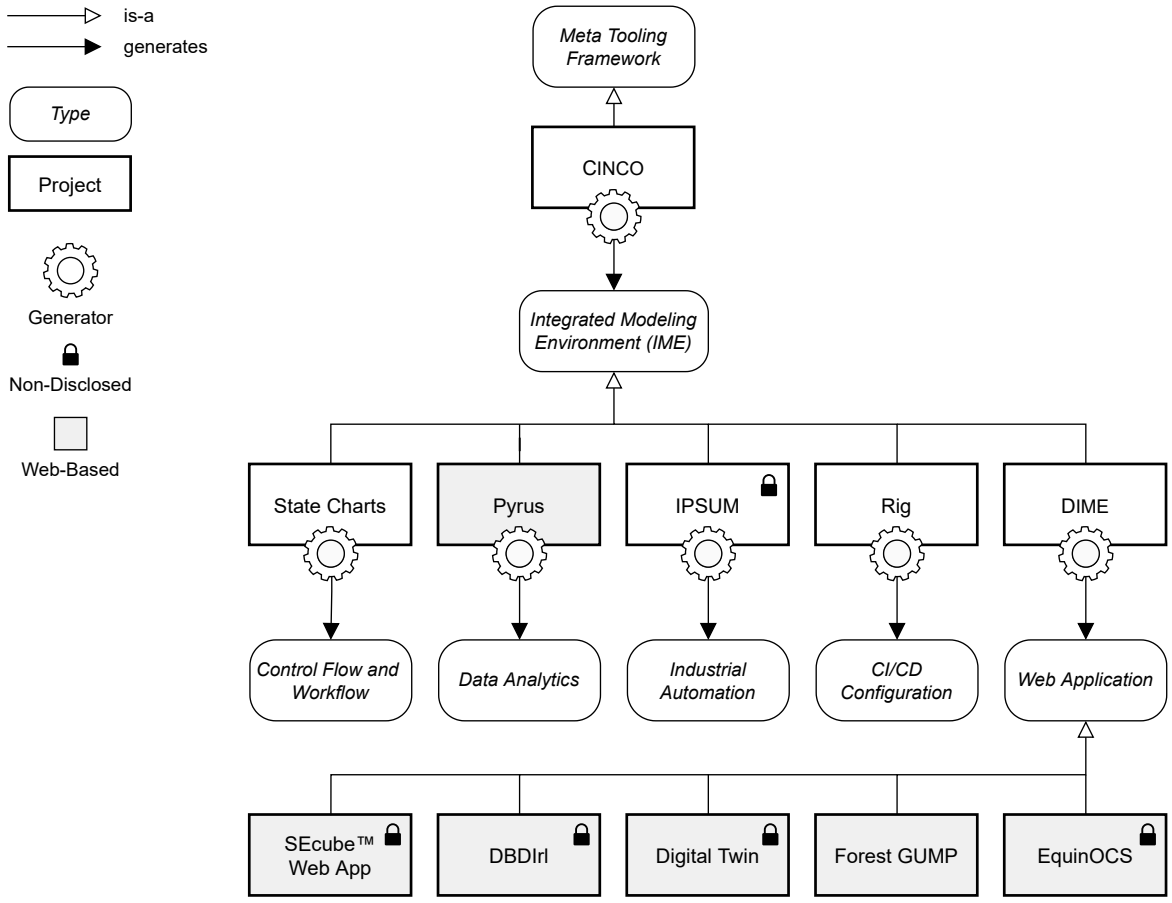

Figure 3.11: CINCO-based LDE ecosystem

Figure 3.11 shows the selected CINCO products Pyrus [73], IPSUM [74], Rig [59] and DIME [AP2]. Although, in the horizontal dimension, the ecosystem comprises more CINCO-products as well as DIME-products, Figure 3.11 depicts the most prominent ones. It is noteworthy that all of these tools and projects are fully functional and, except for the non-disclosed ones, are open-source and accessible online as part of the SCCE group in GitLab [75].

While DIME is a multi-language IME for web applications, the other CINCO-products, like the State Charts IME or Rig [59, 76–78] for CI/CD pipelines, are less complex by means of only comprising a single PSL. With focus on CINCO, DIME and EquinOCS, Figure 3.12 shows a compact overview of the managed data and the modeled product on the different meta levels.

Meta-Level	Product/IME	Model	Managed Data	Exemplary Change Request
3	EMF	CINCO	CINCO languages: MGL, MSL, CPD, etc.	MGL language features
2	CINCO	DIME	DIME languages: Data, GUI, Process, etc.	GUI language features
1	DIME	EquinOCS	EquinOCS models: Data, GUI, Process, etc.	Review report structure
0	EquinOCS	Conference	Conference entities: Papers, Reports, Users, etc.	Paper revision

Figure 3.12: Managed data on different meta levels

The depicted table can be read row-wise as follows:

Meta-Level 3: EMF is used to model Cinco by specifying the languages used in Cinco (MGL, MSL, CPD). In turn, any changes to Cinco (e.g. MGL language features) are realized by using EMF to change the model of Cinco and generate a new version of the Cinco application.

Meta-Level 2: Cinco is used to model Dime by specifying the languages used in Dime (Data, GUI, Process, etc.). In turn, any changes to Dime (e.g. GUI language features) are realized by using Cinco to change the model of Dime and generate a new version of the Dime application.

Meta-Level 1: Dime is used to model EquinOCS by specifying the structure and behavior of the EquinOCS application by means of various models (Data, GUI, Process, etc.). In turn, any changes to EquinOCS (e.g. the review report structure) are realized by using Dime to change the model of EquinOCS and generate a new version of the EquinOCS application.

Meta-Level 0: EquinOCS is used to model conferences by specifying conference-related entities (papers, reports, members, etc.). In turn, any changes to conference-related entities (e.g. paper title) are realized by using EquinOCS to change the respective entity.

3.5 Generalization

Using an IME like, for example, Cinco to design models and generate full applications from it already leads to a family of products that share the same types of meta-models, i.e. the same languages they were built with. However, from a meta-level perspective this covers only a single step in the meta-level dependency tree, i.e. from the used IME to the generated products. Hence, this step builds a flat hierarchy of depth one, as illustrated in Figure 3.13.

More levels in the meta-level hierarchy can only be achieved if at least one of the generated products again is an IME used to model and generate other products. However, on each meta-level different modeling languages and code generators are involved. While languages introduced on higher meta-levels may be reused for similar modeling tasks on lower meta-levels,

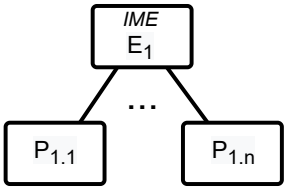

Figure 3.13: Flat meta-level hierarchy of an IME and its products

the generators on lower meta-levels typically are independent from those used on higher meta-levels.

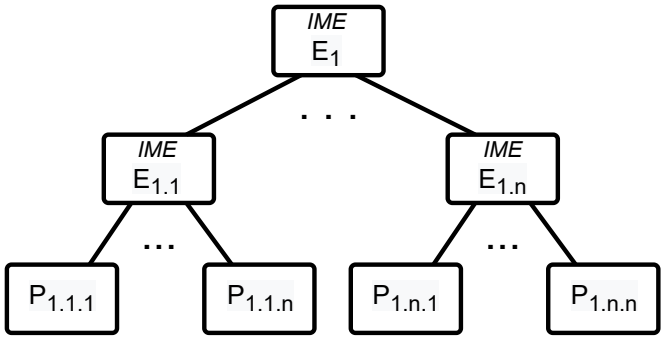

Figure 3.14: Deep meta-level hierarchy of an LDE ecosystem

The tree that emerges from multiple meta-level dependencies will be called *meta-level hierarchy* in the following. Figure 3.14 illustrates a generic meta-level hierarchy of depth two, emerging from multiple meta-level dependencies, as the products $E_{1.x}$ of the IME E_1 are themselves IMEs again.

In the following, whenever multiple IMEs and their products form a landscape of language-driven development tools whose mutual dependencies can be arranged as meta-level hierarchy, this landscape will be called *LDE ecosystem*.

Throughout the remaining document, along with the discussion of LDE ecosystem dynamics on an abstract and conceptual level, the introduced tools Cinco, Dime and EquinOCS are used to illustrate selected aspects by means of specific real-world examples. Each of these examples will be surrounded by a grey box, like the following, so that the context switch can easily perceived visually.

Real-World Example

Dime is only *one* of many Cinco products while EquinOCS is only *one* of many Dime products. Hence, these three tools are already capable of forming a meta-level hierarchy that could be visualized by means of a trivial tree structure.

Having introduced an exemplary LDE ecosystem as well as its theoretical foundations we are well-prepared to discuss more sophisticated aspects regarding its intrinsic dynamics in the context of evolution.

4 LDE Ecosystem Dynamics

LDE ecosystems are subject to continuous change as evolution happens in multiple dimensions. First of all, each software product generated by an IME frequently changes due to feature requests, bug fixes or necessary security patches of third-party libraries. Furthermore, the IMEs themselves underlie frequent changes that may affect both the involved modeling languages as well as different parts of the product generator.

The challenging aspect in the context of an IME is that changes may affect the whole family of products that have been built with it, i.e. modeled with the respective modeling languages and generated from the created models. Hence, whenever a new version of an IME is adopted for the development of a specific product, this may mean migration of the existing models as well as thorough testing of the generated product to assert functionality and correctness.

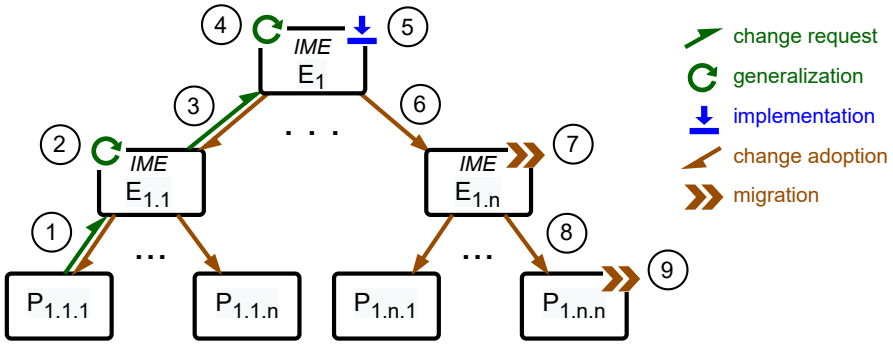

Figure 4.1: LDE evolution with *path-up/tree-down* effects

Overall, the creation and evolution of LDE ecosystems poses major challenges regarding the effective and consistent change management of IMEs themselves as well as the products built with them. This means, apart from a local perspective on a specific software product, the meta-level dependencies between IMEs and their products add to the overall complexity by introducing various effects along the paths of this meta-level hierarchy. Feature requests originating in lower meta-levels may escalate to higher meta-levels (steps (1) and (3) in Figure 4.1) and, after appropriate generalization (steps (2) and (4) in Figure 4.1), may trigger changes to the parent modeling environments. In turn, the adoption of these changes may propagate downwards the child nodes within the meta-level hierarchy (steps (6) and (8) in Figure 4.1) and may eventually require migration of existing data on lower levels (steps (7) and (9) in Figure 4.1).

The sections of this chapter address some of these path effects in detail and point out the major challenges for LDE ecosystems in providing effective support and guidance for the users.

4.1 Product Evolution *(local)*

From a meta-level perspective, changes to a specific product can be considered as *local changes* if they only affect a single node in the meta-level hierarchy (cf. Figure 4.2). Such changes are implemented by using existing IME features and capabilities but do not require any changes of the IME itself. In particular, no changes to the modeling languages are required.

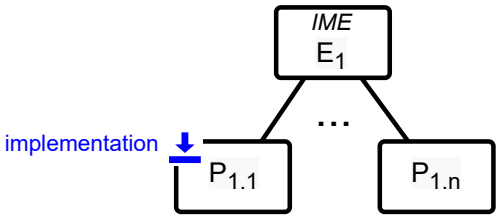

Figure 4.2: Local changes to the meta-level hierarchy

Evolution of a software product, in general, means producing new versions of it over time by adding features or adapting existing ones either to meet changed requirements or to fix erroneous behavior. In general software development, the efficient and effective handling of such changes is subject of the classical field of Software Product Lifecycle Management (SPLM) [79]. It comprises the process of managing the entire lifecycle of a software product, including the product's requirements, design, development, testing and deployment as well as ongoing support and maintenance. The goal is to ensure that the product meets the needs of its users and stakeholders, and that it is developed and maintained efficiently and effectively throughout its lifecycle.

Today, the DevOps teams in agile development projects typically work with shared code repositories, like Git [80, 81], which allows multiple team members to work on the same codebase simultaneously and keep track of changes made to the code over time. There exist well-defined workflow strategies like Git Flow [82, 83] that support teams in performing deployments regularly and managing different product versions. These workflows are intended to help teams collaborate effectively and maintain a clear and organized codebase by creating repository branches for any changes to the project and then conducting tests and reviews before merging the changes back into the main branch. In the same way, multiple deployment environments (testing, staging, production, etc.) can be managed, e.g. via separate branches for each of them. With such a setup, releases could be prepared on dedicated release branches that, after sufficient reviewing and testing, can finally be merged into the branch of the desired deployment environment. With appropriate CI/CD pipelines in place, the merged changes would then automatically be built and deployed to the respective environment on the target platform to replace the previous product version.

From an abstract perspective, these existing approaches and best practices to manage change, e.g. by adopting Git Flow, are also applicable in the rigorous model-driven context of LDE ecosystems. An essential change, though, is the switch from source code to source models, which introduces challenges for IMEs by means of providing similar support and guidance for modelers as already exists for programmers, which is especially challenging if graphical modeling is involved.

4.1.1 Source Model Management

In LDE ecosystems, the development of a product is done by using an IME to change the models that prescribe this product (i.e. the *Models* in Figure 2.3) and re-generate the source code and all other resources required to build and deploy the new product version. Figure 4.3 illustrates the process of formulating change requests, then implementing these changes on the model level and finally generating a new product version. Overall, this indicates a shift of focus from source code to source models, not only in terms of the perceived origins of a software product, but specifically in terms of the resources that need to be managed. As a consequence, the common approach of agile development projects working on shared code repositories cannot be adopted immediately without certain drawbacks. Version control systems, like Git, are tools that track changes made to files, allowing users to collaborate on projects by working on the same files simultaneously by merging the respective changes and handling potential conflicts. Git also maintains a complete history of all changes made to the files in a project, allowing users to see who made specific changes and when they were made.

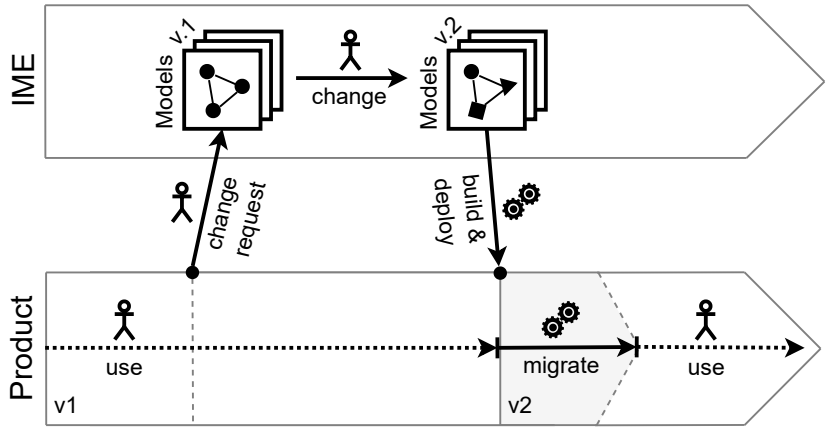

Figure 4.3: Product evolution via model evolution

Although Git can manage any type of files, it is widely used to manage source code because the mentioned features are specifically tailored towards text files. However, even if the models are stored in a textual format it is hard to read and especially difficult to understand differences in model versions or to merge conflicting versions of model files. This also makes it difficult to track changes to a model over time and to collaborate with others on the same model. Additionally, because graphical models can be complex and large, they can take up a lot of storage space and make it difficult to maintain a readable history of changes.

As a summary, with particular focus on graphical modeling, the challenges for IMEs in LDE ecosystems to establish such support are:

- **Collaboration:** Allow multiple users to work on the same project, and in particular on the same models, simultaneously. Provide visual presentations of model differences and functionality for automatically merging changes made by different contributors. Support conflict resolution or avoid conflicts completely.
- **Traceability:** Provide a way to keep track of the evolution of a project by maintaining a complete history of all changes made to the models in a project. Allow users to see who made specific changes and when they were made. This helps to understand how the project has changed over time and allows for better decision making as well as for identifying problems and their causes.

- **Reverting changes:** Allow users to roll back to previous versions of files, and models in particular, so that they can easily undo changes that introduced bugs or caused other problems.
- **Branching and merging:** Allow to work on multiple versions of models at the same time and to easily merge them together when they are ready. This supports experimenting with new features or bug fixes without affecting the main product version.
- **Sharing:** Provide an easy way to share models with others. This is especially useful for open-source projects where multiple people contribute to the same project.

There are specialized version control systems specifically designed for graphical models, but they are less common than general-purpose version control systems, like Git, as they are usually directly integrated with modeling environments or frameworks, and provide specific features for managing graphical models, such as visual presentations of model differences and merging capabilities. An example for such a built-in version control system is *Simulink* [34], a graphical environment for modeling, simulating and analyzing dynamic systems, where the user can save different versions of a model and compare them with each other. Another example is the modeling environment *Daffodil* [84], which has an integrated version control system for data flow diagrams.

However, in the context of LDE ecosystems an adequate generative approach is desired towards providing the appreciated features and benefits of version control systems, but on the model level.

4.1.2 Continuous Integration and Deployment

To automate the build and deployment tasks depicted in Figure 4.3 based on source models requires the used IME to support the effective creation of an appropriate CI/CD pipeline, either directly in the modeling environment or via the provision of all resources and artifacts required for executing a pipeline on a suitable CI/CD platform, like GitLab or GitHub. First and foremost this means providing a standalone code generator to be called and executed in an early step of the pipeline to perform the code generation task on the provided source models and produce the source code as well as other resources required for the subsequent build, integration and deployment steps (cf. section 2.2 on application generation).

As LDE aims at directly involving stakeholders into the development process, ideally, there would be no programming required to apply certain changes to a product. This in turn means the modelers totally rely on the features and capabilities of the used IME as well as on the consistency and quality of the generated artifacts used for building and deploying the product. As these artifacts, too, may not be within the modelers' area of expertise, whenever a particular stakeholder should be enabled to decide and control anything related to these build, integration and deployment steps, e.g. what is deployed on a particular environment, the IME must provide an appropriate language for these configurations that is tailored towards the mindset and capabilities of the respective modeler.

4.1.3 Data Migration

Virtually every software system operates on a specific data model consisting of well-defined data types and data fields. It is a conceptual representation that defines the structure, relationships and constraints of the data. It serves as a blueprint for how data should be organized and processed within a software system. Especially in regard to web applications, this data is often stored as key/value pairs in rows and columns of tables within relational databases following schemas that reflect the actual data model.

If a new product version is meant to replace a previous version that has been used in production (cf. Figure 4.3), particular attention needs to be paid to existing data that might need to be migrated to a new structure, i.e. transformed to be conform with a new data model. This can be a complex and error-prone task and may require significant testing and validation to ensure that the data is correctly migrated and still consistent and accurate.

This careful handling of production data applies independent of the actual type of product built with an IME and, in particular, if the product itself again is an IME. In this case, the main difference is the type of data that might require transformation, because the data processed by an IME are models and the meta-level changes are changes to the modeling languages. This aspect will be discussed in the upcoming section 4.2.

In general, data migration involves a number of challenges and pitfalls, including:

- **Data loss or corruption:** When the data model changes, it is important to ensure that existing data is not lost or corrupted in the process. This can be a particular concern when adding or removing fields, or when changing the structure of tables or other data objects.
- **Compatibility issues:** Changing a data model can also cause compatibility issues with existing systems and applications that rely on the old data structure. This can require significant effort and testing to ensure that these systems continue to function correctly with the new data model.
- **Performance impact:** Changes to a data model can also have an impact on the performance of the system, particularly when it comes to querying and indexing data. It's important to consider these performance implications and to test the system thoroughly to ensure that it can handle the expected load and usage patterns.
- **Complexity:** Data models can become complex as the number of entities and relationships increase. Understanding the dependencies and relationships between different entities and tables can be difficult. If not done correctly, data model changes can introduce bugs and errors that are difficult to detect and fix.
- **Lack of documentation:** Data models that lack documentation can make it difficult to understand the relationships and dependencies between different entities and tables, and can make it difficult to make changes to the data model without introducing errors.

For productive environments, here are several aspects to be considered in regards to efficient strategies for data migration, including data backup and restore, incremental migration vs. switchover, downtime minimization and many more. The best strategy depends on the specific requirements of the migration and the type, size, complexity, and criticality of the data being migrated. However, it is important to plan the migration process carefully, and to test the new system thoroughly before going live to minimize the risk of downtime. Consultation with the cloud provider and database experts as well as testing the migration scenarios before going live are recommended.

In the context of LDE, these considerations introduce challenges for the respective IMEs by means of providing appropriate support and guidance for the modelers at model design time. The consequences of changes, in particular with regards to potential need for data migration, should be clearly presented to the modelers for them to decide whether to apply these changes or work on an alternative solution. If data migration is inevitable, ideally, the IME would provide an automated or semi-automated way to speed up the migration process, which means that certain steps of the migration process are done by adequate tools. But still, the process might involve human intervention by means of carefully planning and maybe testing the migration before automation to ensure data integrity and consistency.

4.2 Language Evolution *(step-up)*

During product evolution, the available language features of an IME eventually are not sufficient to realize the functionality demanded by a change request for a specific product. In this case, a change request can be formulated towards the IME on a higher meta-level to demand new or improved language features. Hence, this approach can be considered a *step-up* in the meta-level hierarchy as depicted in Figure 4.4.

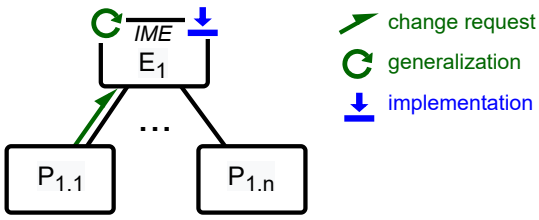

Figure 4.4: Step up in the meta-level hierarchy

In general, if the available language features of an IME are not sufficient the developers of a product (i.e. the users of the IME) have the following options:

- **Provide a custom implementation.** The developers of the product may create a custom-build component tailored towards the specific purpose without using the available modeling language features. However, this approach is recommended only if the IME natively supports service integration, i.e. the integration of custom-build components at model design time. Otherwise, it is hard to maintain consistency at design time and the use of custom-build components without IME support must be regarded as a workaround.
- **Escalate the change request.** If the available language features are insufficient, the product developers may formulate a feature request towards the used IME explaining the specific use case and demanding the additional functionality.
- **Reject the original change request.** If the requested functionality is not worth the effort of a custom implementation nor of escalating the change request towards the used IME, the original change request for the product may be rejected or postponed to a later date. However, in real-life scenarios this hardly ever is considered a suitable option as mostly there are substantial reasons for such change requests that cannot just be omitted.

In general, the escalation of the change request is the preferred way of dealing with this kind of issue. Because, even if the IME supports the integration of custom components, the formulation and publication of any improvement potential supports the continuous optimization process of the IME and hence of the complete LDE ecosystem. Custom implementations can be seen as intermediate solution or fallback if the escalated change request is rejected and the requested language features will not be implemented.

Real-World Example

The EquinOCS system comprises a user interface for the proceedings production, i.e. the arrangement of papers into topical parts of a proceedings book. A recurring task of the EquinOCS users is moving papers within and between topical parts to change their order.

During the project, eventually, a particular new requirement for greater user convenience has been introduced, that papers should be movable via drag-and-drop. The EquinOCS user interface, and the graphical representation of lists in particular, is defined with table components of DIME's GUI modeling language and at the time when this drag-and-drop requirement came up, these table components lacked drag-and-drop support for both resorting entries within the same table as well as moving entries between different tables.

Fortunately, following a service-oriented paradigm, DIME supports the use of product-specific components, i.e. custom components to be integrated into the GUI models. However, the realization of such native components requires programming expertise and thorough knowledge not only of the frameworks used for the frontend application (i.e. AngularDart, HTML and CSS) but also about DIME's code generator as these native components need to integrate seamlessly into the generated code. In a nutshell, it requires more advanced skills than the typical DIME user brings in and can be seen as a costly fallback to compensate missing language features.

In this concrete EquinOCS example, a custom table component providing the required drag-and-drop behavior would need to be implemented to replace the built-in tables of DIME's GUI language. The latter fact even adds to the already very high costs and complexity of this task because the custom table would need to support all the features that built-in tables provide in order to constitute a sufficient replacement. Hence, an extension of the GUI language features (i.e. the built-in tables) was preferred.

In contrast to non-IME products, any changes that affect the IME itself, like changes to the modeling languages, may also affect the products that have been built with it. Therefore, from a meta-level perspective, such changes are considered as *vertical changes* because they affect multiple meta levels in the meta-level hierarchy.

Generalization on the Meta Level

If the product developers formulate a feature request towards the used IME they need to explain their specific use case and may elaborate on the generalization potential. Preferably, they team up with the IME developers to discuss possible solutions in a joint and coordinated manner.

In general, every feature request regarding an IME has to be evaluated concerning its generalization potential. This means the IME developers need to evaluate whether the requested language feature may be useful for a variety of both already existing products as well as future products to be eventually built with the IME. It should be avoided to inflate the modeling environment with features that are mere product-specific or can only be used in very narrow or complex scenarios. As modeling languages are designed to provide only a restricted set of language features to strive for efficiency and simplicity, new language features need to be effective and comprehensible to maintain the ease of use and make them available to a multitude of users.

If the feature request is accepted and new language features are implemented and eventually released, the product developers can adopt the new version of the IME and realize the originally requested product feature with it.

If the feature request is not accepted, however, the product developers need to implement the required product feature by means of custom components. This would also be necessary if the new product feature is required immediately and the product developers cannot wait for the IME to eventually evolve. However, these custom components may later be used as basis for

the required generalization on the meta level. It may be seen as reference implementation for the new IME feature to be developed and as such comprise the targeted requirements to be met. Hence, even if the product developers realize the required functionality by means of custom components, it might be useful to formulate a change request for the IME to promote its continuous improvement and eventually adopt new language features in the future. However, at the end It should be possible to achieve a similar outcome by applying the new IME features as has been achieved with the custom components to be replaced.

Real-World Example

The EquinOCS developers filed a request for a drag-and-drop feature of the table components of DIME's GUI language, which has eventually been accepted. Its development was supported by means of formulating specific requirements originating from the planned application.

The drag-and-drop feature could be generalized on the meta level successfully, as not only EquinOCS but many other DIME products (i.e. web applications) benefit from such a drag-and-drop feature for tables. Even in EquinOCS, once available, this new feature has eventually been adopted for other tables than originally requested for.

4.3 Adoption of Language Changes *(step-down)*

If the feature request is accepted and new language features are implemented and eventually released, the product developers can adopt the new version of the IME and realize the originally requested product feature with it.

Once the new feature has been implemented and integrated into the models the new version of the product can be generated, built and deployed via the CI/CD pipeline.

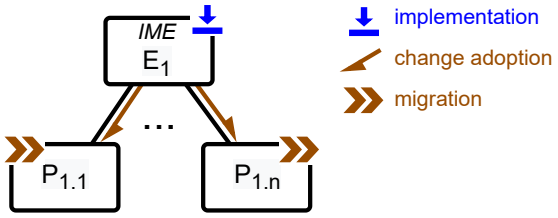

Figure 4.5: Step down in the meta-level hierarchy

While in general evolution of a software product means producing new versions over time by adding or removing features as well as adapting existing ones, in the context of an IME this might very well involve changing the corresponding modeling languages. Hence, in contrast to non-IME products, such changes not only affect the IME itself in a *tree-down* fashion this might affect all products that have been built with the IME. Therefore, such changes are considered as *vertical changes* because they affect multiple meta levels.

Real-World Example

The implementation of the requested drag-and-drop support required syntactic as well as semantic changes of the table components of DIME's GUI language.

Syntactically, the two requirements of reordering table entries and moving entries between different tables have been met by adding a configuration option for table components in DIME's GUI language to specify so-called *drop targets*. i.e. those tables where entries can be moved to. If this list of drop targets contains the respective table component itself this is interpreted as support for reordering of its entries.

Semantically, the new *drop target* configuration option for table components in DIME's GUI language required changes in DIME's code generator to introduce the required drag-and-drop functionality for tables. This comprised both the drag-and-drop support in the user interface, i.e. the frontend application, as well as the necessary functionality in the backend application to deal with re-arranging elements in a list as well as moving elements from one list to a specific location in another list.

Model Transformation

From an abstract perspective, data transformation may become necessary whenever the meta-model changes. This holds independent of the actual type of data. In particular, the statements 4.1.3 about data transformation particularly hold true if the considered software application is an IME where the actual data created and processed by this IME comprises models. Each of these models conforms to a specific modeling language (i.e. the meta-model) where the defined model element types are the actual data types.

As a result, the general approach of data transformation is very well applicable independent of whether the considered software application is the product of an IME or the IME itself. More vividly, any change to a node in the meta-level hierarchy (cf. Figure 3.14) has an eventual effect on its child nodes concerning the need of data transformation. If the affected node is an IME, the term *model transformation* is used synonymous to *data transformation*.

In general, model migration due to language changes can present several challenges, including:

- **Language Compatibility:** Ensuring that the new language is backwards-compatible with the previous version, so that existing models can be easily migrated. If the new language is not compatible with the existing models, it may be difficult to migrate the models without significant modifications.
- **Learning Curve:** Addressing the challenge of retraining the modelers, e.g. domain experts, who are familiar with the previous version of the language, to become proficient in the new version.
- **Validation and Testing:** Ensuring that the new language and migrated models are thoroughly validated and tested to guarantee that they are functioning correctly and are of sufficient quality for deployment.
- **Expressiveness and Accuracy:** Making sure that the new language is as expressive and accurate or better than the previous version.
- **Simplicity:** Ensuring that the new version is as easy to use or better than the previous version.

The model transformation process may prove to be very cumbersome, especially if the number of models is very large. Hence, ideally, the model transformation is carried out automatically without any user interaction. The actual logic which applies certain transformation rules on the existing models may either be programmed by developers or, ideally, generated from the differences of specific meta-model versions. However, there is existing and ongoing research in the field of MDE on effectively generating migration logic from the differences of two specific

meta-models to transform an old version of a model to a new version [85, 86]. The overall goal is to provide a migrator with every new version of a modeling language which automatically transforms existing models that have been created with a previous version to be conform with the new one.

Quality Assurance

Once a new language feature has been implemented the target code of the new version of the IME can be generated, built and deployed via the CI/CD pipeline. Users can then grab this new version and, after eventually having solved syntactical issues with existing models, use it to build new features into their products. But even if the syntax is intact and the models did not break, the semantics, i.e. the code generator, may have changed independently and produce code that causes a different structure or behavior of the products generated with the IME. This might happen even without any changes to the product models at all and hence, initially, go unnoticed at design time.

Real-World Example

The introduced *drop targets* language feature of Dime's GUI language is a pure language extension, i.e. a new feature is introduced while existing features remain untouched. Existing models, however, lack this new configuration option because it did not exist when they were created.

In this specific case, breaking the existing models could be avoided by dealing with the absence of this configuration option gracefully, e.g. by falling back on a default value. But when doing so it had to be made sure that using this default value results in the original behavior. As an example, if drag-and-drop for tables would have been enabled per default, all products that have been built with Dime would suddenly behave differently when using the new Dime version to generate these products' code.

While anomalies due to semantic changes might be avoidable in specific use cases (e.g. by introducing appropriate defaults that match the prior semantics), in general, however, this is a hard task as sometimes the effects can only be observed at product runtime when the generated code is executed. Tackling such issues introduces new challenges regarding effective and efficient quality assurance, to be discussed in section 4.5.1.

4.4 Multi-Level Language Evolution *(path-up)*

In LDE ecosystems it very well happens that feature requests affect meta levels other than the one directly above. Instead, they may indirectly escalate further up the meta-level hierarchy. As an example, a feature request originating from a non-IME product on the lowest level of the meta-level hierarchy may escalate up to an IME two meta levels higher. Generally spoken, this escalation chain builds a *change request path* that crosses multiple meta-level boundaries in a *path-up* manner.

Real-World Example

The discussed new *drop targets* feature of Dime's GUI language has been realized by

linking the respective table component with other table components to express that at runtime of the built web application the entries of one table can be moved to the other tables via drag-and-drop.

For this, on language level, the table components received a new complex attribute to hold a list of references on other table components. This is only possible if on the meta level the language workbench, i.e. Cinco in this case, supports the definition of such complex attributes in the first place. If not, the Dime developers would have filed a feature request which, just like the step from EquinOCS up to Dime on the next meta level, this time would affect the language workbench Cinco on the next meta level above Dime.

In turn, the change request path that targets Cinco has been formulated by Dime developers but has its origin in new requirements formulated for a Dime product, i.e. EquinOCS two meta levels deeper.

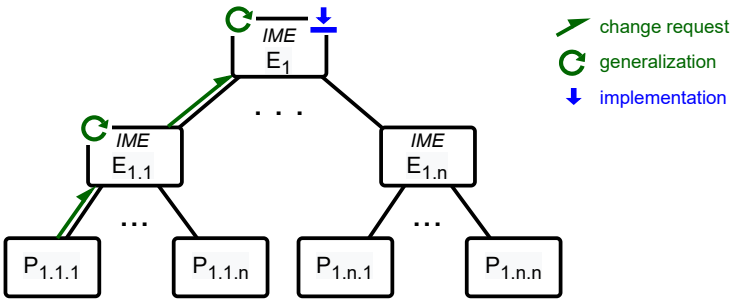

Figure 4.6: Path-up effect in the meta-level hierarchy

In general, the path-up propagation of changes happens step-wise, by means of filing a change request, generalizing the requirements and eventually filing another change request towards the next meta-level. While LDE and the involved IME projects should support this step-wise propagation, it is typically not necessary, or even counter-intuitive, to concern about development tasks higher than the next meta-level.

Real-World Example

It is a coincident that the EquinOCS development team overlaps with Dime's development team, and even with Cinco's development team. In general, the development of EquinOCS with Dime has nothing to do with the development of Dime with Cinco. In particular, EquinOCS developers would not need to know about the existence of Cinco at all.

4.5 Multi-Level Adoption of Language Changes *(tree down)*

While the general management processes regarding feature requests, product evolution and continuous integration are basically the same for different *types* of software products (desktop/web application vs. IME), a particular specialty arises from the meta-level-dependencies between these products (cf. Figure 3.14) which causes the aforementioned PUTD effects.

If a feature request at the end of a change request path is accepted and the requested feature is implemented, the new version of the respective IME can be generated, built and delivered,

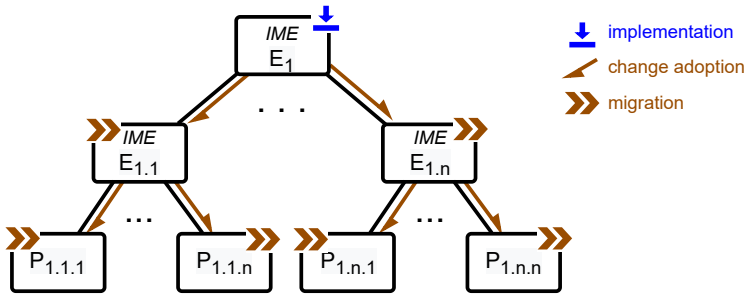

Figure 4.7: Tree-down effect in the meta-level hierarchy

e.g. via a CI/CD pipeline. Users can grab this new version to build new features into their mIDEs. This is where, again, the aspect of migration comes into play. But in this scenario, the *tree-down* effect is even bigger because changes of the language workbench may have effect on *all* IMEs built with it. And in the course of the necessary migration of these IMEs as well as the subsequent migration of *all* products that have been built with any of these IMEs, the original changes to the language workbench may affect even *all* of these products, although in an indirect manner.

Real-World Example

If Cinco changes, users can grab its new version to build a new version of Dime or other similar IMEs with it. However, as the meta-models may have changed, the migration of Dime and these other IMEs may be necessary to adapt to these changes. Hence, in this scenario, the *tree-down* effect means that changes of Cinco affect *all* IMEs built with it and may even propagate further. Because these IMEs, due to the necessary migration, need to be changed, too, this effect may propagate to *all* of their products as well, including EquinOCS as a Dime product in this concrete example.

Small changes on the meta-level, however, may have huge effects on lower meta levels, which even more stresses that the successful management of these effects needs appropriate and effective quality assurance processes, to be discussed in the next section.

4.5.1 Multi-Level Quality Assurance

In contrast to the evolution of non-IME products (i.e. the leaves in the meta-level hierarchy), the evolution of IMEs concern the involved modeling languages, and hence completely different meta levels.

In an LDE ecosystem, any change to an IME may cause effects that propagate downwards in a *tree-down* manner. The changes to *one* IME node in the meta-level hierarchy may affect *all* descendant nodes, although in an indirect manner (cf. Fig. 4.7). This is because with the changes of an IME all of its products may need to be migrated to assert conformance with the new meta-model. If the products are IMEs themselves, the changes carried out by migration may then have effect on the products of those IMEs so that the overall effect may propagate to *all* of their products as well.

The successful management of this *tree-down* effect requires appropriate and effective quality control that surpasses classical approaches as it must consider aspects across meta-level

boundaries. While syntactical changes to modeling languages will eventually lead to non-conform models, semantic changes, i.e. changes to the generator consuming these models, can lead to structural or behavioral differences in the generated products two or more meta-levels deeper. Hence, observing semantic changes eventually requires building or even executing the products of an IME. This stresses why in the context of evolution of any IME in an LDE ecosystem, quality control should be an eminent part of the release process.

Real-World Example

In terms of quality control, when assessing the effect of changes to Cinco, we cannot only focus on Cinco itself and the products built with it (like Dime), because the potentially unintended effects of a particular change might only be recognizable in tests applied on products multiple levels deeper in the meta-level hierarchy. In particular, as the Dime generator uses the model API generated by Cinco, any error that may have been introduced by changes may lead to unintended behavior of the running EquinOCS system, even without getting noticed on Dime level.

As already motivated, the assessment of test results by means of *path-up* feedback loops to be established is essential. Building support for *tree-down* effects into meta-model-driven development approaches is equally challenging. Here, too, the respective projects are located on completely different levels of the meta-level hierarchy and involve completely different stakeholders as well as separate development teams working on independent code repositories. On the other hand, automation, again, is a key enabler in this context due to the potentially huge number of products down the tree. The huge advantages in terms of quality control one would gain from a successful implementation of this approach motivates research on how to effectively build this into a meta-model-driven development stack.

Real-World Example

For any new version of Dime one automatically could generate all Dime products, i.e. web applications, then trigger the dynamic tests built for these products and evaluate the test results with focus on changed and unintended behavior. This may require comparing these test results with those generated in preceding steps, just like done in regression testing. However, this approach would mean a very powerful way to iteratively yet naturally improve Dime itself.

Altogether, in the context of vertical changes in an LDE ecosystem, Quality Assurance (QA) becomes an eminent part of the release process of all IMEs. It involves both, validation rules (i.e. static analysis) on the meta level as well as runtime testing on the product level. However, QA is a challenging task in this context because of the potentially different stakeholders involved on the various meta levels.

As discussed in [AP5], a very promising approach to support multi-level quality assurance in LDE ecosystems is *lifelong learning* [87] as a form of continuous quality control. A suitable lifelong learning cycle, as depicted in Figure 4.8, has been conceptualized for the IME-based generation of web applications [88] based on the experiences with Dime and EquinOCS in operative practice. It comprises applying active automata learning on the web application and verifying certain behavioral properties via model checking as well as monitoring its runtime behavior to report observed errors to the modeler [89].

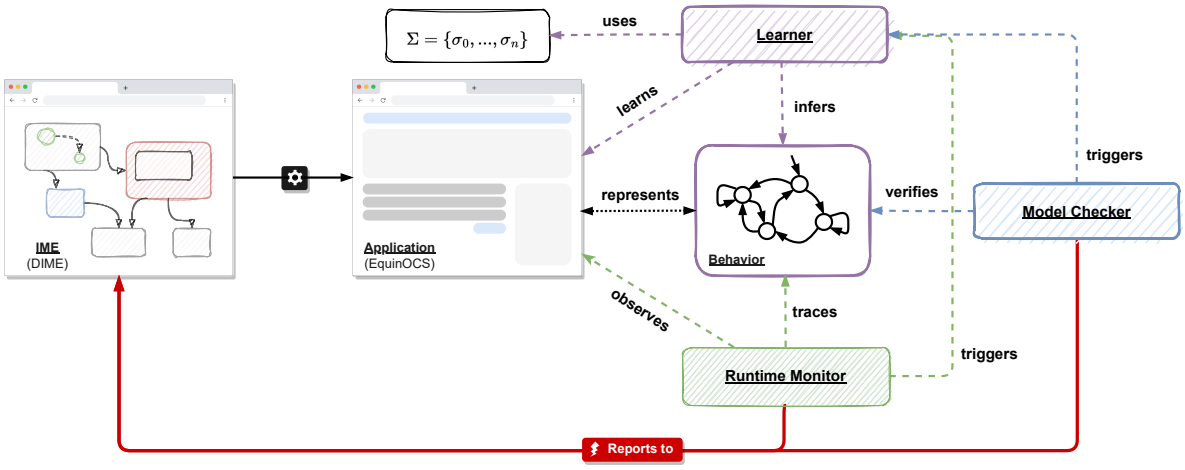

Figure 4.8: Lifelong learning of web applications (from [AP5])

4.5.2 Quality Feedback Loop

The dynamic validation of IME products, such as web applications, cannot be scaled horizontally as it is product-specific and requires custom test cases for each product. However, the process of creating these test cases can have a positive indirect scaling effect. Given a specific IME, the code generator for this IME's products works on the meta-models of the IME's modeling languages and thus is developed alongside the IME itself. To ensure the generator produces accurate results, it must process the models designed with the IME to generate and run code of the IME products.

The key point is that testing *any* IME product provides feedback on the overall quality of the generator, which is developed two meta-levels higher. This observation can be considered a *path-up* effect in the meta-level hierarchy. If any errors are discovered during testing, they can be reported upward, leading to a change request for the generator's development and eventually to quality improvement. The same goes for change requests originating from IME products. If from the viewpoint of the product's development team, a desired outcome cannot be achieved because the IME's language capabilities are insufficient, a respective feature request can be passed to the next meta-level and may eventually be considered for further language enhancements.

Measuring the impact of releases on the stability of affected systems could be a means to assist developers in building more reliable systems [55]. By accessing the valuable information from test results of IME products in a structured way, we could naturally improve the quality control of an IME's code generator, as more test cases are implemented and reported upwards the meta-level hierarchy for various IME products. Essentially, this implies that instead of solely building test applications for an IME's generator, we could also utilize the tests developed for productive products to verify and enhance the overall quality of the product generator. However, accessing and evaluating this test data is difficult because the corresponding projects are not linked directly in the meta-level hierarchy. Typically, these projects involve distinct DevOps teams working on separate code repositories at different meta-levels and hence must not even know from each other. Secondly, automation plays a crucial role in this scenario due to the potentially large number of IME products involved, each associated with a likely significant amount of test cases.

The research on how to exploit these effects and dynamics that transcend meta-level boundaries is still at the beginning and effective methods of integrating generic feedback loops for enhanced

quality control into each layer still need to be explored. This might take advantage of the recently formulated concept regarding a lingualization strategy [90] for knowledge sharing based on the DevOps lifecycle, which might be applied on PUTD dynamics of LDE ecosystems. Part of this concept is the introduction of so-called Meta DevOps teams that discover, store and distribute knowledge by means of developing and distributing appropriate PSLs.

5 Related Work

The foundation of the LDE approach lies in the combination of two aspects. Conceptually, its goal is to allow all involved stakeholders to collaborate on software development without coding by providing them tailored IMEs comprising (graphical) languages, ideally based on those they are already familiar with. Technologically, this requires suitable meta tooling support to efficiently create highly specialized low-code development environments that are capable of generating complete applications from the created models. To the best of my knowledge, these two aspects, tailored solutions for stakeholders and full application generation, are rarely addressed in combination.

Furthermore, the core topic of this dissertation is less a matter of *how* to develop languages, language workbenches or modeling environments, but focuses on the consequences from rigorously doing so wherever it seems appropriate. Basically, LDE can be implemented using any language workbench that supports the types of languages (such as graphical or textual) needed for the addressed user groups and provides comprehensive support for constructing a consistent model structure from multiple interconnected models. The focus of this dissertation lies on dynamics and challenges emerging from using meta-modeling tools to apply LDE in ambitious software development projects, including the development projects of IMEs themselves to eventually construct LDE ecosystems.

While there exists quite some related work with regards to languages, language workbenches, modeling environments, code generation, low-code platforms and others, the interplay and dynamics of respective tools across multiple meta-levels, to the best of my knowledge, are barely researched and discussed. Therefore, the discussion of related work is divided into different sections that address these related aspects in a separate manner.

Meta-Modeling Frameworks

Meta-modeling frameworks, like the EMF [61, 62] and the MetaObject Facility (MOF) [91], on their own do not provide the full range of functionality typically expected from a language workbench, such as syntax highlighting, parsing, and error reporting. However, they can be used in conjunction with other tools to effectively build custom DSLs.

MOF is a standard for defining the structure of meta-models, and it provides a way of representing and manipulating models in a common, machine-readable format. Ecore is a subset of the MOF meta-model and is specifically designed for use in the Eclipse modeling ecosystem [92] and central to the development of applications using EMF. Ecore is the meta-model at the heart of EMF for defining the structure of models and the relationships between different model elements.

EMF is a framework for creating, storing, and manipulating models based on a way of representing models in a common, machine-readable format. EMF provides a range of features

for building tools and applications based on meta-models defined in Ecore, including model transformations and code generation. It can also be used in conjunction with other tools, such as Xtext, to provide a complete solution for building custom textual languages. Here, Xtext provides the language-specific functionality, such as syntax highlighting and parsing, while EMF provides the infrastructure for storing and manipulating models.

In the Eclipse context, Acceleo [93] is an open-source code generation framework that is used to generate code from models written in a variety of modeling languages, including UML and Ecore. Acceleo provides a way of defining code generation templates based on relationships between models and code and generating consistent code from the models automatically. It supports a range of features for customizing the code generation process, such as defining custom generation rules, using conditional statements, and invoking custom code. It also provides a range of tools for working with the generated code, such as debugging and testing.

Language Workbenches

Language workbenches provide an integrated environment for the creation and maintenance of DSLs for arbitrary domains. They can automate many of the tasks involved in building a language, such as parsing, syntax highlighting and code generation, e.g. for the respective (graphical) editors and other utility tools that make it easier to adopt and work with the DSL.

There exist some renowned language workbenches, including Meta Programming System (MPS), Xtext, Spoofax, Graphical Modeling Environment (GME), Rascal, MetaEdit+ and KIELER, which provide tools for defining custom syntaxes, generating parsers and editors for the new languages, and integrating them into existing development environments, such as Eclipse [94] or Visual Studio [95], making it easier for developers to use these languages in their development workflows. Also of notable mention in this context are Pounamu/Marama [96–98], Sirius [99, 100], and DeVIL [101, 102] for generating graphical modeling tools from meta-model specifications.

Xtext [60, 103] is an open-source toolset for building custom acpDSLs and IDEs. Xtext provides a range of features for defining the syntax and semantics of textual languages, generating parsers and editors, and integrating the new languages into existing development environments. With Xtext, developers can define the syntax and grammar of a new language using a simple and concise grammar syntax, and Xtext generates the necessary code and tools for parsing, validating, and editing code written in the defined language. Xtext also provides features for customizing the user experience, such as syntax highlighting, code completion, and error reporting. Overall, Xtext is a powerful and flexible tool for building custom domain-specific languages and it is well-suited for use in a wide range of development scenarios, from small internal projects to large, complex systems.

Spoofax [104, 105] is an open-source language workbench for developing custom domain-specific languages. Spoofax is based on the Stratego transformation language [106] and it provides a declarative, composable and reusable way of building custom languages. It supports the development of both textual and graphical languages and provides a range of features for customizing the user experience, such as syntax highlighting, code completion and error reporting.

The GME [107–109] is a software framework for building custom modeling languages and environments for use in model-driven development. GME provides a way of defining the syntax and semantics of both graphical and textual languages and generating custom editors and visualizations for working with models written in those languages. It supports the development of both graphical and textual languages and provides a range of features for customizing the user experience, such as syntax highlighting, code completion, and error reporting.

Rascal [110–112] considers itself a meta-programming language [113]. Meta-programming is a technique in which a program generates or modifies other programs. With Rascal, developers can create tools for analyzing, transforming and manipulating code, such as code refactoring tools, code generators and code quality checkers. It supports the development of both textual and graphical languages and provides a range of features for customizing the user experience, such as syntax highlighting, code completion, and error reporting.

MetaEdit+ [114, 115] is a commercial tool for defining and using domain-specific languages and modeling environments. MetaEdit+ supports the definition of both textual and graphical languages and provides a range of tools for customizing the user experience, such as syntax highlighting, code completion, and error reporting. In addition to its language definition features, MetaEdit+ also provides a range of tools for working with models, including support for model transformations and model validation. MetaEdit+ is widely used in a variety of domains, including software engineering, telecommunications, finance, and more. It is well-suited for use in a wide range of development scenarios, from small internal projects to large, complex systems. It is often referred to as the reference implementation for Domain-Specific Modeling (DSM) [47]

KIELER [116, 117] is an open-source framework for developing graphical modeling environments and integrating them into existing development tools, such as Eclipse. It is by now generalized as the Eclipse incubation project Eclipse Layout Kernel (ELK) [118]. KIELER provides a range of libraries for transforming and querying models as well as for visualizing and laying out models. It is widely used in a variety of domains, including software engineering, telecommunications, and more. Furthermore, it provides means to automatically generate domain-specific graphical views for textual DSLs realized in the Eclipse modeling context. However, its primary goal is to provide application experts with graphical notations for illustrative purposes, e.g. to better communicate with non-programmers, while creating the actual (textual) models still requires programmers or highly technically skilled domain experts.

MPS [119, 120] is a language workbench developed by JetBrains [121], which provides a platform for creating custom programming languages, language extensions and DSLs. It provides a visual language definition and supports the full language development life cycle, including parsing, type checking, code generation, and more. Due to the comprehensive and powerful approach of MPS to language specialization by means of projectional editing [122], and the meanwhile enhanced support for graphical editors, MPS can best be compared to Cinco [22]. As the MPS project shares the assessment that a single DSL per development environment may not be sufficient, it supports true language modularity, just like Cinco. In fact, the service-oriented language extension and DSL-based refinement using the Cinco framework (cf. [23]) can be regarded as light-weight language specialization for building component libraries, reminiscent of language concepts used in MPS, which are the building blocks of custom DSLs in MPS. While developing languages with MPS is very complex due to an extremely steep learning curve, the objective of developing Cinco from the outset was simplicity to overcome the major obstacle often faced in language-driven approaches, that the effort required for developing an appropriate IME exceeds the benefits, especially if the created modeling solution is highly specialized as in the context of LDE. The experience from practice gained from various industry projects shows that this is a main reason for why the resulting solution often tends to be rather general-purpose than highly domain-specific. To address this, simplicity was a key focus and prioritized over the generality offered by other language workbenches during the development of Cinco. This is reflected by Cinco's support for code generation by means of providing a dedicated model API specifically generated for each different Cinco product that is thus more intuitive and powerful than the underlying generic Ecore API. In contrast, cost-effectively building good LDE solutions with MPS is probably reserved for a few experts who are really well-versed with all the intricate details of

the framework.

Along with the general trend towards developing software as web applications, in recent years, there has been a trend towards shifting development environments to the web. This is because web applications can be accessed from any device with an internet connection, making them more convenient, flexible and accessible for users as they can be updated more easily and rapidly, without requiring users to download and install new software. In this course, the graphical modeling features of Cinco have been ported to the web with the Pyro project [66] so that users can easily use the software hosted in a cloud environment. Currently, the feature set is yet slightly restricted as some of the secondary Cinco features developed for the Eclipse RCP [71] have not been migrated yet. As can be experience with the data-flow modeling tool Pyrus [73] , the editors provided by Pyro support collaborative modeling, similar to real-time document editing features of well-established cloud-based solutions, like Google Docs [123] or Microsoft 365 [124]. This means that multiple users can access and edit the same model at the same time, without needing to worry about overwriting each other's changes.

In a similar manner, the ROCCO tool [125] based on Eclipse EuGENia [126] makes graphical modeling languages accessible in a web-based development platform [127]. While the EuGENia framework allows users to define their own modeling languages and then generate code from those models, ROCCO extends the EuGENia framework to generate graphical web editors for these languages. The corresponding migration to a web-based platform is done by generating diagram models for evaluation in the Psi Engine [127], a web-based low-code environment. Compared to Pyro, the modeling environment generated by ROCCO to date does not offer collaborative editing of the diagram models.

Low-Code Development

Low-code development describes a type of software development that enables faster and more efficient application development by reducing the amount of manual coding required, thus reducing the risk of errors and allowing organizations to respond more quickly to changing business requirements. The approach is particularly designed to empower non-technical users, such as domain experts, application experts, subject matter experts and other stakeholders to contribute directly to the development process without requiring extensive programming skills. Here, *directly* means that the artifacts constructed by these users are accurate and sufficient to make changes to applications without further human translation, e.g. by programming experts. The origin of the term "low-code" is not entirely clear, and it has been used in various contexts over the years. However, one of the earliest references to the term in its current meaning was made in a 2014 blog post by Forrester Research analysts [128].

Although DSLs can be involved in low-code development, they are not always necessary as low-code platforms typically include pre-built components and templates, a visual interface for designing and arranging application components via drag-and-drop, as well as tools for integrating with external data sources and systems. Some low-code platforms also include built-in functionality for testing, deployment, and ongoing maintenance.

Low-code approaches are continuously growing in popularity. It is hard to provide a comprehensive overview, as today there exist more than 300 vendors in the market and every low-code development platform is different [129]. Big software companies are heavily pushing their low-code platforms, like for instance Google's AppSheet [130] (a successor to App Maker) or Microsoft's Power Apps [131]. Both AppSheet and Power Apps enable users to connect to data sources, such as spreadsheets or other tabular data sources, allowing users to quickly bring data into the platform and use it to build custom applications. They both also provide tools for customizing the look and feel of the applications, as well as for integrating with

other systems and services. These platforms, in essence, can be considered manually built development solutions that resemble a spreadsheet-oriented mindset in their approach to application development. This approach is designed to be intuitive and accessible to those users who are familiar with spreadsheets and other data-focused tools. In turn, it provides simplicity only for those users with spreadsheet-oriented mindsets.

Although DSLs can be involved in low-code development, they are not always necessary as, typically, low-code development platforms provide pre-built components and visual interfaces for common tasks, without the need for users to create or understand a specific DSL. Some prominent platforms that use DSLs to provide a higher level of abstraction for non-technical users are OutSystems [132], Mendix [133], Appian [134], Bubble [135] and Lightning [136] from Salesforce [137]. Mendix, for example, uses a visual modeling language called "Domain Modeler" to define the data models and business logic of an application, while OutSystems uses a visual modeling language called "Service Studio" to define the user interface, business logic and data models of an application. Appian includes a rule engine and a workflow modeling language called "Process Modeler" that allows users to define complex workflows and decision logic without the need for traditional coding.

The mentioned platforms are particularly well-suited for developing custom business applications by defining workflows, forms and data-centric views, like dashboards. To extend the available components, developers can write custom code in programming languages specifically supported by the respective platform. For example, OutSystems provides the option of writing custom code in the dedicated "OutSystems Language", while Mendix supports multiple programming languages, including Java [43], Python [38] and Ruby [138]. Appian supports JavaScript [40], and Lightning provides the option of writing Apex, a proprietary programming language from Salesforce.

With regards to workflow modeling, these platforms focus on specifying workflows in a (graphical) flowgraph style using the (manually programmed) application-specific activity blocks. Similar to the spreadsheet-oriented mindset discussed above, the adequacy of solutions based on a process-oriented mindset strongly depends on a good fit with the mindset inherent in the target application. These approaches address situations where a process-oriented combination of predefined building blocks is adequate. Thus, it can best be compared to the approach taken by the jABC project [69], a predecessor of the Cinco workbench project, developed decades ago. In general, generating executable code from graphical process models has long been the underlying principle of process modeling tools, and in particular for the jABC framework, whose SLGs heavily inspired Dime's process models. In turn, one could argue that jABC and other early modeling frameworks supported the low-code approach long before the term was invented. However, just like the spreadsheet-centric or process-oriented approaches mentioned above, jABC was less flexible as it did not build on top of any language workbench. In general, it is noticeable that the modeling features of popular low-code platforms are provided as-is and meant to be adopted by users in terms of adapting their mindset to the one supported by the platform. With regard to jABC, this changed when the Cinco language workbench was created to provide a powerful yet flexible foundation for the development of Dime (comprising jABC's SLGs) and other IMEs. Because the approach tackling the mindset issue underlying LDE, i.e. how to best address and fit the mindset of the domain and target audience, is explicitly addressed by those low-code modeling approaches that rely on language workbenches [19], as they allow tailoring the language of their development environments towards skills and needs of the targeted domain experts. This way, they can flexibly play with a simplicity/generality trade-off. However, this flexibility comes at the price of substantial costs for meta-level development [22].

To summarize, the LDE approach supports the goal of low-code development, i.e. to empower

non-technical users, such as domain experts, application experts, subject matter experts and other stakeholders to contribute directly to the development process. However, LDE promotes a *simplicity-oriented language engineering* approach, i.e. by means of supporting different language purposes and user mindsets to engineer adequate languages accordingly. Language-Oriented Programming (LOP) [139, 140] is a software development paradigm similar to LDE that focuses on creating DSLs as an integral part of the development process. LOP takes the idea of using a language to describe a problem and extends it to include creating custom languages tailored to a specific problem domain, which can then be used by developers to work at a higher level of abstraction and solve that problem more naturally and efficiently. This involves not only writing code in these languages but also designing the language itself and its associated tools, such as compilers and interpreters. This may also involve the composition of multiple languages that work together to solve a specific problem, and thus LOP fosters a language-oriented approach to software development. LOP is used in a variety of applications, from writing web applications to modeling complex systems and simulations. Its use is becoming increasingly popular, especially in domains where software is used to solve complex problems, such as finance, scientific computing, and gaming. However, LOP primarily focuses on providing programmers with a more effective and efficient way to write software, as indicated by their motivation to *"provide software developers tools for formulating solutions in the languages of problem domains."* [140].

6 Conclusion

This dissertation is concerned with the evolution of LDE ecosystems and demonstrates the inherent PUTD dynamics between their meta-levels. These dynamics are discussed based on observations and findings from concrete development projects within the Cinco-based ecosystem and generalized towards generic LDE ecosystems in order to identify challenges and opportunities regarding efficient and effective tool support.

The intrinsic idea underlying the LDE approach is to understand all forms of evolution on various meta-levels as integral part of the development life cycle of a specific software product. This generalizes the common conception that a software system is continuously evolving to encompass the entire development and evolution scenario. All elements including the system itself, the infrastructure it operates on, the DSLs used for system modeling and the meta-metamodels for constructing these DSLs co-evolve concurrently to adapt to ever-changing requirements on the respective meta-levels. The effective handling of possible effects induced by this concurrent evolution requires an LDE ecosystem to be flexible and responsive but also demands the utilization of comprehensive methods and strategies to maintain consistency and coherence across different meta-levels.

My extensive research and development efforts on the tools and technologies at the heart of the presented Cinco-based ecosystem has significantly influenced and shaped not only the involved tools themselves but also the view of the big picture and the conceptual framework of the LDE approach on the whole. In particular, the identification and investigation of the PUTD effects crossing meta-level boundaries as described in this dissertation represent immediate results of my work on various projects at different meta-levels, which allowed for an analysis of these dynamics in the first place. In turn, these results would not have been possible without first building an appropriate ecosystem comprising tools powerful enough to generate software products suitable for productive use. Hence, working on specific meta-levels and at the same time establishing a holistic view on ecosystem evolution are mutually reinforcing tasks.

As a starting point, the development of the integrated modeling environment Dime based on meta-models created with the Cinco workbench has paved the way for generating complete, full-stack web applications solely from conceptional models. What began with some proof-of-concept products developed with Cinco and Dime has since successively evolved into an extensive ecosystem, thereby constantly incorporating lessons learned and experiences gained into its ongoing development. Finally, by applying the LDE approach in the course of my work with Dime on the EquinOCS project in cooperation with an industry partner I have ultimately proven the practical applicability of the approach and demonstrated its benefits, thereby providing validation and improving credibility of the overall LDE-related research.

The EquinOCS project demonstrated how the LDE approach performs in a real-world environment and helped identifying challenges and limitations from which improvements of the whole ecosystem could be derived accordingly. In total, my work on the EquinOCS project generated valuable feedback from the involved stakeholders and profound insights into the process of

building, deploying, operating and maintaining a productive web application in the cloud with LDE technology. I have then managed to successively adopt and generalize these insights (in a step-up manner, cf. section 4.2) towards generic web applications in order to formulate new requirements and drive the further development of Dime's code generator and modeling languages. Some lessons learned were even propagated upwards in the meta-level hierarchy (in a path-up manner, cf. section 4.4) and influenced the further development of Cinco's language features. Overall, this also meant experiencing and assessing the so-far unexplored ecosystem dynamics set out in this dissertation and developing solutions to effectively handle the induced effects.

The LDE approach already comes with technical challenges comprising the effective and efficient definition of modeling languages, their provisioning by means of building corresponding IMEs and the integration of languages (and their IMEs) into consistent multi-language IMEs. To solve these challenges, various approaches have been discussed comprising meta-level modularity supporting the reuse of language features and approaches for shallow versus deep integration of languages for efficiently building multi-language IMEs. Although from a meta-level perspective, these topics seem to address foremost local aspects of an LDE ecosystem the introduction of a new IME to an existing ecosystem already means exposing it to the prevailing PUTD dynamics crossing meta-levels. To handle the induced effects, LDE needs integrated support to guide involved users and stakeholders. In this context, this dissertation addressed corresponding requirements and outlined the need for enhanced tool support.

7 Future Work

The utilization of DSLs, and in particular graphical notation, results in a closer integration of domain experts into the system development process due to its innate intuitiveness and ability to facilitate the desired mindset and objectives and thus effectively bridge the communication gap between domain experts and technical teams. Graphical modeling languages provide a visual representation that facilitates a more intuitive and accessible way to represent complex systems and processes, making it easier for non-technical stakeholders to understand and provide input which ultimately leads to better alignment between them and more successful development outcomes. Additionally, these languages often have a lower barrier to entry, as they rely on graphical symbols and diagrams rather than code, reducing the overall learning curve for domain experts. And as organizations seek to increase collaboration and integration between domain experts and technical teams, advanced software development methodologies, such as Agile and DevOps, emphasize the importance of collaboration and integration across teams and often provide guidelines and best practices for achieving these goals. Hence, It is anticipated that the number of languages utilized in individual projects will experience significant growth. The LDE approach generally supports this welcome development. Examples can be found in various application domains in terms of dedicated (graphical) languages, like network layouts, workflow graphs, or piping and instrumentation diagrams but also business models and other strategic notions, as well as spreadsheets or data flow graphs for data analytics. In general, illustrative charts and diagrams used by domain experts for the documentation of specific relationships or factual connections are a good starting point for creating efficient PSLs aligned with the mindsets of the respective stakeholders. However, as thereby some so-far clear boundaries become increasingly blurred some additional care is needed to manage associated effects.

Blurring Line between Meta-Levels

The meta-level dependencies of an LDE ecosystem have been introduced as a tree structure that represents a meta-level hierarchy. However, with the advancing developments of the LDE concept and lessons learned over years of use in academia and industrial practice, a slow but steady shift of focus can be observed from separate, distinct IMEs to shared modeling platforms that provide access to various modeling languages for different purposes. In this context, modeling languages and meta-models still exist and meta-level dependencies need to be managed but the concept of meta-level hierarchies needs to focus less on IMEs as complete, standalone tools but rather understand them as dynamic modeling environments that, for example, can be set up on demand for a particular project. Still, assembling such environments and tailoring them towards the mindsets and capabilities of involved stakeholders remains challenging, especially with regards to simplicity.

Another aspect that adds to the perceived blurring of boundaries is that, in general, an IME

may be used to extend itself in a bootstrapping fashion. This approach has initially been formulated for Cinco [21], in particular by elaborating that *with* Cinco graphical languages can be designed *for* Cinco by means of generating Cinco plugins providing the respective model editors to be integrated. However, the approach may as well be applied on similar meta-modeling environments in a straight-forward manner. following the same logic, an IME, like Cinco, could also be used to build another IME that is then used to extend the original one, hence inducing mutual dependencies.

These considerations regarding dynamic modeling environments can further be backed up by the observation that the LDE ecosystem dynamics investigated up to today are not complete. In fact, there are more complex scenarios with regards to ecosystem evolution that need to be considered, including:

- **Adding IMEs:** Users may use existing IMEs, like Cinco, to develop new IMEs to be added to the ecosystem. This may either be done by creating a completely new branch or by means of introducing a new layer in between two existing IMEs in the meta-level hierarchy. The latter approach can be used to, for example, create product-specific IMEs, i.e. modeling environments that are more tailored towards the specific needs of a particular project than the IME previously used. As an example, one could add a EquinOCS-specific IME, i.e. a Meta-EquinOCS environment, that provides modeling languages specifically tailored towards the development of EquinOCS. This new Meta-EquinOCS would most probably generate Dime models and then re-use the existing Dime generator to produce the same source code and resources as before.
- **Forking IMEs:** If the capabilities of the available IMEs are not sufficient for building a specific class of products, an appropriate IME project from the ecosystem could be forked to build a similar IME that is more tailored towards the respective use cases. As an example, one could fork the Dime project that is tailored towards generating single-page web applications in order to build a Dime variant that is more suited for generating static or progressive web applications.
- **Merging IMEs:** In opposite to forking, two sperate IMEs could be merged to form a single IME with joined functionality by means of a new multi-language environment. Existing products would need to be migrated to use the resulting IME.
- **Removing IMEs:** Existing IMEs may be discontinued and thus finally vanish from the ecosystem, for example if there are no particular products anymore to build with it. Otherwise, existing products would need to be migrated to a different IME in orderor its development needs to

There is need for comprehensive tools and methods that support the handling of these types of changes and derived effects. Such tools would aid in the efficient management of the ecosystem, promoting seamless evolution and reduce the risk of adverse impacts on specific products and the whole ecosystem.

In the context of constructing dynamic modeling environments and managing their evolution, these types of changes would need to be supported on language basis, i.e. by means of adding, removing, merging and splitting languages to the environment.

Blurring Line between Building and Using

With the progressive development of the LDE approach, the distinction between *building* and *using* applications increasingly becomes a mere matter of perspective. To explain and support this argumentation, in the following a comprehensive explanation is given from a

language-centric perspective. The provided examples are based on actual insights from the EquinOCS project but can easily be transferred to similar projects as the described dynamics can be considered common for current development scenarios.

EquinOCS supports a peer review process during which assigned reviewers upload their reports for submitted papers. For this, a simple report form is provided that comprises two dropdown-boxes 'Evaluation' and 'Confidence' as well as a 'Comment' text area for their argumentation. Figure 7.1 shows a simplified illustration of the form on the right side. Early in the project, this report form was static, i.e. the same form was used in all conferences managed by the system. The structure and appearance of the form were modeled with DIME's GUI language. And being a static feature, eventual changes to the form required changing its model and then re-generating and re-deploying a new application version. Then eventually the stakeholders requested to change the form and make its fields configurable in terms of allowing to enable or disable them dynamically at system runtime. This meant transforming the static report form into a configurable form and creating a second one with which the report form could be configured at runtime via the admin interface. Figure 7.1 shows both forms and their relations by means of the *IF*-edges, which express that the respective field in the report form is only then shown if the corresponding checkbox in its configuration is checked.

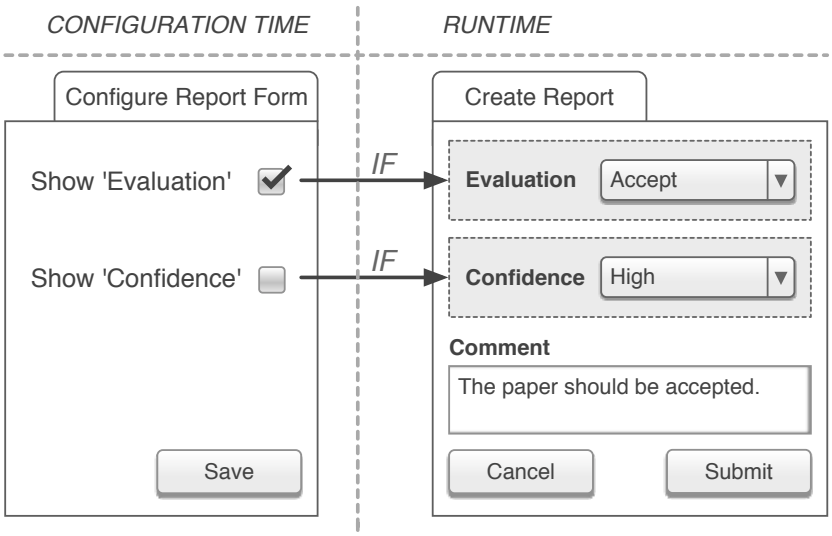

Figure 7.1: Report form configuration (from [AP6])

This example shows that switching the abstraction layer does not necessarily mean switching from system level to development level. On the other hand, by enabling the administrator and other stakeholders to configure this form at system runtime a small part of what previously has been *building* the application and thus reserved for the system developers has now become *using* the application. However, this effect can already be observed in classical software development where it is actually bound to the perception of how changes are realized, i.e. it depends on whether the affected feature can be configured at system runtime (configurable feature) or development effort is required to change its source code (static feature). Note how the sheer existence of differences between the change process of configurable features compared to that of static features causes different perceptions between *building* and *using* a system in this context. These differences include:

1. **Languages:** Simply setting in-system properties via a provided user interface to configure certain features at runtime is basically different from writing source code in a programming language.

2. **Users:** While runtime configuration can be done by an administrator or other stakeholders,

source code editing is typically carried out by programmers or other software developers.

3. **Platforms:** While the configuration of features can be done directly in the running web application, code editing is done, for example, in a separate IDE, which typically is a desktop application.

4. **Implementations:** The current feature configuration can be evaluated dynamically at system runtime while the source code created by programmers has to be compiled, built and packaged to finally get re-deployed and replace the previous application version.

By extending the report form example and continuously shifting control from programming level to system level it is shown that the LDE approach can overcome these differences simultaneously and thus completely blur the line between *building* and *using* a system by changing the overall perspective on system design. So far, the report form is configurable but which fields are available is still predefined. Adding additional fields would still require changes on programming level. To change this, the configuration form could be extended to allow stakeholders to create new report form fields at system runtime. This can be realized by introducing a manageable list of available report form fields with actions to add and remove fields on demand. Figure 7.2 shows the resulting forms and their relations by means of the *FOR*-edge, which expresses that the available fields in the report form depends on the defined entries in the configuration form.

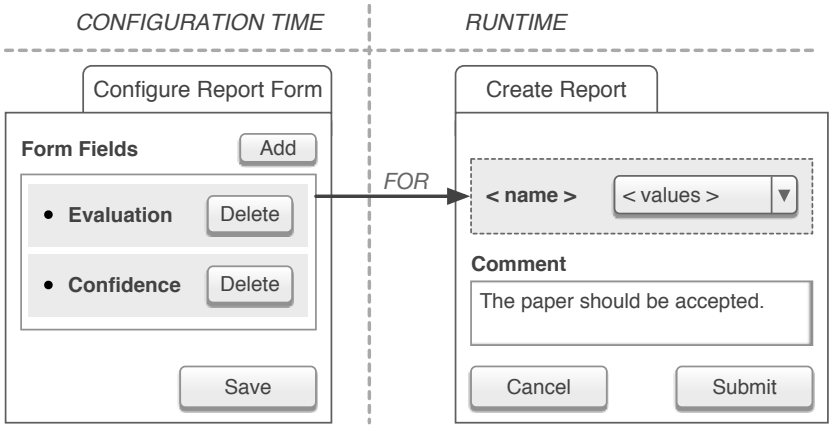

Figure 7.2: Report form field creation (from [AP6])

In comparison to the first version depicted in Figure 7.1 the feature set of the configuration form and hence the control over the report form has increased. It is another good example of how aspects that previously were associated with *building* the application are now associated with *using* the application. This increased control on the one hand means increased flexibility but on the other hand also increased complexity. The process of shifting control over the report form from development level to system level could be continued by adding editing functionality for texts, images, buttons, and other components until eventually administrators and other stakeholders would have full control over something that could be considered a complete user interface editor to change every little bit of the report form according to their needs. But in turn, the resulting complexity may not be manageable by this user group anymore. Because they now basically have access to instruments of a system developer and thus would need knowledge and skills that are traditionally associated with the programming domain to use such instruments effectively. They provide all flexibility but were not tailored towards the respective mindset and capabilities and thus cannot achieve an adequate level of simplicity.

Another interesting aspect in this context is the blurred line between configuration and system design. In the process of continuously shifting control it is hard to locate the exact point

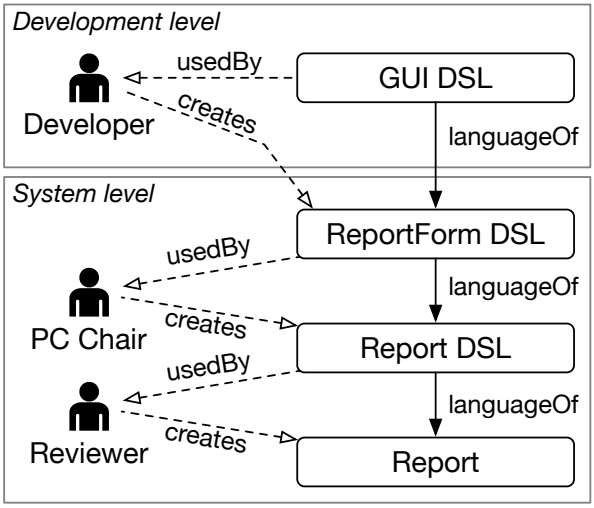

Figure 7.3: Top-down DSL creation (from [AP6])

where changing the report form would stop being mere configuration and actually become user interface design. But is this even necessary, or may both be considered being the same thing? A slightly different perspective on the purpose of DSLs may help shed light on this. While from a technical viewpoint models represent relevant aspects of a system and DSLs are considered necessary means used to create and edit these models, in the perception of the system user, the DSLs themselves provide an interface for directly changing a system's structure or behavior. Because, in this context, models can be considered a rather technical concept hidden from these users' view as they only exist by means of data in the memory or database. From this perspective, classical user interfaces, like the configuration forms mentioned above cannot be distinguished from graphical model editors as they both are means to create and edit data that may be a model for some kind of system structure or behavior. Consequently, any user interface that creates or manipulates some kind of data can be considered a DSL tailored to this specific use case, including the report form in the above example which, from this viewpoint, is used to create a model of a report that may later be used for some kind of analysis, reasoning or decision making. Following this line of thought, the consecutive changes to the configurability of report forms can be seen as the creation of specific DSLs tailored towards the respective requirements. This means step-wise transfer of control from development level to system level, as depicted in Figure 7.3 that shows the different DSLs involved in the described EquinOCS scenario.

Adopting this new perception of user interfaces being DSLs and thinking this idea further, other aspects in system development become blurred, including:

- **Interfaces:** As LDE promotes source models instead of source code, any changes may be carried out using adequate DSLs, like graphical modeling languages. This applies independent of whether the modeling is done upfront during the initial development or later at system runtime to change existing features on demand. Consequently, even those features that previously were considered static can now be changed via a unified (graphical) user interface at system runtime. The only difference that remains is how the changes are implemented, process may be hidden from the modeling users (see "Implementation" below).

- **Users:** When using tailored DSLs certain modeling tasks can be performed by stakeholders without programming skills or technical backgrounds. There might still be different types of users involved, though not necessary anymore for a wide range of use cases.

Ideally, in most scenarios there should be no need for involving programmers or other software developers anymore. Hence, *who* develops and *who* configures/uses the system becomes blurred.

- **Platforms:** With IMEs becoming available in the web, like the upcoming Cinco Cloud [24, 66, 141, 142], there may be no distinction between the system and the development platform anymore. Additional adjustment and alignment of the user interface of the modeling environment with that of the modeled web application (e.g. by adapting the appearance or appropriate branding) could further eliminate the perceived distinction between the modeling environment and the modeled application. As an alternative, by integrating the respective model editors into the web application directly the perceived distinction between the different applications may be overcome completely.
- **Implementation:** Following the LDE approach, the application code is generated from models and then built, packaged and deployed in a predefined manner, e.g. by means of an automated CI/CD pipeline. This process is consistent and repeatable, independent of whether the modeling is done upfront during the initial development steps or later at runtime when changing existing features on demand. In the latter case, the newly generated version of the application would replace the previously deployed version. However, this implementation process may be hidden from the modeling users so that the foremost distinction between runtime configuration and software programming becomes blurred and the only remaining difference may be the actual time it takes to deploy (vs. apply) the changes.

Summarizing, certain development tasks that previously involved programming can be replaced by modeling tasks carried out by stakeholders without programming knowledge and by doing so formerly clear system boundaries become increasingly blurred. However, as the above report form examples show, this is a welcome development demanded by involved stakeholders to be addressed and supported by future enhancements of the LDE approach, especially by means of integrating the idea of dynamic modeling environments into the overall concept.

Blurring Line between Abstraction Layers

The described findings are also consistent with very similar observations I made as a sidearm of my research in the slightly different application domain of business modeling [7, 9, 11, 12, 14]. Business modeling is the process of creating a simplified representation of a business idea or concept, which includes various elements such as the business's value proposition, revenue streams, target customers, and key resources. It is typically used to help entrepreneurs and business owners plan and develop their businesses, and to communicate their ideas to others. There are various methods and frameworks for business modeling, such as the Business Model Canvas (BMC) [143] or Lean Startup [144] methodology.

In this context, there exist various strategic notions and many of them have a canvas-based visualizations. Here, a canvas is a visual framework that provides a simple and intuitive way to capture the key components, e.g. of a business model, and are used to structure and organize information about a business or product. In this domain, the BMC established a de-facto standard modeling technique based on a structured template along nine essential components [145, 146]. It is a one-page template divided into tiles that are meant to hold the key elements of a business model, including customer segments, value propositions, channels, customer relationships, revenue streams, key resources, key activities, key partnerships, and cost structure. It is a useful tool for entrepreneurs and business owners to map out their business ideas and to test and refine them. Compared to an empty-page approach, this

lightweight canvas-based approach already provides a level of support that decreased potential pitfalls of overlooking relevant aspects and eased communication.

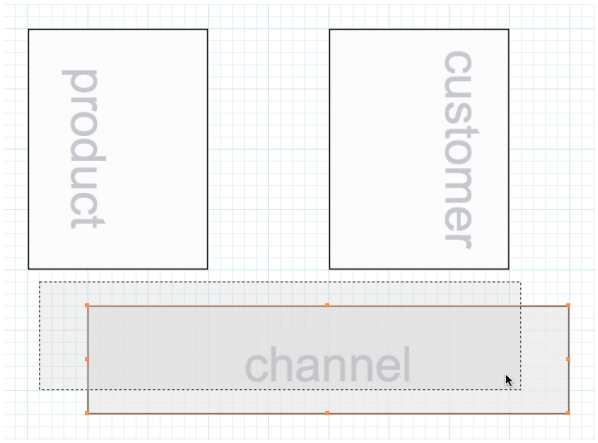

Figure 7.4: Tile arrangement for a custom canvas (from [12])

To ease collaboration, these canvases are often copied to and filled out on generic collaboration platforms, like Miro [147]. There even exist a few dedicated modeling tools, like the well-known Strategyzer [148] app. It is a web application that can be regarded as a starting point for the advent of various IT-supported, canvas-based business modeling tools.

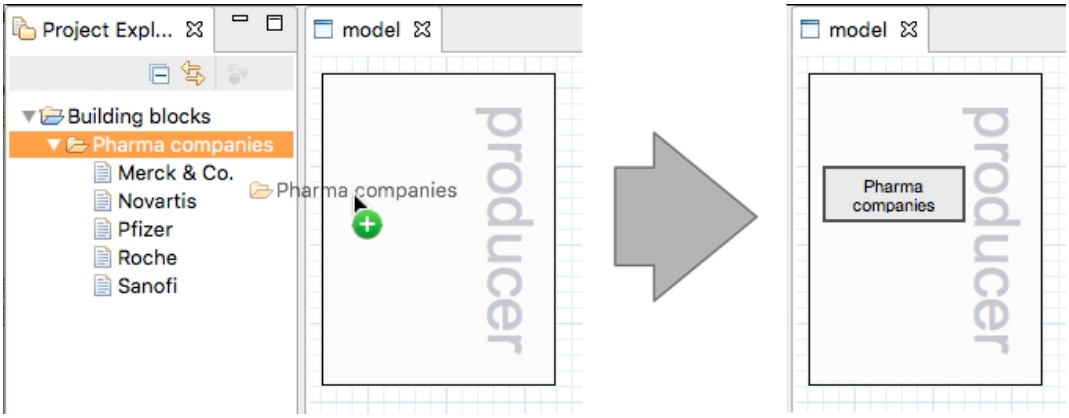

Figure 7.5: Definition of containment rules (from [12])

The blurring line in this context can be observed in terms of two different user groups emerging from the work with these canvas-based approaches. On the one hand, entrepreneurs and business owners fill out these canvases to sketch their business ideas. On the other hand, some business methodologists create new canvases or amend existing ones to better fit to individual mindsets or specific domains (cf. Figure 7.4). The latter task can be seen as a process of creating a new modeling language, here carried out *by* domain experts *for* domain experts. Hence, it is another example that fits the general concept of language-centered feature definition and shows how switching the abstraction layer does not necessarily mean switching from system level to development level. Instead, it means replacing certain development tasks that previously involved programming with modeling tasks carried out by stakeholders without programming knowledge, i.e. blurring formerly clear system boundaries.

Regarding syntax, the task of defining and arranging tiles to create custom layouts is fairly simple. The more challenging part is the definition of constraints. As the canvas designers

are interested in specifying containment constraints, i.e. which components can be inserted into which tiles, they need access to the type information within appropriate taxonomies of building blocks. Then again, defining the relation between these types and the tiles is an easy task, as in graphical modeling languages it may be performed simply in a drag-and-drop fashion (cf. Figure 7.5).

Altogether, this business modeling use case fits the general concept of language-centered feature definition which might be supported by the LDE approach in terms of providing modeling support for various abstraction layers and dynamic yet controlled switching for end users. However, a particular challenge in this application domain is simplicity for the users, which in this case are business professionals on both, the template definition level as well as the template instantiation level. Hence, comprehensive tool support is needed for guiding the users in fulfilling these complex tasks without actually making them feel complex.

Abbreviations

API Application Programming Interface 7

BMC Business Model Canvas 66

CI/CD Continuous Integration/Deployment 13

CPD Cinco Product Definition 17

DAD Dime Application Descriptor 24

DSL Domain-Specific Language 1

DSM Domain-Specific Modeling 55

EMF Eclipse Modeling Framework 17

ELK Eclipse Layout Kernel 55

GME Graphical Modeling Environment 54

HTML Hypertext Markup Language 7

IaC Infrastructure as Code 13

IDE Integrated Development Environment 20

IME Integrated Modeling Environment 1

ISE Immersive Software Engineering 4

jABC Java Application Building Center 21

LDE Language-Driven Engineering 1

LOP Language-Oriented Programming 58

LSP Language Server Protocol 19

MDE Model-Driven Engineering 7

MDSD Model-Driven Software Development 9

mIDE Mindset-Supporting Integrated Development Environment 19

MGL Meta Graph Language 17

MOF MetaObject Facility 53

MPS Meta Programming System 54

MSL Meta Style Language 17

MVC Model-View-Controller 27

OCS Online Conference Service 27

OOP Object-Oriented Programming 10

OTA One Thing Approach 7

PSL Purpose-Specific Language 1

PUTD Path-Up/Tree-Down v

QA Quality Assurance 49

RCP Rich Client Platform 25

SCCE Sustainable Computing for Continuous Engineering

SIB Service-Independent Building Block 21

SLG Service Logic Graph 21

SPLM Software Product Lifecycle Management 38

SQL Structured Query Language 7

UI User Interface

UX User Experience

XMDD eXtreme Model-Driven Design 7

Print References

Each of the URLs listed beneath has been **last visited on February 1st, 2023**.

[AP1] Barry D. Floyd and Steve Boßelmann. “ITSy–Simplicity Research in Information and Communication Technology”. In: *Computer* 46.11 (2013), pp. 26–32.
DOI: 10.1109/MC.2013.332

[AP2] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. “DIME: A Programming-Less Modeling Environment for Web Applications”. In: *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 809–832.
DOI: 10.1007/978-3-319-47169-3_60

[AP3] Steve Boßelmann, Dennis Kühn, and Tiziana Margaria. “A fully model-based approach to the design of the SEcube™ community web app”. In: *2017 12th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE. 2017, pp. 1–7.
DOI: 10.1109/DTIS.2017.7930159

[AP4] Dominic Wirkner and Steve Boßelmann. “Towards Reuse on the Meta-Level”. In: *Electronic Communications of the EASST* 74 (2018).
DOI: 10.14279/tuj.eceasst.74.1047

[AP5] Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, and Bernhard Steffen. “Towards Continuous Quality Control in the Context of Language-Driven Engineering”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 389–406.
DOI: 10.1007/978-3-031-19756-7_22

[AP6] Steve Boßelmann, Stefan Naujokat, and Bernhard Steffen. “On the Difficulty of Drawing the Line”. In: *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018)*. Vol. 11244. Lecture Notes in Computer Science. Springer, 2018, pp. 340–356.
DOI: 10.1007/978-3-030-03418-4_20

[7] Barbara Steffen and Steve Boßelmann. “Domain-Specificity as Enabler for Global Organization aLignment and Decision”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 340–365.
DOI: 10.1007/978-3-031-19762-8_26

[8] Tim Tegeler, Steve Boßelmann, Jonas Schürmann, Steven Smyth, Sebastian Teumert, and Bernhard Steffen. “Executable Documentation: From Documentation Languages to Purpose-Specific Languages”. In: *Leveraging Applications of Formal Methods,*

Verification and Validation. Software Engineering. Vol. 13702. Springer International Publishing, 2022.
DOI: 10.1007/978-3-031-19756-7_10

[9] Barbara Steffen and Steve Boßelmann. "GOLD: Global Organization aLignment and Decision - Towards the Hierarchical Integration of Heterogeneous Business Models". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 504–527.
DOI: 10.1007/978-3-030-03427-6_37

[10] Steve Boßelmann, Johannes Neubauer, Stefan Naujokat, and Bernhard Steffen. "Model-Driven Design of Secure High Assurance Systems: An Introduction to the Open Platform from the User Perspective". In: *The 2016 International Conference on Security and Management (SAM 2016). Special Track 'End-to-end Security and Cybersecurity: from the Hardware to Application'*. Ed. by T.Margaria and Ashu M.G.Solo. CREA Press, 2016, pp. 145–151

[11] Barbara Steffen, Steve Boßelmann, and Axel Hessenkämper. "Effective and Efficient Customization Through Lean Trans-Departmental Configuration". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 757–773.
DOI: 10.1007/978-3-319-47169-3_57

[12] Steve Boßelmann and Tiziana Margaria. "Domain-Specific Business Modeling with the Business Model Developer". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2014, pp. 545–560.
DOI: 10.1007/978-3-662-45231-8_45

[13] Tiziana Margaria, Steve Boßelmann, and Bertold Kujath. "Simple Modeling of Executable Role-Based Workflows: An Application in the Healthcare Domain". In: *Journal of Integrated Design and Process Science* 17.3 (2013), pp. 25–45.
DOI: 10.1007/978-3-642-34032-1_8

[14] Steve Boßelmann. "Simplicity in Application Development for Business Model Design". In: *International Conference of Software Business*. Springer. 2013, pp. 225–226.
DOI: 10.1007/978-3-642-39336-5_24

[15] Tiziana Margaria, Steve Boßelmann, Markus Doedt, Barry D. Floyd, and Bernhard Steffen. "Customer-Oriented Business Process Management: Visions and Obstacles". In: *Conquering Complexity*. Ed. by Mike Hinchey and Lorcan Coyle. Springer London, 2012, pp. 407–429.
DOI: 10.1007/978-1-4471-2297-5_16

[16] Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley / ACM Press, 2010

[17] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Createspace Independent Publishing Platform, 2013

[18] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. "Transformation in Intentional Programming". In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. USA: IEEE Computer Society, 1998, p. 114

[21] Stefan Naujokat. "Heavy Meta. Model-Driven Domain-Specific Generation of Generative

Domain-Specific Modeling Tools". Dissertation. Dortmund, Germany: TU Dortmund University, Aug. 2017.
DOI: 10.17877/DE290R-18076
URL: http://hdl.handle.net/2003/36060

[22] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. "CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools". In: *Software Tools for Technology Transfer* 20.3 (2017), pp. 327–354.
DOI: 10.1007/s10009-017-0453-6

[23] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. "Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019.
DOI: 10.1007/978-3-319-91908-9_17

[24] Philip Zweihoff, Tim Tegeler, Jonas Schürmann, Alexander Bainczyk, and Bernhard Steffen. "Aligned, Purpose-Driven Cooperation: The Future Way of System Development". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2021, pp. 426–449.
DOI: https://doi.org/10.1007/978-3-030-89159-6_27

[26] Tiziana Margaria, Barry D. Floyd, and Bernhard Steffen. "IT Simply Works: Simplicity and Embedded Systems Design". In: *IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), 2011*. July 2011, pp. 194–199.
DOI: 10.1109/COMPSACW.2011.42

[27] Peter Naur and Brian Randell, eds. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Scientific Affairs Division, NATO, Brussels 39 Belgium, 1969

[28] Chitra Dorai and Svetha Venkatesh. "Bridging the semantic gap with computational media aesthetics". In: *Multimedia, IEEE* 10 (May 2003), pp. 15–17.
DOI: 10.1109/MMUL.2003.1195157

[29] Andreas Makoto Hein. "Identification and Bridging of Semantic Gaps in the Context of Multi-Domain Engineering". In: 2010

[30] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. "The Relevance of Model-Driven Engineering Thirty Years from Now". In: *Proc. of the 17th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '14)*. Lecture Notes in Computer Science 8767. Springer International Publishing, 2014, pp. 183–200.
DOI: 10.1007/978-3-319-11653-2_12

[31] Tiziana Margaria and Bernhard Steffen. "Service-Orientation: Conquering Complexity with XMDD". English. In: *Conquering Complexity*. Ed. by Mike Hinchey and Lorcan Coyle. Springer London, 2012, pp. 217–236.
DOI: 10.1007/978-1-4471-2297-5_10

[32] Tiziana Margaria and Bernhard Steffen. “Business Process Modelling in the jABC: The One-Thing-Approach”. In: *Handbook of Research on Business Process Modeling*. Ed. by Jorge Cardoso and Wil van der Aalst. IGI Global, 2009

[33] Arie Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35 (Jan. 2000), pp. 26–36

[35] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. 2nd ed. Springer, 2010

[36] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 2004

[37] Sven Jörges. *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*. Vol. 7747. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Germany, 2013.
DOI: 10.1007/978-3-642-36127-2

[46] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012

[47] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Apr. 2008.
DOI: 10.1002/9780470249260

[48] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Professional, 1998

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994

[51] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. 1st. O’Reilly Media, Inc., 2016

[54] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. “Borg, omega, and kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57

[55] Patrick Debois, Jez Humble, Joanne Molesky, Eric Shamow, Lawrence Fitzpatrick, Michael Dillon, Bill Phifer, and Dominica DeGrandis. “Devops: A software revolution in the making”. In: *Journal of Information Technology Management* 24.8 (2011), pp. 3–39

[56] G. Kim, J. Humble, P. Debois, J. Willis, and N. Forsgren. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. IT Revolution Press, 2021
URL: https://books.google.de/books?id=8kRDEAAAQBAJ

[57] Jennifer Davis and Ryn Daniels. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. 1st. O’Reilly Media, Inc., 2016

[59] Tim Tegeler, Sebastian Teumert, Jonas Schürmann, Alexander Bainczyk, Daniel Busch, and Bernhard Steffen. “An Introduction to Graphical Modeling of CI/CD Workflows with Rig”. In: *Leveraging Applications of Formal Methods, Verification*

and Validation. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2021, pp. 3–17.
DOI: 10.1007/978-3-030-89159-6_1

[62] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, Boston, MA, USA, 2008

[63] Clément Guy, Benoît Combemale, Steven Derrien, Jim R. H. Steel, and Jean-Marc Jézéquel. "On Model Subtyping". In: *Modelling Foundations and Applications*. Ed. by Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 400–415

[64] Markus Völter. "Language and IDE modularization, extension and composition with MPS". In: *Pre-proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)* (2011), pp. 395–431

[66] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen. "Pyro: Generating Domain-Specific Collaborative Online Modeling Environments". In: *Proc. of the 22nd Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*. 2019.
DOI: 10.1007/978-3-030-16722-6_6

[68] Rachel Murphy Ciara Breathnach and Tiziana Margaria. "From No- to Low-Code: Transcribathons as Practice-Based Learning for Historians and Computer Scientists". In: *Proc. OER 2021: The 5th IEEE Int. Workshop on Open Education Resources for Computer Science & Information Technology, at COMPSAC 2021, July 16th, 2021, IEEE Computer society*. 2021

[69] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. "Model-Driven Development with the jABC". In: *Hardware and Software, Verification and Testing*. Ed. by Eyal Bin, Avi Ziv, and Shmuel Ur. Vol. 4383. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 92–108.
DOI: 10.1007/978-3-540-70889-6_7

[70] Bernhard Steffen, Tiziana Margaria, Volkar Braun, and Nina Kalt. "Hierarchical Service Definition". In: *Annual Review of Communications of the ACM* 51 (1997), pp. 847–856

[71] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. 2nd. Addison-Wesley Professional, 2010

[73] Philip Zweihoff and Bernhard Steffen. "Pyrus: An Online Modeling Environment for No-Code Data-Analytics Service Composition". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2021, pp. 18–40

[74] Nils Wortmann, Malte Michel, and Stefan Naujokat. "A Fully Model-Based Approach to Software Development for Industrial Centrifuges". In: *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 774–783.
DOI: 10.1007/978-3-319-47169-3_58

[77] Tim Tegeler, Frederik Gossen, and Bernhard Steffen. "A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications". In: *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. 2019, pp. 1–6.
DOI: 10.1109/CONFLUENCE.2019.8776962

[78] Sebastian Teumert. "Visual Authoring of CI/CD Pipeline Configurations". Bachelor's Thesis. TU Dortmund University, Apr. 2021
URL: https://archive.org/details/visual-authoring-of-cicd-pipeline-configurations

[79] John Stark. *Product Lifecycle Management*. Springer International Publishing, 2011.
DOI: 10.1007/978-0-85729-546-0

[81] Scott Chacon and Ben Straub. *Pro git: Everything you need to know about Git*. English. Second. Apress, 2014

[85] Louis Rose, Markus Herrmannsdoerfer, James Williams, Dimitrios Kolovos, Kelly Garcés, Richard Paige, and Fiona Polack. "A Comparison of Model Migration Tools". In: vol. 6394. Oct. 2010.
DOI: 10.1007/978-3-642-16145-2_5

[86] Wael Kessentini and Vahid Alizadeh. "Semi-automated metamodel/model co-evolution: a multi-level interactive approach". In: *Software and Systems Modeling* 21 (Apr. 2022).
DOI: 10.1007/s10270-022-00978-2

[87] Antonia Bertolino, Antonello Calabrò, Maik Merten, and Bernhard Steffen. "Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring." In: *ERCIM News* 2012.88 (2012), pp. 28–29
URL: http://ercim-news.ercim.eu/en88/special/never-stop-learning-continuous-validation-of-learned-models-for-evolving-systems-through-monitoring

[88] Alexander Bainczyk, Bernhard Steffen, and Falk Howar. "Lifelong Learning of Reactive Systems in Practice". In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen. Cham: Springer International Publishing, 2022, pp. 38–53.
DOI: 10.1007/978-3-031-08166-8_3
URL: https://doi.org/10.1007/978-3-031-08166-8_3

[89] Alexander Bainczyk, Alexander Schieweck, Malte Isberner, Tiziana Margaria, Johannes Neubauer, and Bernhard Steffen. "ALEX: Mixed-Mode Learning of Web Applications at Ease". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 655–671.
DOI: 10.1007/978-3-319-47169-3_51

[90] Tim Tegeler. "A Lingualization Strategy for Knowledge Sharing in Large-Scale DevOps". PhD thesis. TU Dortmund University, 2023

[92] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Boston, MA, USA, 2008

[97] Nianping Zhu, John Grundy, and John Hosking. "Pounamu: A Meta-Tool for Multi-View Visual Language Environment Construction". In: *2004 IEEE Symposium on Visual Languages and Human Centric Computing*. 2004.
DOI: 10.1109/VLHCC.2004.41

[98] John Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li. "Generating Domain-Specific Visual Language Tools from Abstract

Visual Specifications". In: *IEEE Transactions on Software Engineering* 39.4 (2013), pp. 487–515.
DOI: 10.1109/TSE.2012.33

[100] Vladimir Viyovic, Mirjam Maksimovic, and Branko Perisic. "Sirius: A rapid development of DSM graphical editor". In: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, July 2014.
DOI: 10.1109/ines.2014.6909375

[101] Uwe Kastens, Peter Pfahler, and Matthias T. Jung. "The Eli System". In: *Proc. of the 7th Int. Conf. on Compiler Construction (CC '98)*. Vol. 1383. Lecture Notes in Computer Science. Springer, 1998, pp. 294–297.
DOI: 10.1007/BFb0026439

[102] Carsten Schmidt, Bastian Cramer, and Uwe Kastens. *Generating visual structure editors from high-level specifications*. Tech. rep. University of Paderborn, Germany, 2008

[103] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend*. 2nd. Packt Publishing, 2016

[105] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 444–463.
DOI: 10.1145/1869459.1869497

[106] Eelco Visser and Zine-el-Abidine Benaissa. "A Core Language for Rewriting". In: *Electronic Notes in Theoretical Computer Science* 15 (1998), pp. 422–441.
DOI: 10.1016/S1571-0661(05)80027-1

[108] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Chuck Thomasson, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. "The Generic Modeling Environment". In: *Workshop on Intelligent Signal Processing (WISP 2001)*. 2001

[109] Akos Lédeczi, Miklós Maróti, and Péter Völgyesi. *The Generic Modeling Environment*. Tech. rep. Nashville, TN, 37221, USA: Institute for Software Integrated Systems, Vanderbilt University, 2003
URL: http://www.isis.vanderbilt.edu/sites/default/files/GMEReport.pdf

[111] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2009, pp. 168–177.
DOI: 10.1109/SCAM.2009.28

[112] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "EASY Meta-programming with Rascal". In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–289.
DOI: 10.1007/978-3-642-18023-1_6

[113] Paul Klint, Jurgen Vinju, and Tijs Storm. "Language Design for Meta-Programming in the Software Composition Domain". In: *Proceedings of the 8th International*

Conference on Software Composition. SC '09. Zurich, Switzerland: Springer-Verlag, 2009, pp. 1–4.
DOI: 10.1007/978-3-642-02655-3_1

[115] Steven Kelly, Kalle Lyytinen, and Matti Rossi. "MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment". In: *CAiSE*. Vol. 1080. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1996, pp. 1–21.
DOI: 10.1007/3-540-61292-0_1

[117] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming Graphical Modeling". In: *Proc. of 13th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*. Lecture Notes in Computer Science 6394. Springer, 2010, pp. 196–210.
DOI: 10.1007/978-3-642-16145-2_14

[120] Fabien Campagne. *The MPS Language Workbench, Vol. 1*. 1st. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2014

[122] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. "Towards User-Friendly Projectional Editors". In: *Proc. of 7th Int. Conf. on Software Language Engineering (SLE 2014)*. 2014.
DOI: 10.1007/978-3-319-11245-9_3

[125] Fatima Rani, Pablo Diez, Enrique Chavarriaga, Esther Guerra, and Juan de Lara. "Automated Migration of EuGENia Graphical Editors to the Web". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020.
DOI: 10.1145/3417990.3420205

[127] Fatima Rani, Pablo Diez, Enrique Chavarriaga, Esther Guerra, and Juan de Lara. "Automated Migration of EuGENia Graphical Editors to the Web". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020.
DOI: 10.1145/3417990.3420205
URL: https://doi.org/10.1145/3417990.3420205

[139] Sergey Dmitriev. "Language Oriented Programming: The Next Programming Paradigm". In: *JetBrains onBoard Online Magazine* 1 (2004)
URL: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/

[140] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. "A Programmable Programming Language". In: *Communications of the ACM* 61.3 (Mar. 2018), pp. 62–71.
DOI: 10.1145/3127323

[141] Philip Zweihoff. "Aligned and Collaborative Language-Driven Engineering". Dissertation. Dortmund, Germany: TU Dortmund, 2022.
DOI: 10.17877/DE290R-22594
URL: https://eldorado.tu-dortmund.de/handle/2003/40736

[142] Alexander Bainczyk, Daniel Busch, Marco Krumrey, Daniel Sami Mitwalli, Jonas Schürmann, Joel Tagoukeng Dongmo, and Bernhard Steffen. "CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering". In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering.*

Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 407–425.
DOI: 10.1007/978-3-031-19756-7_23

[143] Alexander Osterwalder. "The business model ontology a proposition in a design science approach". PhD thesis. Université de Lausanne, Faculté des hautes études commerciales, 2004

[144] Mark A Hart. "The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses". In: *Journal of Product Innovation Management* 29.3 (2012), pp. 508–509.
DOI: 10.1111/j.1540-5885.2012.00920_2.x

[145] Alexander Osterwalder and Yves Pigneur. *Business Model Generation*. John Wiley & Sons, Inc., 2010

[146] Alexander Osterwalder, Yves Pigneur, Manuel Au-Yong Oliveira, and João José Pinto Ferreira. "Business Model Generation: A Handbook for Visionaries, Game Changers and Challengers". In: *African journal of business management* 5.7 (2011), pp. 22–30

Online References

Each of the online references listed beneath has been **last visited on February 1st, 2023**.

[19] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* June 2005
URL: http://martinfowler.com/articles/languageWorkbench.html

[20] *Cinco SCCE Meta Tooling Suite*
URL: https://cinco.scce.info

[25] EU Seventh Framework Programme (FP7). *ITSy - IT Simply Works*
URL: https://cordis.europa.eu/project/id/258058

[34] MathWorks. *Simulink*
URL: http://www.mathworks.com/products/simulink

[38] Python Software Foundation. *Python*
URL: https://www.python.org

[39] *Jinja*
URL: https://palletsprojects.com/p/jinja/

[40] Mozilla Foundation. *JavaScript*
URL: https://developer.mozilla.org/JavaScript

[41] *Handlebars*
URL: https://handlebarsjs.com

[42] *Mustache*
URL: https://mustache.github.io

[43] Oracle Corporation. *Java*
URL: https://www.java.com

[44] Apache Software Foundation. *Freemarker*
URL: https://freemarker.apache.org

[45] Apache Software Foundation. *The Apache Velocity Project*
URL: https://velocity.apache.org/

[50] Apache Software Foundation. *Maven*
URL: https://maven.apache.org

[52] Docker Corporation. *Docker*
URL: https://www.docker.com

[53] The Kubernetes Authors. *Kubernetes*
URL: https://kubernetes.io

[58] Sebastian Teumert. *Rig — Low-Code CI/CD Modeling*
URL: https://scce.gitlab.io/rig/

[60] *Xtext - Language Engineering Made Easy!*
URL: http://www.eclipse.org/Xtext/

[61] *Eclipse Modeing Framework*
URL: http://www.eclipse.org/modeling/emf/

[65] *DIME Integrated Modeling Environment*
URL: https://dime.scce.info

[67] *Official page for Language Server Protocol*
URL: https://microsoft.github.io/language-server-protocol/

[72] Springer International Publishing AG. *Equinocs*
URL: https://equinocs.springernature.com

[75] *SCCE - Sustainable Computing for Continuous Engineering*
URL: https://gitlab.com/scce

[76] *Rig*
URL: https://gitlab.com/scce/rig

[80] *Git*
URL: https://git-scm.com

[82] *Git Flow*
URL: https://www.gitkraken.com/learn/git/git-flow

[83] Vincent Driessen. *A successful Git branching model*
URL: http://nvie.com/posts/a-successful-git-branching-model/

[84] Apache Software Foundation. *Daffodil*
URL: https://daffodil.apache.org/

[91] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*
URL: http://www.omg.org/spec/MOF/2.4.1/PDF

[93] Eclipse Foundation Corporation. *Acceleo*
URL: https://www.eclipse.org/acceleo

[94] *The Eclipse Foundation*
URL: https://www.eclipse.org/

[95] Microsoft. *Visual Studio*
URL: https://visualstudio.microsoft.com

[96] *Marama*
URL: https://wiki.auckland.ac.nz/display/csidst/

[99] *Eclipse Sirius*
URL: http://www.eclipse.org/sirius/

[104] *Spoofax: The Language Designer's Workbench*
URL: https://spoofax.dev

[107] *WebGME*
URL: https://webgme.org/

[110] Eclipse Foundation Corporation. *Rascal*
URL: https://www.rascal-mpl.org

[114] *MetaCase - Domain-Specific Modeling with MetaEdit+*
URL: http://www.metacase.com

[116] *KIELER Project*
URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER

[118] *Eclipse Layout Kernel*
URL: http://www.eclipse.org/elk/

[119] JetBrains. *Meta Programming System*
URL: https://www.jetbrains.com/mps/

[121] *JetBrains*
URL: https://www.jetbrains.com

[123] Google Corporation. *Google Docs*
URL: https://docs.google.com

[124] Microsoft Corporation. *Microsoft 365*
URL: https://www.microsoft.com/microsoft-365

[126] Eclipse EuGENia. *Graphical Model Editor development with EuGENia/GMF*
URL: https://www.eclipse.org/epsilon/doc/eugenia/

[128] Clay Richardson and John Rymer. *New Development Platforms Emerge For Customer-Facing Applications*
URL: https://www.forrester.com/report/New-Development-Platforms-Emerge-For-CustomerFacing-Applications/RES113411

[129] Mendix. *The Definitive Guide to Low-Code Development*
URL: https://www.mendix.com/low-code-guide

[130] AppSheet. *App Sheet*
URL: https://www.appsheet.com/

[131] Microsoft Corporation. *Microsoft Power Apps*
URL: https://powerapps.microsoft.com

[132] *Outsystems*
URL: https://www.outsystems.com

[133] Mendix Technology BV. *Mendix*
URL: https://www.mendix.com/

[134] *Appian*
URL: https://appian.com

[135] *Bubble*
URL: https://bubble.io

[136] Salesforce Corporation. *Lightning*
URL: https://www.salesforce.com/lightning

[137] *Salesforce Corporation*
URL: https://www.salesforce.com

[138] *Ruby Programming Language*
URL: http://www.ruby-lang.org/

[147] *Miro*
URL: https://www.miro.com/

[148] *Strategyzer*
URL: `https://www.strategyzer.com/app`