Aligned and Collaborative 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

Philip Zweihoff

Dortmund

2022

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

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Sven Jörges

Acknowledgements

"No one can whistle a symphony. It takes a whole orchestra to play it". (H.E. Lucook)

First of all, I would like to thank Bernhard Steffen for being my supervisor for the dissertation and for giving me the opportunity to work at his chair. Throughout my time at the chair, he made sure that we could innovate as a team, was always open to new ideas, and made sure that we were pursuing a common big picture.

Furthermore, I would like to thank Sven Jörges for being the second reviewer of my dissertation. I would also like to thank Heinrich Müller and Günter Rudolph for being part of my doctoral committee.

Without the support of the whole team of LS5 over the last years it would not have been possible for me alone to realize the concepts of CINCO Cloud, DIME and Pyro. Starting with Stefan Naujokat, Michael Lybecait and Johannes Neubauer through whom I got in touch with the chair and was able to develop Pyro. Dominic Wirkner, who shares the office and the same taste in music with me, always has an open ear for problems and with whom I had a lot of fun outside the office and at conferences. I want to thank Tim Tegler for never letting me forget to always think about the Ops side as well. I also want to thank Alex, Sami and Joel who have supported me as project members of Pyro and the CINCO Cloud since the beginning.

I want to thank my wife Cristina who supported me throughout the whole time and especially in the last year so that I could fully concentrate on my target.

Last but not least I would like to thank my parents Gina and Andi, who have spared no effort to encourage and support me at every opportunity even if they understand little of what I do.

Abstract

Today's software development is increasingly performed with the help of low- and no-code platforms that follow model-driven principles and use domain-specific languages (DSLs). DSLs support the different aspects of the development and the user's mindset by a tailored and intuitive language. By combining specific languages with real-time collaboration, development environments can be provided whose users no longer need to be programmers. This way, domain experts can develop their solution independently without the need for a programmer's translation and the associated semantic gap.

However, the development and distribution of collaborative mindset-supporting IDEs (mIDEs) is enormously costly. Besides the basic challenge of language development, a specialized IDE has to be provided, which should work equally well on all common platforms and individual heterogeneous system setups.

This dissertation describes the conception and realization of the web-based, unified environment CINCO Cloud, in which DSLs can be collaboratively developed, used, transformed and executed. By providing full support at all steps, the philosophy of language-driven engineering is enabled and realized for the first time.

As a foundation for the unified environment, the infrastructure of cloud development IDEs is analyzed and extended so that new languages can be distributed on-the-fly. Subsequently, concepts for language specialization, refinement and concretization are developed and described to realize the language-driven engineering approach, in a dynamic cluster-based environments. In addition, synchronization mechanisms and authorization structures are designed to enable collaboration between the users of the environment. Finally, the central aligned processes within the CINCO Cloud for developing, using, transforming and executing a DSL are illustrated to clarify how the dynamic system behaves.

Contents

List of Figures

Listings

Chapter 1

Introduction

The need for software solutions to digitalize processes and open up new business areas is constantly increasing. Due to the great potential for optimization in everyday life and within companies, individual software development is becoming increasingly important. The large demand cannot be mastered thereby exclusively by the employment of classical programmers, since this requires large resources and personnel employment.

For this reason the Low-Code and No-Code platforms gain more and more popularity [SSS21, AMU$^+$21, WW21], since they enable non-programmers to solve their challenges independently in a domain-specific and thus simplified development environment. The conceptual basis for these platforms are domain-specific languages (DSLs) [Fow10], which are tailored to a specific target group and application purpose. By using DSLs it is possible to close the semantic gap [Gho11] between a domain expert and the technical realization by applying techniques of model-driven software development (MDSD) [Sch06] and eXtreme model-driven design (XMDD) [MS20] which even produces better code [MdSMAM21]. As already shown by [CHB$^+$19, HSCJ09, MMBL93], graphical DSLs provide the most intuitive and easiest access for non-technical users to a DSL by mapping the established notations used for a domain. Graphical DSL tools such as jABC [SMN$^+$06], DIME [BFK$^+$16] and jETI [MNS05] have existed since the nineties and have already demonstrated the advantages of the XMDD approach in various scenarios [SM20, LMN15, BWLM17].

However, the realization of XMDD requires that DSLs must be designed and implemented for each individual domain and purpose, which in turn results in enormous development effort. This necessity is addressed by the so-called Language Workbenches like MPS [Cam14] or Xtext [Bet16] which support the development of new textual DSLs so that the manual implementation effort is reduced to a minimum. Language-Driven Engineering (LDE) [SGNM19] extends this approach by vertically refining and horizontally specializing DSLs to provide a tailored language at the appropriate level of abstraction. The execution of a DSL is then done by successive DSL to DSL transformation and a final code generation. To optimally support non-technical users, it is also necessary to establish a simplified and intuitive collaboration between users by enabling a parallel and automatically synchronized usage of each DSL [DRFMM17].

For the area of graphical DSLs development, the CINCO Meta Tooling Suite [NLKS18] was realized, which follows the principle of DFS (Domain-Specific, Full-Code-Generation and Service-Oriented) described by Naujokat [Nau17], supporting the declarative definition of domain-specific tools by DSL. The development of a new DSL with CINCO is based on a metamodel description in the form of the declarative specification languages. To enable the simplified and setup-less use of a DSL developed with CINCO, the Pyro [ZNS19] project was launched, which generates an alternative web-based and collaborative environment for each DSL. To preserve the benefits of Pyro for existing tools, the entire generation process can be started from existing DSL metamodels of the CINCO Meta Tooling Suite. Pyro provides an alternative to the Eclipse-based generation process of Lybecait [Lyb19] and creates a web-based tool instead. Pyro thus enables the effortless migration of a local desktop tool previously created with CINCO to a distributed environment that can be accessed immediately by any user as described in [LKZ$^+$18]. However, a realization of the LDE approach with CINCO and Pyro is still not possible, since no unified and reflexive structure exists in which DSLs can be refined, specialized and transformed easily.

To overcome these challenges, the benefits of Pyro and CINCO are combined in an aligned and collaborative environment called CINCO Cloud. In line with the one-thing-approach (OTA) [MS09], the development and use of each DSL can be realized in the same web-based platform. With the introduction of the CINCO Cloud as an archimedean point [SN16] the following goals can be achieved:

- **Aligned DSL Development:** The simplified realization of new graphical DSLs already offered by CINCO can be raised to a new level in terms of accessibility and setup-less usage. This allows users to develop and directly use their own DSLs in a web-based environment without installation or special local resources. The entire development setup including all dependencies, languages and runtime environments is automatically maintained and managed within the CINCO Cloud.

- **Collaborative Modeling:** In contrast to classical version control systems, the reuse of Pyro's synchronization architecture enables simultaneous collaboration in real-time so that users can work together like in Google Docs [ZSD12].

- **DSL Refinement:** Thanks to the central management of all DSLs inside the CINCO Cloud, vertical refinement and horizontal specialization are supported by design.

- **DSL Interoperability:** By serving each DSL as a service within the environment, the LDE-specific processes for transformation and generation can be realized in a uniform manner by tailored APIs.

1.1 My Contribution

The idea to align and simplify the development and usage of domain-specific modeling tools inside one holistic environment, which led to the CINCO Cloud project, arose from my experiences in modeling tool development for industrial projects and teaching. Despite the use of language workbenches to simplify the development, collaborative and direct use of a DSL by non-technical domain experts has always posed a problem. For this reason, I began developing easily accessible and collaborative web-based modeling environments that made it easier for users to benefit from the DSLs designed for them. The CINCO Cloud project combines my experience in web development, distributed systems, Dev Ops, service-orientation and language development in one environment so that the language-driven engineering concept can be fully and uniformly realized for the first time. In essence, this dissertation makes three contributions towards the CINCO Cloud that focuses on alignment through overarching service-orientation and intuitive real-time collaboration:

The first main contribution of this dissertation deals with my conception of the online language workbench CINCO Cloud in the context of language-driven engineering. For this purpose, the refinement and specialization of DSLs is first explained in order to support each user at an appropriate level of abstraction depending on their expertise. To ensure the necessary flexibility in the development of new DSLs, I present a uniform structure in which DSLs can be generated with the help of a central meta-metamodel and associated generator. Subsequently, it is clarified how the concretization of a DSL is realized in order to transform DSL instances, generate code and execute code. The step-by-step transformation required for the concretization is executed against the direction of refinement so that a DSL instance is translated into more and more general languages. Finally, a textual code generator is used to transform a DSL instance to executable code which can be executed directly inside the CINCO Cloud. To realize the transformation, a service-oriented concept [MS12] is presented which can be used by uniform interfaces for the instrumentation of DSLs. In this way, transformation chains consisting of modular DSLs can be performed. This contribution focuses on my realization of the LDE approach in the CINCO Cloud, whose vision was described in [ZTS$^+$21].

The second main contribution of this dissertation deals with the concept of collaboration within the CINCO Cloud. In the context of LDE, non-technical users are focused, who are used to intuitive real-time collaboration in contrast to asynchronous version-control systems. At the same time, intuitive access to the CINCO Cloud must be provided by tailoring the environment to a user's needs and intentions. The distributed architecture and the modularization of a DSL into a language server and a client-side editor enables users to work simultaneously on the shared resources inside a project workspace. Various tools [Bru12, SH13, Ros07] have already shown that the use of simultaneous collaboration in domain-specific tools is particularly helpful to enable barrier-free collaboration. To realize collaboration, the concept of operations-based [BAS14] bidirectional synchronization implemented within Pyro is utilized in the CINCO Cloud. To provide the most interruption-free collaboration possible, domain-specific conflict control (DSCC) [ZBS22] is used, which leverages information from the metamodel of a given DSL for efficient conflict detection. This contribution focuses on my concept of simultaneous collaboration for the CINCO Cloud, whose architectural foundation was described in [ZNS19] and conflict handling in [ZBS22].

The third main contribution of this dissertation describes the technical implementation of the CINCO Cloud to realize the LDE approach. The web-based development environment is technically based on the approaches of cloud development IDEs such as Gitpod [12] and Eclipse Che [7], which are characterized by the ability to dynamically create and manage workspaces. The CINCO Cloud extends this concept to a language workbench so that a newly developed DSL can be used immediately in a specialized workspace. First, the system architecture of the CINCO Cloud is illustrated, which includes a central manager to control the cluster-based infrastructure and administers the environment. Then the central workspace component is explained which integrates the modular DSLs as a service. Each DSL module is a standalone component that provides the specific language features and supports simultaneous collaborative editing. It is described which components are involved in a DSL module and how the interoperability between the manager and other DSL modules takes place. Subsequently, the realization of the LDE specific processes within the CINCO Cloud are illustrated:

- **DSL Generation and Release:** Starting from a specification of the abstract and concrete syntax of a DSL, the central generator creates the corresponding DSL module which is build, registered and published within the CINCO Cloud. Subsequently, the generated DSL module can be integrated into a workspace to grant users access to the new DSL.

- **Model Transformation and Execution:** In order to execute a model within the CINCO Cloud, more abstract DSL modules are instrumented via uniform interfaces. In this way, step-by-step transformations and a final code generation can be enabled. Thanks to the dynamic infrastructure of the CINCO Cloud, the generated code can be build, deployed and executed directly without leaving the environment.

This contribution focuses on my implementation of the CINCO Cloud, which has not yet been published. So far, only the vision of the CINCO Cloud has been described in [ZTS$^+$21] and the aspect of service-oriented integration of web-based IDEs has been illustrated using the Pyrus tool as an example in [ZS21b].

To illustrate the special features and advantages of CINCO as a Language Workbench and Pyro as a domain-specific web-based modeling environment, several papers have been published. The following papers include examples of DSL tools for web application creation and unified cross-platform development for the desktop and the web. In addition, the aspects of simultaneous collaboration and service-orientation focused by Pyro were presented in several papers. CINCO was also used in the following student project groups that I supervised: [BBC$^+$19, GvKT$^+$20].

1.2 Context of Attached Publications

Along the development of the CINCO Meta Tooling Suit and Pyro, several domain-specific tools have been realized to illustrate the different aspects and concepts of the Language Workbench. The publications associated with this cumulative dissertation first show various example tools for educational purposes and for model-driven development of web applications. The architecture and communication model of Pyro were described in [ZNS19] and further the special mechanisms for simultaneous collaboration and conflict detection [ZBS22] and the service-oriented integration [ZS21b]. The motivation for developing the CINCO Cloud as a combination of CINCO and Pyro was described in [ZTS$^+$21].

Pyro: Generating domain-specific collaborative online modeling environments

This paper [ZNS19] presents Pyro as an alternative generator to CINCO for creating graphical domain-specific modeling environments for the web. Instead of a single user desktop IDE based on the Eclipse Modeling Framework (EMF) [SBMP08] and the Rich Client Platform RCP [MLA10], Pyro generates an equivalent environment for simultaneous collaboration that can be used independently across any browser platform. The distributed system implemented by Pyro uses the operations-based communication model BASE [Vog09] by exchanging CRDTs [SPBZ11] over bi-directional communication channels. The publication illustrates the architecture of the generated system using the AADL language [FG12] and describes which parts are created depending on the specified metamodel of the graphical DSL. In particular, the synchronization mechanism for a graphical structured language is described in comparison to textual languages.

Aligned, Purpose-Driven Cooperation: The Future Way of System Development

This paper [ZTS$^+$21] describes the problem of reproducing today's increasingly complex development setups on the different systems of the individual members in a development team. Due to the large number of tools, languages, build systems, dependencies, SDKs and runtime environments involved, combined with the heterogeneous systems on which software is developed, users are continuously faced with migration and synchronization issues. In this paper, the use of Cloud Development IDEs is discussed, which allows all developers to work directly and platform independently on a project without performing a previous manual setup. The concept of Cloud IDEs is evaluated with respect to the application for language workbenches and the potential to realize collaborative language driven engineering. The vision of the CINCO Cloud is explained and a first concept is sketched.

Towards Tailored and Precise Conflict Detection in Real-Time Collaboration Modeling Environments

This paper [ZBS22] illustrates the design and implementation of an efficient and accurate conflict detection mechanism to improve real-time collaboration in graphical modeling. Contrary to the classical synchronization mechanisms of ACID and VCS, concurrent users should not be unnecessarily interrupted in their work or perform manual merges. At the same time, semantic conflicts should be prevented directly during modeling, so that an invalid model is not synchronized at any time. The key to this approach is the usage of syntactical and semantical information given by the metamodel of a graphical DSL using Pyro as an example. The publication first describes the optimistic none-blocking commnuication model and the operation-based synchronization mechanism using commands. Based on the given architecture it is explained how consistency can be achieved. The different conflict types are explained and it is described how the information from the metamodel of a DSL is used to influence the behavior of the domain-specific conflict control (DSCC).

Pyrus: an Online Modeling Environment for No-Code Data-Analytics Service Composition

This paper [ZS21b] describes the design, implementation, and use of Pyrus, a domain-specific tool for data-flow-driven composition of data analysis functions. Pyrus extends the concepts of jETI [MNS05], Taverna [WHF$^+$13], and Kepler [LAB$^+$06] by allowing functions from an online IDE such as Jupyter [KRKP$^+$16] to be used without going through a central repository. For the development of Pyrus, CINCO and Pyro were used so that users can work collaboratively within the web-based environment. The publication describes the integration of Jupyter into Pyrus as an example of how language driven engineering can be used to support domain experts in applied data analysis without the need for programming skills.

A Tutorial Introduction to Graphical Modeling and Meta-Modeling with CINCO

In this paper [LKZ$^+$18], CINCO is explained using the example of a simple DSL developed for teaching. The WebStory language used provides users with the ability to design, generate and play point-and-click adventures. In contrast to textual specification languages, the metamodel of the language was declared using the graphical CINCO specification language (GCS) [Fuh18] and subsequently generated into a web-based tool with the help of Pyro. In the publication, the use of the Pyro generated environment is explained by example.

DIME: A Programming-Less modeling Environment for Web Applications

The paper [BFK$^+$16] introduces the DyWA Integrated Modeling Environment (DIME) by allowing users to design, generate, and execute web applications using three different graphical DSLs. DIME follows the principles of one-thing approach (OTA) [MS09] and extreme-model-driven design (XMDD) [MS20] by allowing users to develop the user interface, processes and data models without textual programming. The application created with DIME provides the basis for the modeling environments generated by Pyro by implementing persistence in both cases using DyWA [NFSM14].

(A Generative Approach for User-Centered, Collaborative, Domain-Specific Modeling Environments)

This paper [ZS21a] describes how current requirements of none-technical domain-experts are met by the Pyro modeling environment. In particular, the specialization of the environment and the real-time collaboration are discussed. The paper gives a detailed explanation on the technical realization of Pyro's generator and modeling environment. Finally, a comprehensive analysis and comparison with alternative environments and approaches is presented.

1.3 Nomenclature

Within this dissertation, keywords from graphics are used in the text. The references to graphics are always marked starting with a capital letter and formatted in *italics* to support traceability.

1.4 Organization of this Dissertation

Following this introduction, Chapter 2 illustrates the foundational concepts and technologies for the realization of the CINCO Cloud. For this purpose, the CINCO Meta Tooling Suite, Pyro and the architecture of Cloud Development IDEs are illustrated. Chapter 3 describes the alignment concept to realize the LDE approach, starting with the DSL creation in Sec. 3.2. Then, in Sec. 3.3, the execution of DSL models by transformation and code generation is described. Chapter 4 illustrates the collaboration concept to support parallel DSL instance usage inside the CINCO Cloud. Sec. 4.1 describes the real-time editing of models and Sec. 4.2 shows the definition of views to customize and manage the user environments. Chapter 5 describes the system architecture and implementation of the CINCO Cloud, starting with the dynamic cluster-based infrastructure in Sec. 5.1. The central components of the CINCO Cloud are described in the following Sec. 5.2 and Sec. 5.3. The LDE specific task for defining, transforming and executing DSLs are described from Sec. 5.4 to Sec. 5.6. The dissertation ends with a summary and gives an outlook on the next steps in Chapter 6.

Chapter 2

Background

This chapter introduces the main concepts, technologies and frameworks used for the realization of the CINCO Cloud.

2.1 Domain-Specific Languages

The use of domain-specific languages (DSLs) is an established way to bridge the semantic gap [Gho11] between a programmer and a domain expert when developing a solution. The need for this approach arises from the different mindsets of the two parties, in that the domain expert has precise knowledge of the intended solution but relies on a programmer speaking the same language to implement it. Since this is usually not the case, DSLs can be used to realize a common formal language for communication. For this reason, DSLs are based on the concept of using a specific and restricted language for a given domain which is less expressive than a general purpose language, but much more intuitive.

Besides providing communication support to different parties, DSLs can also be used to increase productivity [Gho11] in software development. Established examples of DSLs in software development are regular expressions [Brz64] or the SQL query language [BED96], which can be used for a certain part of the solution.

Basically, with internal and external DSLs, there are two different ways in which DSLs can be used. Internal DSLs are embedded within an existing language and represent a subset of the original syntax, allowing the existing tools to be used. The development of internal DSL must be supported in advance by the language as demonstrated by the example of Racket [FFF+15] and Kotlin [Sub21].

External DSLs represent an independent language, which was created for a specific purpose and thus has its own syntax and semantics. This flexibility allows to tailor an external DSL to the notations established in a domain. In contrast to internal DSLs, however, the tooling for external DSLs must be developed additionally, since it is not possible to fall back on a root language. To simplify the development of new external

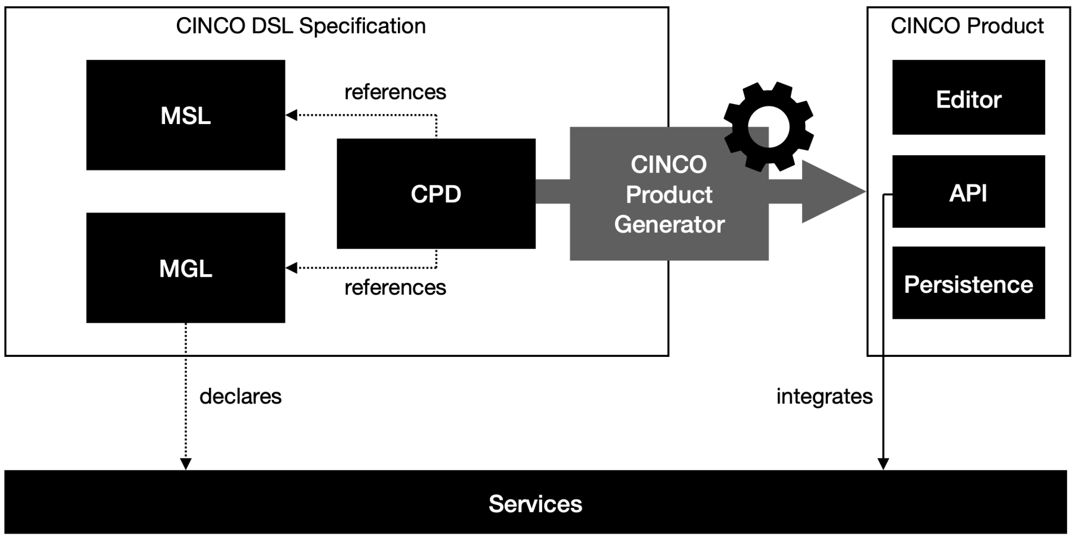


Figure 2.1: DSL specification and CINCO Product generation process.

DSLs, Language Workbenches [EVDSV+13] such as MPS [Cam14] and XText [Bet16] can be used.

Since external DSLs are not based on a textual general purpose language, graphical DSLs [LJJ07] can be used as well, which provide non-programmers with a more intuitive approach to independent language development [HSCJ09]. To develop graphical DSLs there are several language workbenches, which have already been compared by [EVDSV+15]. The CINCO Meta Tooling Suite [NLKS18] represents a framework focused on simplicity during development described by Steffen [MS10], in which DSLs are also used for the description of new DSLs, too.

2.2 CINCO Meta Tooling Suite

CINCO is an open-source language workbench based on the Eclipse IDE [6] and the Eclipse Modeling Framework (EMF) [SBMP08] for the development of graphical DSLs. The concept of CINCO focuses on graph-based graphical languages, allowing the development of new DSLs with a minimal specification. As described by Naujokat [Nau17], CINCO is based on the DFS principles in that the specification is also realized by DSLs, the entire tool is holistically generated and additional functionalities can be integrated in a service-oriented fashion. Fig. 2.1 illustrates the development of a graphical DSL using CINCO. First CINCO offers three specification languages, for the declaration of the abstract and concrete syntax as well as the composition of several languages to a so-called CINCO product.

For the description of the abstract syntax, the Meta Graph Language (*MGL*) is used, which contains the declaration of the nodes and edges present in the DSL. The MGL supports concepts known from object-oriented programming such as polymorphism and abstraction. In addition, constraints can be formulated, which describe how nodes can be connected with edges and embedded into each other.

In order to integrate additional functionalities like validation, code generation, transformers and interpreters or to enable specific behavior like auto completion while using the DSL, CINCO offers the integration of services. Each service is integrated via predefined event-based interfaces, so that depending on the user's interaction with a CINCO product, the respective service can be called.

The Meta Style Language (*MSL*) is used to describe the concrete syntax by assigning a visualization to each node and edge of the language metamodel. For this purpose hierarchical shapes can be declared which can be combined to complex ones. In addition to the shapes, so-called appearances offer the possibility to colorize and parameterize lines, text and shapes to meet the domain's demands.

The CINCO Product Definition (*CPD*) combines different languages to a product and represents the starting point of the following tool generation. The *CINCO Product Generator* process [Lyb19] aggregates all resources and subsequently generates a specialized model *Persistence*, transformation *API* and an *Editor* for the use of the DSL.

CINCO itself is consists of different modules and allows the extension by so-called aspect-oriented meta plugins [Lyb19], which are called during the generation. Meta plugins perform intermediate transformations of the DSL metamodel or additional generation steps, so that further artifacts can be generated.

2.3 Pyro

The Pyro project [ZNS19] uses the meta plugin mechanism of CINCO to realize an alternative tool generation. The goal of Pyro is to create an feature equivalent web-based and collaborative modeling environment based on the same DSL specification and services. In this way, users can use a DSL directly via their browser without local installation and edit the same model in real time, similar to Google Docs [ZSD12].

Fig. 2.2 shows the Model-View Control (MVC) architecture [LR01] of the modeling environment generated by Pyro, realizing a distributed system in contrast to the local desktop CINCO products. The client-server system uses a central relational database to manage all instances of the DSLs in specialized schema. In addition, users, projects and permissions are also persisted in the database, enabling administration. Clients can access the server via corresponding *ReST* controllers [Fie00] to make changes.

The special feature of Pyro is the simultaneous collaboration between different users working on the same model. To ensure high availability, Pyro implements the basically available soft-state (BASE) [Vog09] communication model by mirroring a model on each client before editing. From this point on, only changes are exchanged between client and server by *Commands*. Unlike textual languages, interactions with a graphical model

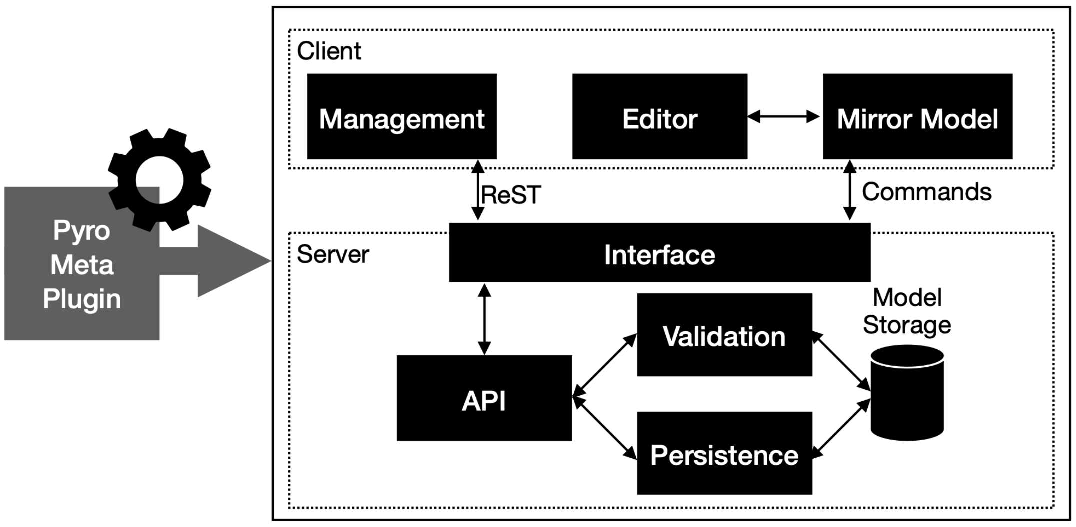


Figure 2.2: Alternative product generation by the Pyro meta plugin.

are clearly identifiable and can be used directly as a context-free replica types (CRDT) [SPBZ11].

Pyro uses CRDTs as operation-based [BAS14] *Commands* to describe the user's changes and send them to the server. The server validates each command with respect to consistency, syntax and semantics of the central model's metamodel and then applies it on the database. Each successfully validated and centrally applied command is then distributed to all users currently working on the model. In case of a conflict, the command is reverted and sent back the the respective client so that the write-repair [TTP$^+$95] mechanism is realized. As described in [ZBS22], global consistency is eventually achieved as soon as no user edits the model and the system calms down.

Besides the direct usability and collaboration of Pyro, all integrated services should be reusable to allow a simplified migration from a CINCO product to a Pyro product. To this end, Pyro generates the same CINCO transformation *API* with an alternative implementation, aligned with the underlying distributed architecture. Thus, all previously created generators, transformers, validators and the event system can be reused.

2.4 Language-Driven Engineering

Language-driven engineering (LDE) [SGNM19] describes an approach in which the individual aspects of an entire software system can be described with appropriate specialized languages by different stakeholders. The necessary translation from the "what" description to the "how" implementation is therefore no longer the responsibility of a programmer, but an inherent part of the DSL.

The key to this approach is the service-oriented horizontal composition and vertical refinement of different DSLs to realize a particular domain-specific tool. To finally generate a software, a modular model-to-model transformation and final code generation is required. In contrast to classical programming, LDE does not aim at mastering all challenges in one general purpose language but offers a suitable DSL for each aspect. In this way, the respective domain experts can work on a project with a DSL tailored to their needs, level of abstraction and expertise.

The realization of LDE is based on the idea of so-called Mindest-Supporting Integrated Development Environments (mIDEs) [SGNM19] which combine the different DSLs. Each DSL in the mIDE represents an independent module, which can be integrated into other DSLs for refinement. The resulting hierarchy of DSLs allows the realization of entire software system by using transformers and generators.

The DIME environment [BFK$^+$16] illustrates the functionality of a mIDE by describing the different aspects of a web application using three different graphical DSLs as described by [BNS18]. The underlying data DSL allows to describe types, attributes and associations that reflect the data schema and corresponding operations. The data DSL is in turn integrated into a process DSL and a GUI DSL so that the representation and manipulation of the data can be refined. The process and GUI DSLs incorporate the types described by the data DSL as variables and operations so that each aspect can be handled by respective experts.

In addition to horizontal composition, a mIDE provides the ability to create new variants of a DSL, enabling vertical refinement. Through this approach, purpose-bound variants of a DSL can be created and DSL refinements can be replaced in a modular way. Through the specific abstraction layers, stakeholders of different expertise can support each other and participate in a project. The generation of the entire system is then realized along the refinement hierarchy by translating step-by-step from the description to executable code.

With respect to the DIME framework, a process DSL could be designed by vertical refinement, which is based on acyclic data flow instead of control flow. By replacing the execution paradigm, for example, a domain expert of data analysis and scientific workflows could be better addressed and supported. The new dataflow process DSL can then be translated to a control flow process by topological sorting as illustrated in [ZS21b], so that the existing generators and transformers of the process DSL can be reused.

However, the realization of LDE is complex and usually requires the use of language workbenches to create new DSLs and to manage the DSL interoperability. The central challenge arises from the implementation of vertical composition, horizontal refinement and the evolution of a DSL, which should be supported in a mIDE. From a technical point of view, the realization of an mIDE requires two mechanisms:

- **Service-Orientation**: Each DSL has to be realized as a modular service to be reusable inside a mIDE. For this purpose, each DSL has to provide unified APIs for all components, like the editor, validation, transformation, persistence and generation to be used by other DSLs. The service-oriented and loosely coupled integration of DSL simplifies the evolution and replacements of DSL inside the LDE hierarchy.

- **Bootstrapping**: The evolution and refinement requires continuous adaptation and creation of DSLs within the mIDE. For this reason, the mIDE has to provide the ability to develop and use DSLs inside the same mIDE by bootstrapping.

The CINCO Meta Tooling Suite allows the description and generation of mIDEs by constructing a DSL with DSLs. However, in reality, refinement, composition and evolution are complicated because the DSLs used together must be tightly coupled. Due to this coupling, it is not possible to replace or evolve an existing DSL without having to edit the entire mIDE.

2.5 Cloud Development IDEs

Cloud development IDEs such as Gitpod [12], CodeSpaces [3] and Eclipse Che [7] are distributed scalable systems that offer the same abilities as a local desktop IDE and become more and more relevant in the industry [DKRS21]. The advantage of this approach is the independence from local resources and individual development setups, which in most cases are different and therefore need to be continuously adapted and maintained. Through the central management of all project resources, it is possible for cloud development IDEs to prepare and provide all dependent SDKs and the runtime environment automatically. The entire project including the development environment and execution environment are executed on the same central infrastructure, managed by the cloud development IDE. A developer can therefore work directly on the current state of a project from any device without local installation of additional software. Fig. 2.3 shows the system architecture of a cloud development IDE and gives an overview of the individual components used to emulate the features of a desktop IDE. The infrastructure of a cloud development IDE allows the orchestration of all user *Workspaces* in separate and isolated virtual containers. Each container has its own resources which are managed by monitors and load balancers so that development is scalable, independently of other users.

The central administration takes place via a so-called *Master Application* which controls the *Orchestrator* behind the cloud development IDE. In this way, the load of development and execution can be continuously monitored and controlled. The users of the cloud development IDE and the associated workspaces are also managed in the *Master Application.*

In addition, a *Plugin Registry* is included that lists embeddable components of an IDE. This way, different developers can work on the same project by using the same unified development setup for a workspace. The combination of *Plugins* is maintained for this purpose as a specification alongside the *Source Code* of the project.

The *Workspace* within the cloud development IDE accordingly symbolizes a combination of the *Source Code* of a project, the *Plugins* and the *Runtime Environment.* Due to

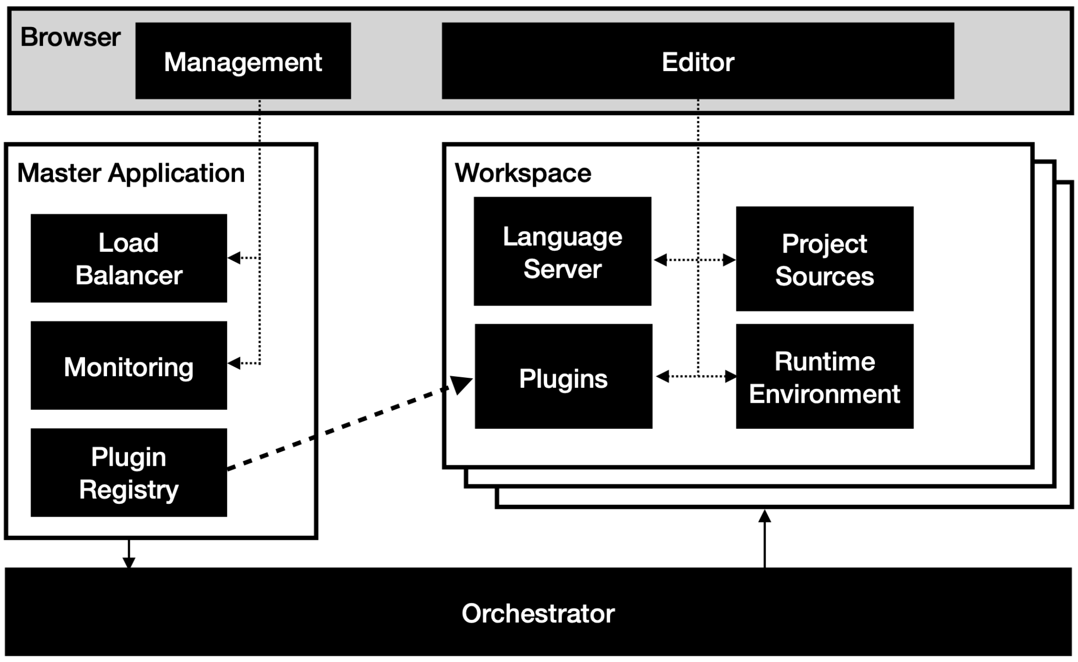


Figure 2.3: Cloud Development IDE system architecture.

the complete specification, preparation can be automated so that users can work directly in on a project.

With regard to the individual programming languages, *Language Servers* are used to enrich the editor with language-specific features such as keyword coloring, validation, content assist and reference jumping. Thanks to the Language Server Protocol (LSP) [20], various language servers can be controlled by the editor via an uniform protocol. This modularization of the language-specific servers makes it possible to dynamically integrate new languages. For this reason, the concepts of Cloud Development IDEs offer an ideal basis for the realization of the aligned LDE.

Chapter 3

Alignment Concept

This chapter describes the challenges and conceptual realization of a holistic and aligned language-driven engineering environment. In contrast to classical programming environments, LDE focuses on the combination of different purpose-oriented and mindset-supporting DSLs, so that users can independently contribute to their respective parts of the solution depending on their expertise.

In order to provide the appropriate level of abstraction for a particular user, it is necessary that DSLs can be vertically refined. For example, a class diagram DSL can be refined by a entity relationship DSL which in turn can be refined by an ontology DSL. Depending on the intended solution and the existing experience of the user, it can be decided which DSL should be used for the realization.

To consider the different aspects of a solution separately, DSLs must be horizontally specialized. In the context of web application development, DIME [BFK^{+}16] illustrates this approach by providing special DSLs for the data schema, processes and the graphical user interface (GUI) declaration. Users can thus model the relevant aspect and collaborate on a solution in parallel.

In addition to the refinement and specialization of DSLs, it is also necessary to ultimately obtain an executable software system. The required source code has to be generated in the context of LDE via successive transformation and generation by continuously concretizing the instances of the DSLs along the refinement hierarchy.

The following sections illustrate how the individual requirements for an LDE environment can be met by using an aligned and holistic tool. For this purpose, Sec. 3.1 first describes the uniform template of all DSLs which combines the core features. Subsequently, Sec. 3.2 describes the vertical refinement and horizontal specialization and Sec. 3.3 the execution by means of transformation and generation.

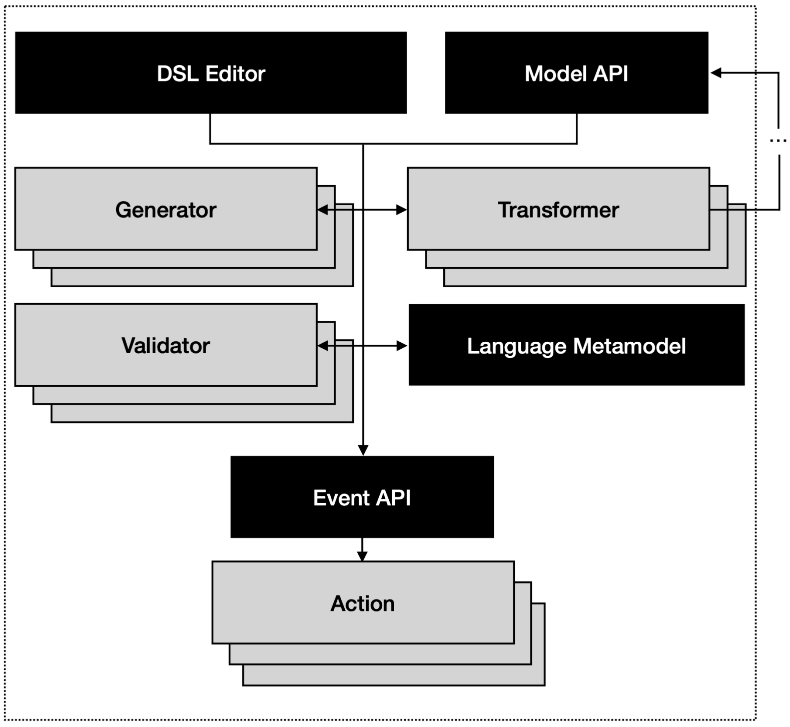


Figure 3.1: Tool template for DSLs of the aligned language-driven engineering environment.

3.1 Tool Template

The prerequisite for the alignment of DSLs within a LDE environment is a uniform structure of the individual mindset supporting tools to ensure simplified refinement, specialization and execution. As already shown by language workbenches such as MPS [Cam14] and MetaEdit+ [SLTM91], there are recurring aspects in the development and use of DSLs which are defined in the tool template. The core of every DSL contains the *Language Metamodel* and a corresponding *DSL Editor*.

In the context of LDE, the additional aspects of validation, generation and transformation are relevant as a supplement to a DSL. By focusing on these aspects, the tool template offers already predefined interfaces so that the development of new DSLs can be carried out more efficiently. At the same time, this creates standardized mechanisms within the environment that can be automated to avoid repetitive work. In reality, further aspects may be necessary, but these are not addressed here, since they can be integrated analogously to the existing service-oriented concept.

Fig. 3.1 illustrates the composition of the individual aspects within a uniform tool template. First, the developed DSL has to be provided by a specific *DSL Editor*, which is customized depending on the *Language Metamodel*. The editor enables the creation of graphical models as instances of the DSL and allows the user specific support.

To ensure the semantical and syntactical correctness of the models, *Validators* are an essential part of the tool template. Through a predefined interface, validators are triggered for the changed part of a model to enable efficient validation. The validators can be partially derived from the metamodel of the DSL and additionally implemented via API. However, within the LDE environment it is also possible to describe additional validators using a custom DSL as described in [ZBS22].

The intuitive use of a graphical DSL requires not only the emulation of known notations but also the recreation of certain functionalities and behaviors of the environment during modeling. The technical basis for these user interaction design is realized with the help of an event-driven system [Cha06] by using the *Event API* to call certain actions as a consequence of a user's interaction. These actions listen for a previously defined interaction such as moving a node to activate a layout mechanism.

Generators and *Transformers* are required to transform a DSL into text [GCN20] or into a model [LKS18]. The distinction is necessary to clarify that *Transformers* are only used to create new graphical models as an instance of another DSL inside the environment via *Model API.* The transformation can be implemented by hand or with the help of a dedicated transformation DSL as described in [KLNS21]. In contrast, generators are used to create textual languages (e.g. source code), which can then be executed.

To enable the transformation of models from one DSL to another DSL, it is necessary that each DSL can be instrumented in a service-oriented way. For this purpose, the tool template contains a standardized *Model API* that provides the same functionality as the editor but can be used by transformers of other DSLs instead.

In the further course of this dissertation, the term DSL always refers to the combination of a language with all associated functionalities as described in the tool template.

3.2 Refinement and Specialization

In the context of LDE, a precise refinement of DSLs has to be realized efficiently by reusing parts of an existing DSL. The continuous refinement of a DSL constraints the expressive power to enable a user to perform a specific task more easily.

Beside the refinement it is likewise crucial to divide the development of a software system on the basis the different aspects (cf. separation of concerns (SoC) [TOHS99]). LDE depicts the separation with the help of different connected DSLs which are designed for a certain aspect of the domain. In this way, different users can work on different parts of the overall solution based on their expertise.

Fig. 3.2 illustrates the concept of vertical refinement and horizontal specialization using the example of a *DSL* (N) that allows the description of aspect N. In this example *DSL* (N) can be used to describe the entire solution. In case the user's expertise is exceeded, the environment provides three refined DSLs N', N'' and N'''. Each of the refined DSLs accordingly supports the description of aspects N', N'' and N''' by horizontal specialization.

The concept of vertical refinement is not limited to the two layers shown in Fig. 3.2. In the context of LDE, the *DSL* (N) is a refinement of a general graphical language DSL

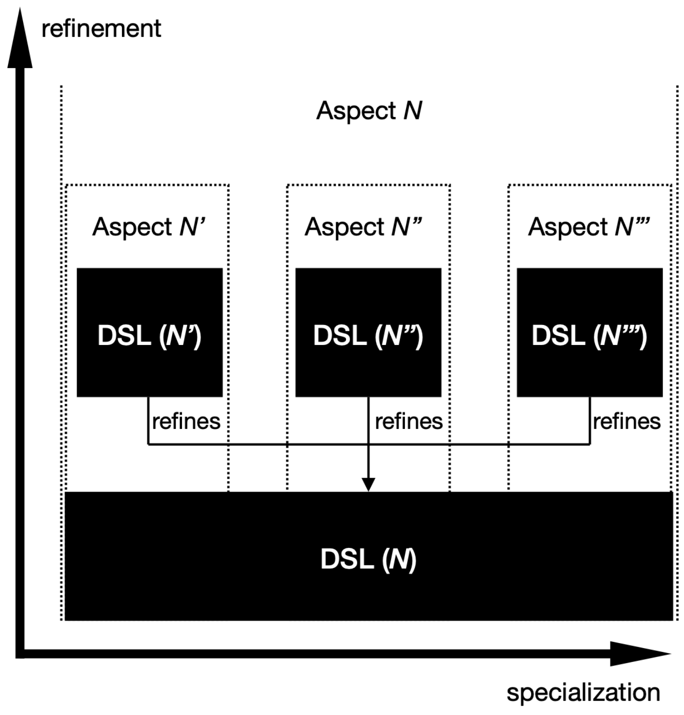


Figure 3.2: Vertical refinement and horizontal specialization of DSL (N).

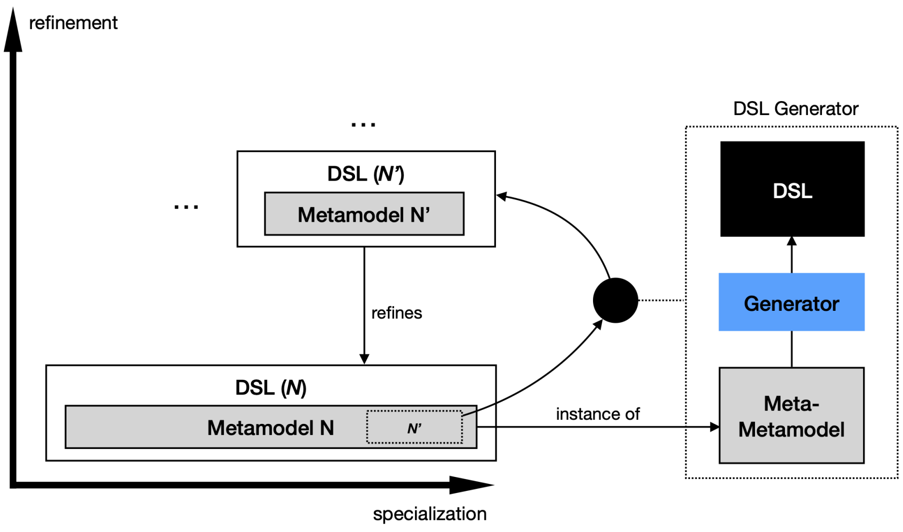


Figure 3.3: DSL specialization and refinement by utilizing a aligned DSL Generator.

which contains all possible nodes and edges. At the same time, the DSLs N', N'' and N''' can be further refined to provide even more specific support for a user.

To realize the refinement and specialization necessary for LDE, the environment must allow the reuse of a DSL to create a new one. Fig. 3.3 shows the conceptual structure of such an environment using the example of *DSL* (N). The key to reuse is the central *DSL Generator* as an archimedian point [SN16], which generates a new DSL for an instance of the *Meta-Metamodel*. Since all DSLs are instances of the meta-metamodel, they can be

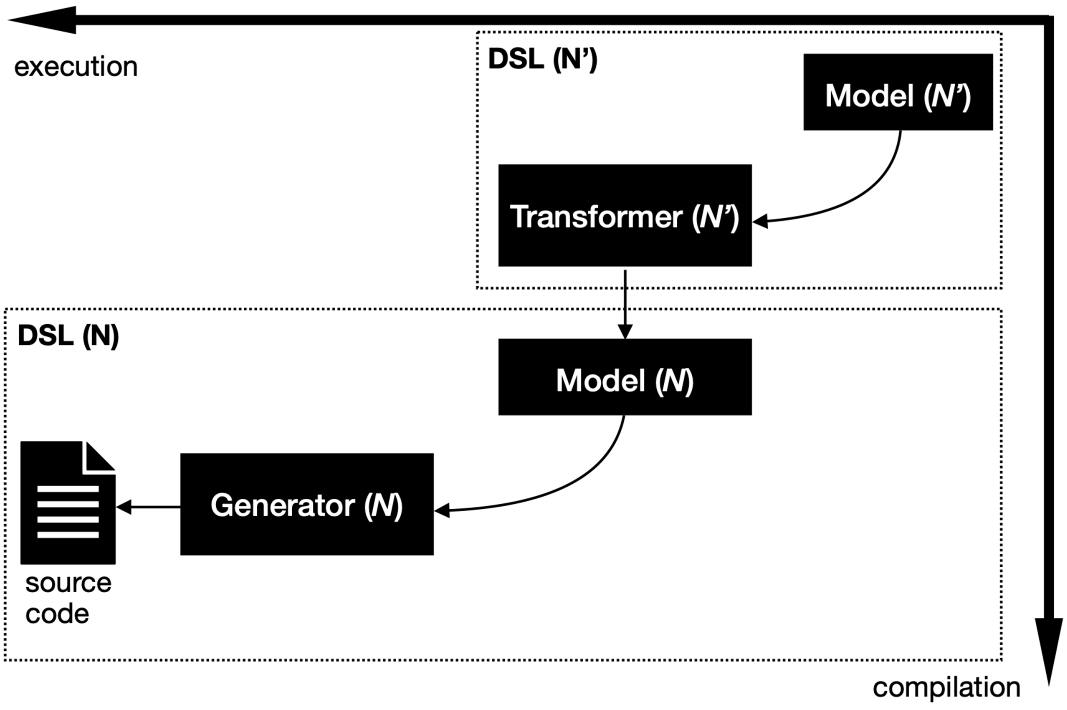


Figure 3.4: Concretization of DSL models by stepwise transformation and final code generation.

refined using the same *DSL Generator* by taking a part of the respective DSL as input. Fig. 3.3 illustrates this procedure using the *DSL* (N), which is refined by the partial aspect (N'). The corresponding part of the metamodel of the *DSL* (N) serves as input for the *DSL Generator*, so that the refined *DSL* (N') can be generated. Thanks to the central *Meta-Metamodel* and corresponding *DSL Generator*, continuous refinement and specialization can be realized within the LDE environment.

3.3 Concretization

Execution of a model within the LDE environment requires successive vertical concretization and horizontal generation of executable source code. Conceptually, concretization requires the transformation of a refined DSL to the original more general DSL. Since concretization and refinement are not idempotent, it is necessary to define a corresponding transformer for each DSL.

Fig. 3.4 illustrates the concept of concretization using the example of a *DSL* (N') which refines *DSL* (N). In order to execute an instance of *DSL* (N'), a transformation to the *DSL* (N) must be performed. The transformation requires the service-oriented instrumentation of *DSL* (N) to create the *Model* (N). Afterwards, the transformed model can be used by a code generator of the *DSL* (N) to obtain executable source code.

The general realization of this concept requires the possibility that each DSL can be both target and source of a transformation. Accordingly, each DSL within the LDE en-

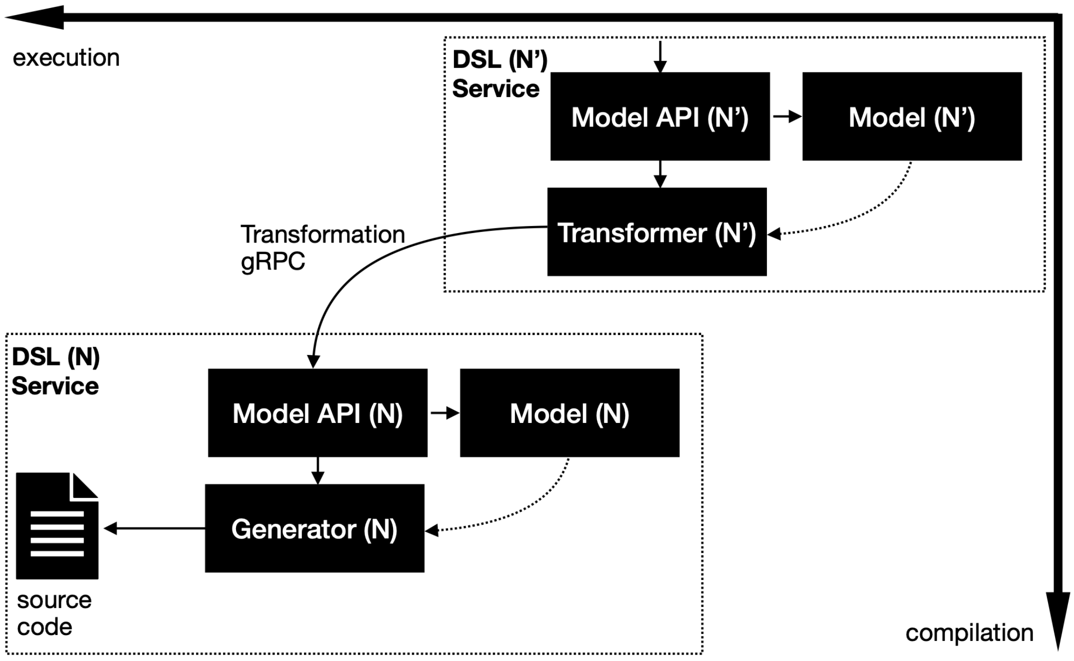


Figure 3.5: Service-oriented stepwise model-to-model transformation and generation.

vironment must represent a service that can be used by other DSLs. The *Model API* described by the tool template provides the necessary interface to create an instance of a DSL within a transformation. At the same time, further transformations to an even more concrete DSL can be initiated recursively to finally generate code from a model.

In order to perform a transformation, a model must be traversable as described in [KLNS21]. For this purpose, each DSL within the LDE environment contains the *Model API* generated from the metamodel (cf. [Lyb19]) to enable a simplified transformation.

Fig. 3.5 illustrates the unified structure using the example of *DSL* (N') which has to be transformed to a *DSL* (N). According to the concept, each DSL is an independent service which accepts transformation requests as remote procedure calls by the gRPC protocol [13]. As described in the tool template, each DSL provides a corresponding *Model API* called by *Transformers* to remotely create instances of a DSL. In this way, the *Transformer* (N') translates the *Model* (N') to an instance of the *DSL* (N) using the *Model API* (N). Thanks to the underlying framework of each *Model API*, subsequent transformers or generators are triggered automatically. As illustrated in Fig. 3.5, the *Model API* (N) of *DSL* (N) is used to execute the associated *Generator* (N) which generates the source code.

Since each DSL within the environment is generated by the same DSL generator, the model API, transformer interfaces and generator interfaces can be generated automatically and specifically for a given DSL, so that the effort to realize transformation chains is reduced to a minimum.

Chapter 4

Collaboration Concept

Collaboration plays a crucial role in the concept of LDE, since various users are involved in the development and usage of DSLs. The target group of LDE is not limited to classical software developers but to any domain expert, who should not be hindered by classical text-based collaboration frameworks like VCS. Instead, an intuitive and fully integrated real-time collaboration system should be used, to enable a way of working as if all persons are in the same room. To ensure such a level of collaboration, it is necessary to enable multiple users to work on the same model of a DSL at the same time without hindering each other.

Additionally, the environment must be able to manage the multitude of users according to their expertise with a tailored interface . This means that different DSLs are available for each user by assigning special access rights. This results in a customized view to supports a user by necessary languages only and the possibilities to optimally collaborate on a project.

In contrast, classical programming environments offer only version control systems well-known from the software development for the central management of the project's resources. The concept of the VCS is based on the asynchronous propagation of changes, which must be synchronized before by each user manually. Even if this way of working is established for software developers, it is not intuitive for domain experts from other disciplines [DRFMM17] and made collaboration difficult. In addition, LDE is based on the use of graphical DSLs which could only be managed with the help of a textual serialization in a VCS. As noted by [REIWC18], change detection in serialized models is imprecise with respect to the cause of a conflict and unintuitive since the textual representation must be repaired unlike the graphical model. For these reasons, the use of classical VCS is unsuitable for the LDE concept as described in [ZBS22].

In this chapter, the concept of collaboration within an LDE environment is explained. First, Sec. 4.1 explains how multiple users can work simultaneously on a model without the need for a VCS. Sec. 4.2 then illustrates how specialized views can be defined to provide each user with a customized access to the environment.

4.1 Real-Time Editing

Simultaneous collaboration during parallel processing of a model by different users requires the use of a distributed system and bidirectional communication channels. In order to synchronize the interactions with a model in real-time between the users, an availability-focused communication model must be implemented in which users cannot slow each other down or block each other. At the same time, consistency must be maintained by ensuring that all users are editing the same state of a model.

As already shown by Pyro [ZNS19], consistent and simultaneous collaboration can be achieved by using the basically available soft-state (BASE) [Vog09] communication model in a client-server architecture [DS05]. The key to this approach is the initial distribution of the model to all users and a subsequent propagation of changes. To ensure consistency, communication takes place exclusively via the server, which manages the models as a central single-point-of-truth (SPOT) [PS14] and executes each received change in an ACID-compliant transaction [GR92]. The operations-based concept of change distribution uses context-free replica types (CRDTs) [SPBZ11] to serialize interactions from a user to a model so that only the relevant information is distributed. In this way, CRDTs avoid the ambiguous computation of deltas of a state-based approach.

With regard to graphical graph-based languages, the following CRDTs thus exist, which are based on the Eclipse graphical Language Server Protocol (Eclipse-GLSP) [8]:

- **Create Node:** Describes the creation of a node with initial position, the parent container and the corresponding type.
- **Move Node:** Describes the moving of an existing node to a new position and a corresponding parent container.
- **Resize Node:** Describes the enlargement of an existing node by a new height and width.
- **Delete Node:** Describes the deletion of an existing node.
- **Create Edge:** Describes the creation of an edge with initial source node and destination node, as well as the corresponding edge type.
- **Reconnect Edge:** Describes the change of the source node or destination node of an existing edge.
- **Bend Edge:** Describes adding or removing corners on an existing edge.
- **Delete Edge:** Describes the deletion of an existing edge.
- **Edit Properties:** Describes the modification of the attributes of an existing edge or node.

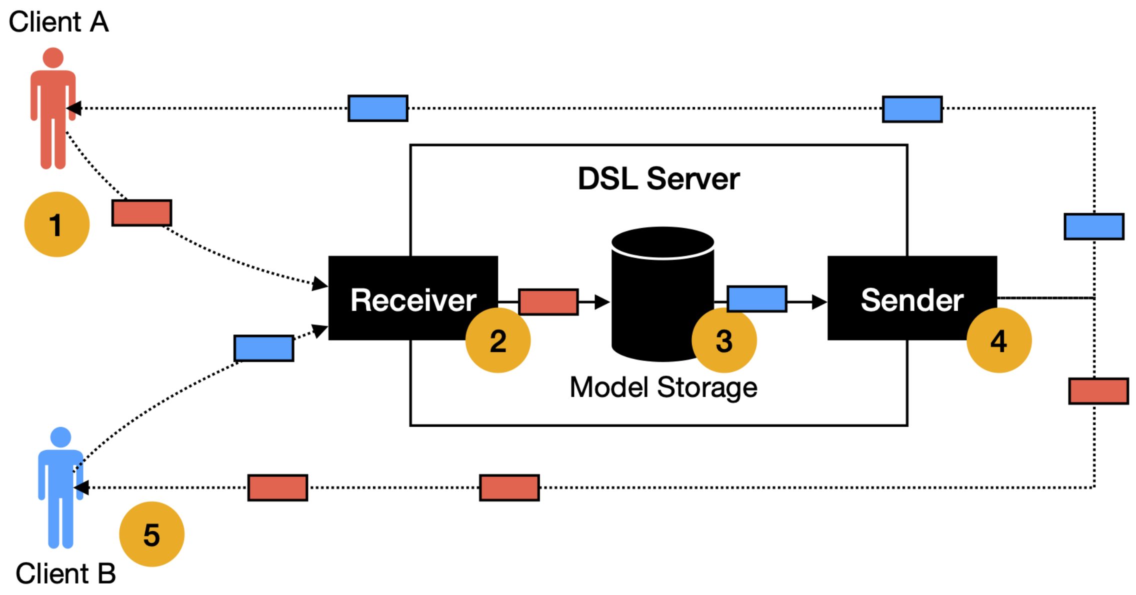


Figure 4.1: Operation-based change propagation between Client A and Client B using a central DSL Server.

The distribution of the changes is then done in several steps which are illustrated in Fig. 4.1 using the example of *Client A* (red) and *Client B* (blue). After the initial distribution of the centrally stored model to each client, the users can edit the model locally. Each change is serialized in a CRDT and sent as a message from the client to the server (see Fig. 4.1 *Point 1*). The server's *Receiver* (see Fig. 4.1 *Point 2*) receives the sent messages and checks if the CRDT is valid based on a predefined API. This check includes both the formal correctness of the given information and a check for concurrent modification as described in [ZBS22]. In case of a conflict, the CRDT is inverted and sent back to the sender, so that the modification of the client's local model is revised.

After a successful check, the change serialized in the CRDT is applied to the central *Model Storage* (see Fig. 4.1 *Point 3*) so that at this point the central model is equal to the client's model. Each successfully applied CRDT is distributed to all clients also working on the model using bidirectional communication channels of the *Sender* (see Fig. 4.1 *Point 4*). Clients receiving a message from the server apply the serialized change locally (see Fig. 4.1 *Point 5*) in each case, so that the write-repair mechanism of BASE [TTP^{+}95] is realized and global consistency can be achieved, as described in [ZBS22].

As already shown in [ZBS22], the collaboration can be improved with respect to a semantic conflict detection. For this purpose, the syntactic and semantic information within the metamodel of a given DSL is used to influence the validation process of the *Receiver* (see Fig. 4.1 Point 3). In this way, unnecessary conflicts can be avoided and semantic conflicts are detected early.

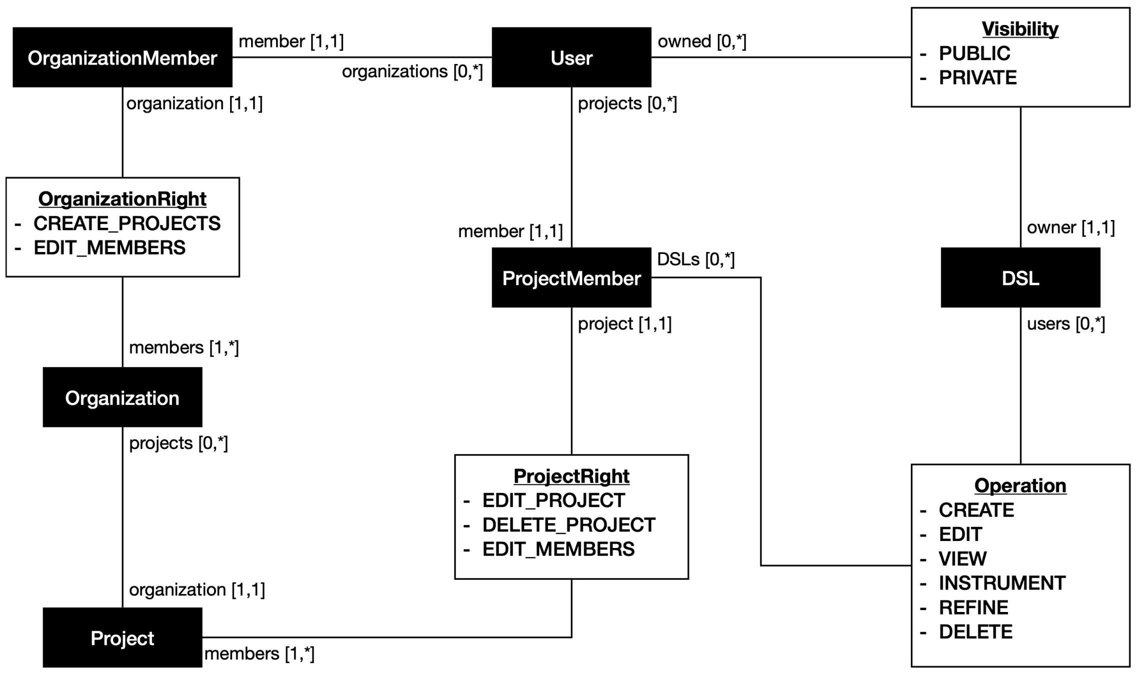


Figure 4.2: Management data model including the view definition for users of the environment.

4.2 View Definition

In contrast to general programming systems, the LDE environment is tailored to the needs of each individual user. For this reason, concepts for higher-level administration and grouping must be realized, which is similar to Gitlab [11], Github [10] and Bitbucket [2] with the help of organizations and projects. To ensure that users only have limited access to the resources and functionalities, authorization structures must be created. In this way, unauthorized access is avoided and users are not overwhelmed by unnecessary functionalities.

An additional challenge is posed by the development and deployment of new DSLs available in the LDE context. The environment must allow language developers to publish a new DSL so that it can be used by others. Fig. 4.2 illustrates the view definition capabilities of the LDE environment using a schematic data model that includes types (black), bit-vector association types (white), and bidirectional associations (black edges). Each bidirectional association is named at the ends of the edge for the respective type and given a cardinality. The view definition shown here is only meant to explain the concept and accordingly only gives an insight into the possible reassurances without a claim to completeness.

The starting point is the *User*, which associates the *OrganizationMember* and *Project-Memeber* types. With the help of the *OrganizationMember* type a user can be assigned to an *Organization*. The bit vector association type *OrganizationRight* describes whether a member is able to create projects within the organization or to edit other members inside the organization. The *EDIT_ MEMBERS* right allows the corresponding user to change the rights of all other members. By default the creator of an organization receives all rights to manage other members and the contained projects.

The projects of the organization are managed by the *projects* association, containing an individual list of members each. A corresponding *ProjectMemeber* instance assigns a *User* to a *Project*, to manage all affiliations. The association between a *Project* and its members is extended by the *ProjectRight* bit vector association type. The vector decides if the respective member is allowed to edit and delete the project. Changing the permissions of other members within the project is specified by the *EDIT_ MEMBERS* flag.

The management of the available DSLs within a project is bound to the *ProjectMember* type by using the *Operation* bit vector association type. For this purpose, it is possible to specify for each member which operations are available for a given *DSL* and which are not. For example, the *CREATE* flag can be used to specify that only certain members can create instances of a given DSL. The *EDIT*, *VIEW* and *DELETE* rights analogously limit the access to the models of the associated DSL.

Whether a created DSL can be used by other users at all is determined by the *Visibility* bit vector association type. Users who have created a new DSL are noted as owner and can decide how it may be used. The *PRIVATE* flag does not allow access so the DSL for users. To prohibit the manual use of the DSL but allow it's use as a transformation target the *INSTRUMENT* flag can be set by the owner. The *PUBLIC* flag, on the other hand, allows both instrumentation and manual creation of instances of the DSL.

Thanks to the simple data model for managing the access control and view definition, even technically inexperienced users can administer their projects and organizations.

Chapter 5

CINCO Cloud

This chapter describes the realization of the LDE environment called CINCO Cloud, which implements the concepts and requirements described above. The CINCO Cloud combines the approaches of the CINCO Meta Tooling Suite for the specification and generation of graphical domain-specific languages and the Cloud IDEs for unified development and deployment in a distributed system. To enable collaborative development within the CINCO Cloud, Pyro's synchronization architecture is used, whose optimistic communication model provides real-time collaboration.

Key to the efficient implementation of LDE is the unified and distributed system of the CINCO Cloud, so that each language by design represents a modular service that can be instrumented via standardized protocols and APIs. Thanks to the language composition capabilities within the environment, the various requirements of language-driven engineering can be realized. To this end, the CINCO Cloud focuses on the following scenarios:

- **Language Serving:** Each DSL created within the environment is stored and managed centrally and can subsequently also be used in the same environment. With the help of additional rights management, authorized users can reuse the DSLs of others.
- **Language Generation:** New DSLs are created using the Pyro Product Generator so that each DSL is an independent distributed system. The DSL is described and extended using CINCO's established specification languages and APIs. Thanks to the uniform generation of all DSLs utilizing the same generator within the environment, refinement, specialization, transformation and execution can be realized uniformly for each language, thus simplifying composition.
- **Language Transformation:** The model-to-model transformations translate from a specialized to a more abstract DSL to gradually reach execution. Thanks to the language-as-a-service structure of the CINCO Cloud, the target language of the transformation can be dynamically instantiated and instrumented. In this way, the transformation is performed through specialized APIs using remote procedure calls instead of a detour through a serialization or interchange format.

- **Language Execution:** The execution of a model within the CINCO Cloud is realized by a model-to-text generation which creates executable code. Thanks to the dynamically scalable infrastructure, necessary runtime environments to serve the generated code can be instantiated and used directly within the same environment.

The following Sec. 5.1 first describes the system architecture and infrastructure of the CINCO Cloud based on the Cloud IDEs. Sec. 5.2 illustrates the DSL Module, which enables the use of DSL in a unified workspace. The administration of the environment including users, projects and system resources is explained in Sec. 5.3. Subsequently, Sec. 5.4 shows how the DSL generation and delivery scenario is realized. Sec. 5.5 and Sec. 5.6 conclude by describing the step-by-step transformation and generation for DSL model execution within the CINCO Cloud.

5.1 Environment Architecture

To meet the various requirements, the CINCO Cloud must be able to manage a large number of users, services, organizational structures and projects. Each project symbolizes a workspace, which in turn consists of a selection of DSLs.

In contrast to a classic programming system, the CINCO Cloud should be device independent and platform independent so that a installation-less usage is possible. This means that all necessary resources for development, use and execution of the DSLs must be provided within the environment. Nevertheless, each user should be able to work in isolation in the CINCO Cloud without being affected by others. Fig. 5.1 illustrates the environment architecture of the CINCO Cloud and clarifies the components involved.

Cluster

The infrastructural foundation for the CINCO Cloud environment is realized by a *Cluster* based on Kubernetes [17]. The cluster allows the orchestration and scaling of isolated containers. In contrast to classical virtual machines, container virtualization [And15] does not require operating systems within each container, as the so-called container runtime (e.g. Docker Daemon) is the abstraction layer to the operating system core.

Each container is based on an image, which can be built using a framework such as Docker [4]. A container image describes the setup of a concrete runtime environment, starting with the operating system and ending with the application. In the context of the CINCO Cloud, an image describes a corresponding service such as the workspace (see Listing A.1 and A.2), which can then be executed dynamically as a container.

In the context of Kubernetes, the containers are instantiated in *Pods* [Luk18] which can be dynamically managed and scaled. The CINCO Cloud uses the pods to execute the individual services within the environment. At the same time, the Kubernetes-based infrastructure offers the possibility that the services within the pods can communicate with each other to realize a unified micro-service architecture [AAE16].

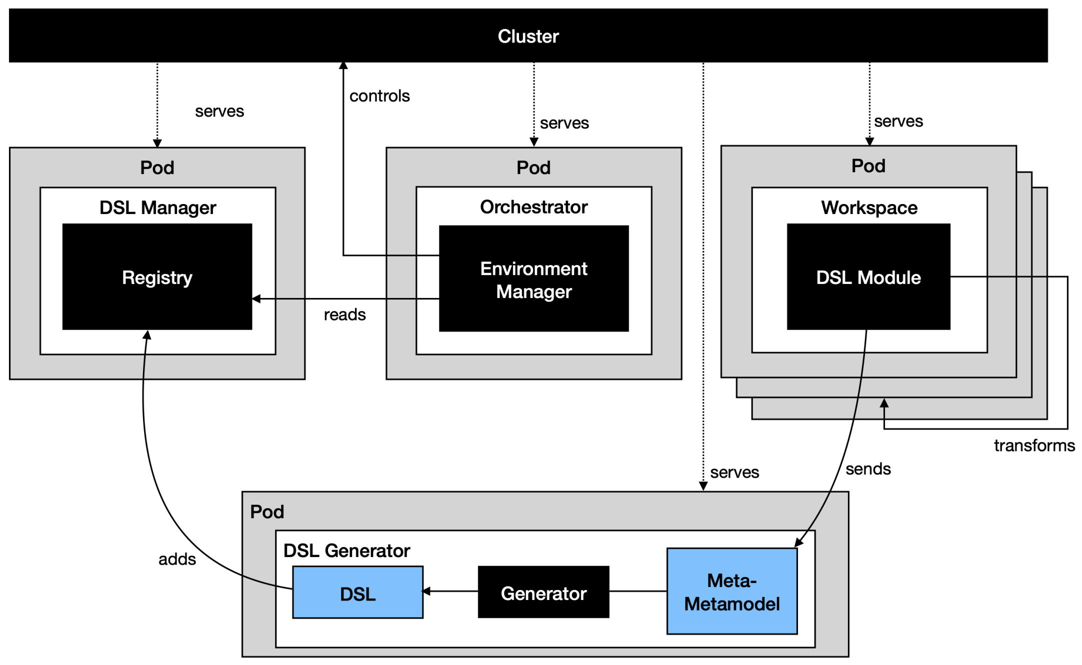


Figure 5.1: CINCO Cloud environment architecture overview.

Orchestrator

The cluster is controlled according to the master-slave architecture [Luk18] with the help of an orchestrator which manages all pods. Similar to the master application of the Cloud Development IDEs (see Sec. 2.5), the orchestrator uses the APIs of the cluster to control the resources. Based on a resource monitoring, integrated schedulers can disable unused pods or scale heavily used ones to distribute the load. At the same time, the orchestrator offers the possibility to create new pods, for example to provide more workspaces.

Within the CINCO Cloud, the *Orchestrator* service is extended by the *Environment Manager*, which includes the management of users, access rights, projects and organizations. All other pods can access the environment management via corresponding interfaces, for example to perform authentications. The definition of views for an individual user can also be performed within *Environment Manager* by implementing the authorization and access structures described in Sec. 4.2.

The central task of the *Environment Manager* is the creation of new workspaces by instantiating a container from an image and executing it in a new pod. In this way, the CINCO Cloud is able to automatically and dynamically provision new workspaces with any DSLs by having all the necessary runtime specifications predefined within the image.

DSL Manager

The management of all DSLs present in the environment is supported by the *DSL Manager* service, which also runs inside a pod within the CINCO Cloud. The *DSL Manager* uses a *Registry* to store and version all built container images. Since each image is hierarchical and contains a large number of dependencies (see Listing A.1 and A.2), the *DSL Manager* uses a dependency cache to speed up the build time.

In order to use the DSLs developed in the context of the CINCO Cloud in a new workspace, each DSL must first be added to the *Registry* of the *DSL Manager*. For this reason, the *DSL Manager* service includes an additional image builder process that prepares the runtime environment and then adds it to the *Registry*.

Workspace

The Workspace allows one or more users to collaborate on a project. Technically, the workspace is an Eclipse Theia [9] based application that allows editing the isolated memory of the pod with a browser based IDE.

In the context of the CINCO Cloud, each *Workspace* consists of a number of different *DSL Modules*, which are implemented as Theia plugins. Each *DSL Module* combines the components of a DSL as described in Sec. 3.2. Thanks to the modular structure of Ellipse Theia, additional DSL modules can also be added later during runtime. In this way, all users gain platform-independent access to any DSL within the CINCO Cloud without the need for manual setup, dependencies and devices.

The development of new DSLs is also provided by the *Workspace Service* by using the metamodel specification languages of CINCO (see Sec. 2.2). To enable the Xtext specification languages inside the CINCO Cloud, corresponding textual DSL modules are used. Thanks to the service-oriented environment, a newly designed graphical DSL can be transferred to the *DSL Generator* via API.

The execution of models as DSL instances is realized by stepwise transformation (as described in Sec. 3.3) orchestrating different more concrete *DSL Modules* within the CINCO Cloud. For this purpose, each *DSL Module* has a corresponding model API which allows the creation of instances in a service-oriented fashion . The necessary underlying *DSL Modules* can be instantiated as needed.

DSL Generator

The *DSL Generator* is a central service within the CINCO Cloud environment, which can also be scaled by the cluster. DSLs are realized according to the CINCO concept (see Sec. 2.2) using the full generation approach [NLKS18], which generates the entire tool based on a language description.

In the context of the CINCO Cloud, this generation step is outsourced and centralized so that it can be used through a service interface. The input contains the metamodel of the DSL in the form of an abstract and concrete syntax description as well as additional components, which are combined into a complete *DSL Module* with the help of the *DSL Generator*.

After generation, the *DSL Generator* uses the interfaces of the *DSL Manager* to initialize the build process of the new *DSL Module* including a runtime description for the

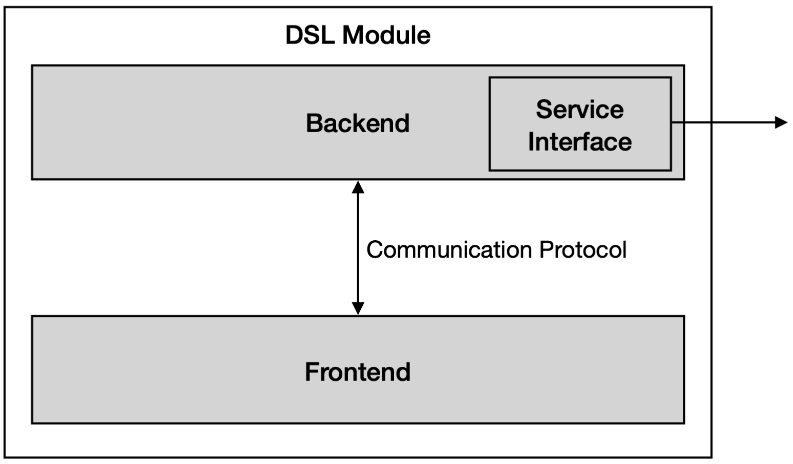


Figure 5.2: DSL Module client-server application to serve a workspace inside the CINCO Cloud.

container image and to be added to the *Registry.* In this way, a new DSL is automatically made available and to be used in the CINCO Cloud.

5.2 DSL Module

The *DSL Module* is a distributed client-server application to extend a *Workspace* within the CINCO Cloud (see Fig. 5.2). Different users can access the *Workspace* simultaneously and thus get access to the predefined general purpose programming languages, textual DSLs and graphical DSLs in a unified interface based on Eclipse Thea. A *DSL Module* contains a back-end application to provide language specific features including a persistence layer and a *Service Interface* for other pods within the CINCO cloud. The front-end includes a client application that provides an editor for the browser.

5.2.1 Back-End

The back-end of the *DSL Module* (see Fig. 5.3) combines the different languages available in the workspace so that the client can access the necessary features. Each language is represented by a corresponding language server [Bün19] that provides functions for e.g. syntax highlighting, validation and auto-completion. The communication protocols used between client and server are language independent so that the editor does not have to be adapted for each language. The clear separation between the client and the language server enables the usage of different editors and platforms for a language.

The persistence of the language files is also handled exclusively within the back-end of the *DSL Module.* Thanks to the isolated file system of a workspace pod within the CINCO Cloud, users of different workspaces can act and manage their files independently from each other. As soon as a file is opened, the back-end checks whether a corresponding language server is available for the selected file type and then applies the corresponding function-

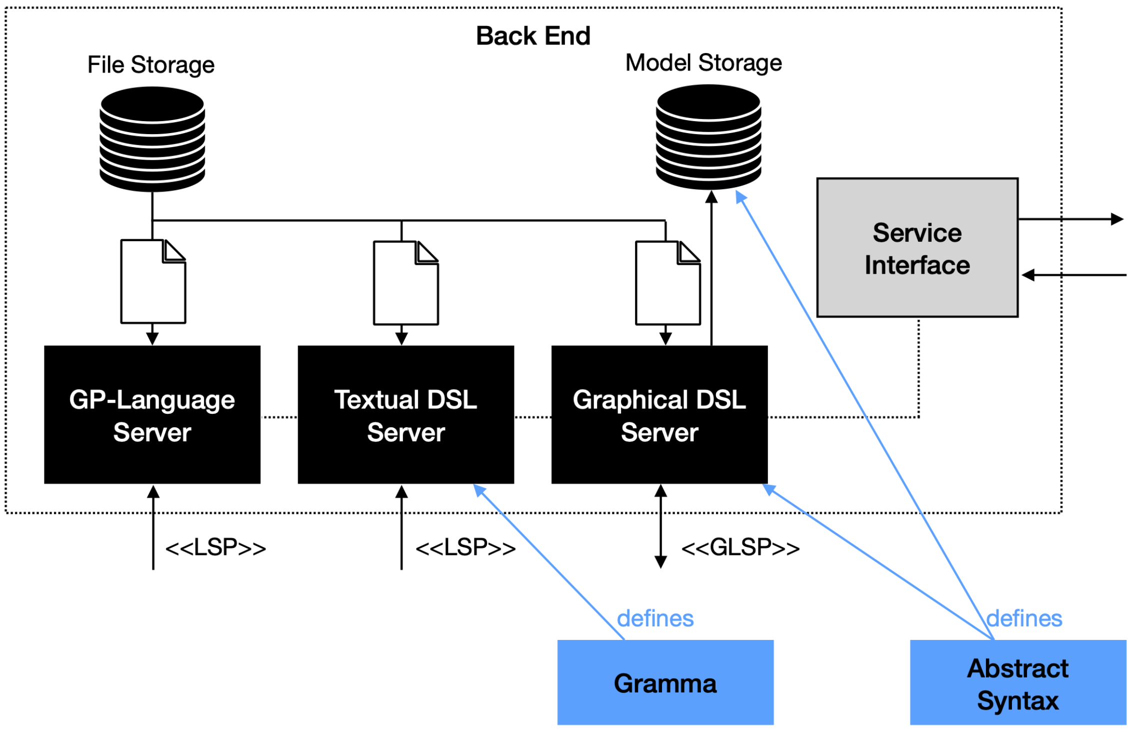


Figure 5.3: Backend of a DSL Module combining language server, persistence and the service interface.

alities. The DSL module of the CINCO Cloud offers three different types of language servers:

GP Language Server

Developing DSLs with CINCO requires implementing components such as validators, transformers, generators, and actions (see Sec. 3.1) to provide additional language features and functionality to a user. The components are integrated into the generated tool within the subsequent generation step. For this reason, in addition to DSLs, classical general purpose programming languages such as Java are also required within the CINCO Cloud. There is already a large number of available language server implementations [19], which can be integrated into a *DSL Module* if necessary in order to provide additional programming languages within the workspace. The *GP Language Server* utilize the standardized Language Server Protocol (LSP) [20] to provide the language specific features for the editor in the front-end.

Textual DSL Server

To enable the use of new textual DSLs within the CINCO Cloud, corresponding previously generated language servers can be integrated into a DSL module. In this way, CINCO's specification languages MGL, MSL and CPD (see Sec. 2.1) can be used within a workspace to develop new graphical DSLs. Thanks to the Xtext framework [Bet16], the textual DSL servers can be generated starting from a description of the *Grammar* as an extended BNF [Sco93] (see blue arrow of Fig. 5.3). From a technical point of view, the generated *Textual DSL Servers* work analogously to the *GP-Language Servers* by accessing the files stored in the back-end and providing features via the standardized communication protocol. The *Textual DSL Servers* use the same language server protocol as the *GP Language Servers*, so that the same text editor can be used in the front-end to use the DSL features.

Graphical DSL Server

A *Graphical DSL Server* represent the result of a graphical DSL previously developed within the CINCO Cloud by using the abstract syntax described by the metamodel (see blue arrow of Fig. 5.3). The language server combines the generated components with the manually implemented extensions and makes the functions available to the client via a special protocol.

The *Graphical DSL Server* differs from the textual language servers by managing a structured graph-based model instead of text. In contrast to textual languages, a graphical model corresponds to the underlying metamodel at all times, which is why the graphical language server only allows syntactically correct interactions by a client. For this reason, the Graphical Language Server Protocol (GLSP) [8] is used with bidirectional communication which represents structural changes by a user as described in Sec. 4.1.

To preserve the advantages of the existing structure the graphical models are not stored serialized in a file but within the *Model Storage* realized as a relational database. The schema of the database is generated depending on the metamodel definition of the DSL (see blue arrow of Fig. 5.3) so that an efficient management of the instances is possible. In this way, references between nodes and edges within a model and between different models can be mapped, read and validated consistently. Thanks to the ACID compliant transaction mechanism of the database, a consistent and semantically correct collaboration can be realized in real-time as described in [ZBS22].

To use the already existing language independent features of the Theia editor, each instance of a graphical DSL, is still represented by a file. These files serve as a reference to a model within the *Model Storage*, so that the editor's file management functions can still be used. As soon as a model file is opened by the user, the corresponding graphical DSL server loads the corresponding information from the database and delivers it to the client-side editor instead of the file content.

5.2.2 Service Interface

Besides the different language servers, the back-end contains APIs to use the services of the CINCO Cloud. Since the user management and authentication within the environment is realized by the orchestrator, the *Language Servers* can check via the *User API* whether the current user is authenticated and authorized for the workspace. In addition, the view

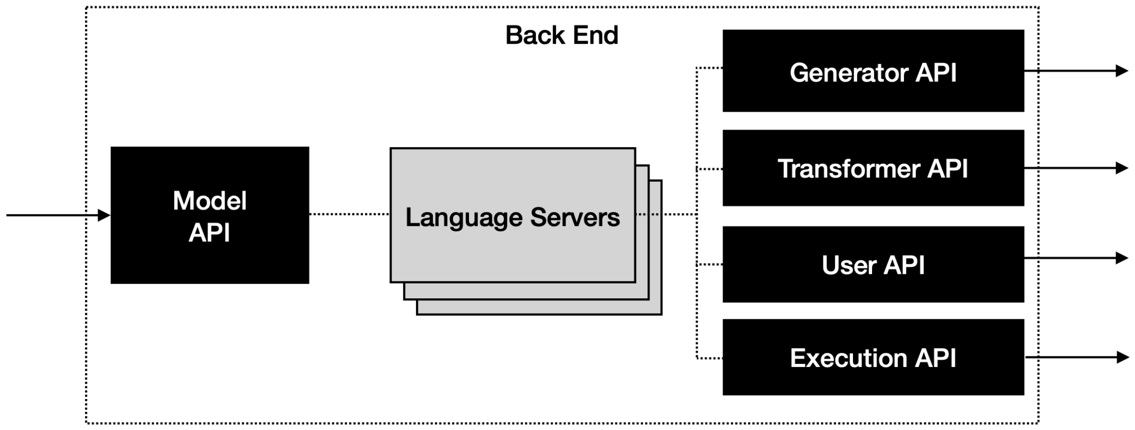


Figure 5.4: The service interface components for the communication between different pods inside the CINCO Cloud.

definitions available in the orchestrator can be queried so that the customized selection of languages and features can be provided for a user.

The *Generator API* provides an interface to the *DSL Generator* so that new DSL specifications and additional components within the workspace can be transferred easily. The DSL generator can then create the new DSL module and register it within the DSL Manager as described in Sec. 5.4.

To enable the execution of instances of a DSL via step-by-step transformation, the back-end includes the *Transformer API* interface. It allows access to a target DSL module to instrument the local language server as transformation target. For this purpose, each DSL module includes a model API that standardizes access within the environment and thus supports the remote creation of models of a given language.

The *Execution API* provides the possibility to execute the manually written or generated source code within a workspace. For this purpose, the specification of the runtime environment and the code is passed to the interface, which then creates a new workspace for execution.

5.2.3 Front-End

The *Front-End* of a DSL module (see Fig. 5.5) contains an editor based on Eclipse Theia that gives users access to the resources and languages od a workspace. Fig. A.1 showed a screen shot of the editor within a browser. Due to the modular structure of Eclipse Theia, the editors can be used with a browser as well as established IDEs like Eclipse and Visual Studio Code. In the background the editor communicates with the DSL module and its language servers to execute the language specific features.

For the use of textual languages the *Monaco Editor* [21] is used which offers syntax highlighting, code completion, hints, reference tracking and validation. The necessary

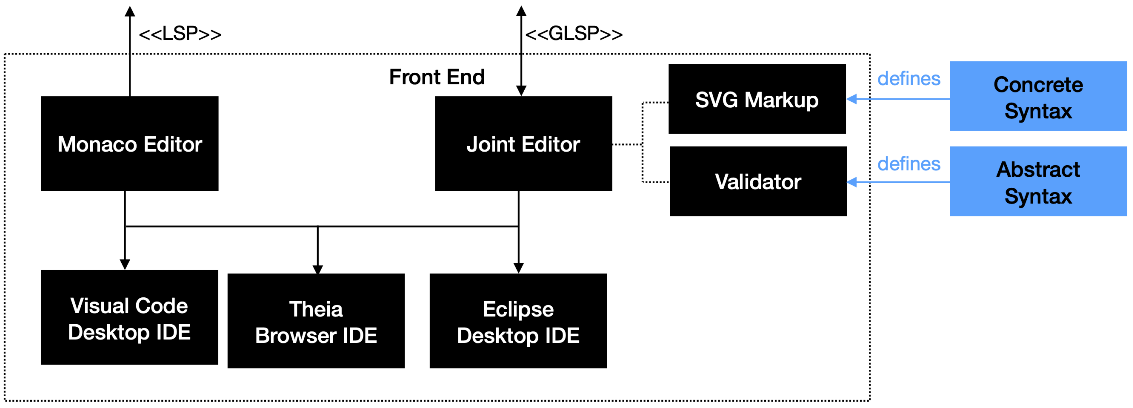


Figure 5.5: DSL Module front-end providing the workspace editor interface for all available languages.

information is requested via language server protocol, so that the editor can act independently of the language's grammar and does not need to be specialized.

In contrast to textual languages, graphical DSLs offer a specialized concrete syntax which is highly customized. A generic graphical editor would therefore have to load not only the structural information of a model but also all shapes and appearances. This complete transfer causes a high communication overhead with regard to the amount of data, because the editing of a model resembles a screen-cast. For this reason, the editors for graphical languages are generated within the CINCO Cloud depending on the language definition and integrated into the front-end (see blue arrow of Fig. 5.5),. In addition, the full generation of the graphical editors allows the syntactic validity of the models to be checked directly in the front-end and avoids unnecessary communication of invalid interactions. For this purpose, the abstract syntax of the DSL is used to generate a client-side validator.

The basis for the graphical editors is the JointJS framework [14] which allows the interactive manipulation of SVGs in the browser. To visualize the nodes and edges specified in the concrete syntax, *SVG Markups* are generated (see blue arrow of Fig. 5.5), which are used by the *Joint Editor* to display the corresponding shapes and colors. In addition, the editor provides components such as the node creation palette, manipulation menus and validation markers.

5.3 Environment Manager

The *Environment Manager* is delivered within the central orchestrator pod as a client-server application. Fig. 5.6 illustrates the system architecture of the *Environment Manager*, which is divided into a back-end and a front-end. Each initial access to the CINCO Cloud is handled by the *Environment Manager*, since the overarching structures and the cluster-based infrastructure are managed by this service. The DSL modules of the

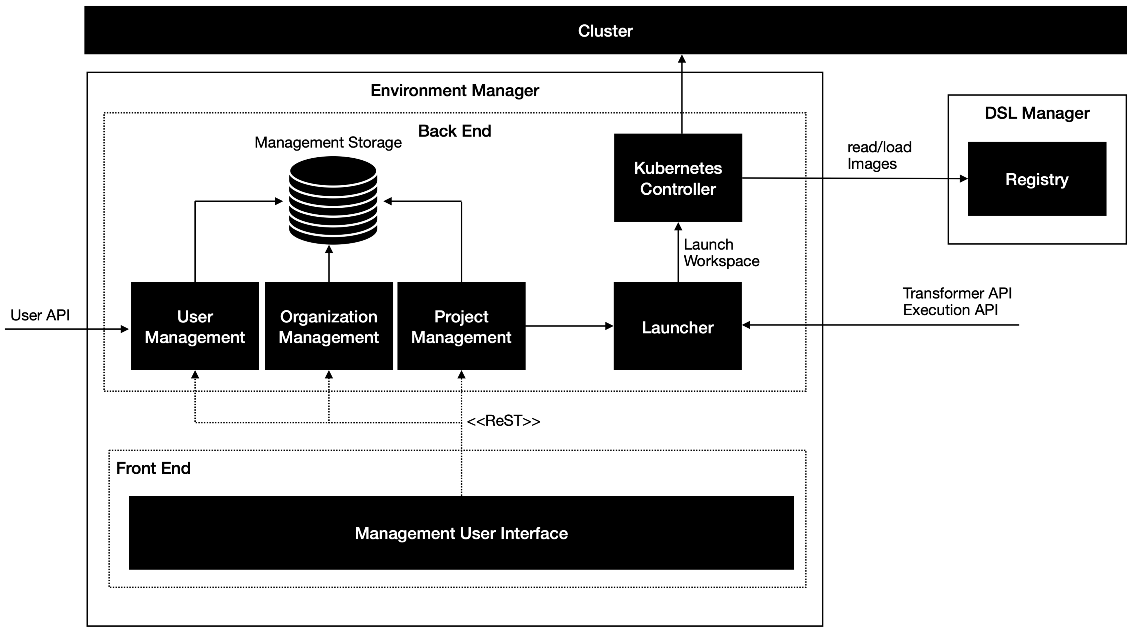


Figure 5.6: Environment Manager system architecture and interface overview.

workspace pods can use the existing interfaces to access the user-base and infrastructure services. The individual components and features are explained below.

5.3.1 Back-End

The back-end of the environment manager administrates the users, access rights, projects and organizations of the CINCO Cloud in a relational database, which is structured as described in Sec. 4.2. The *User Management* provides an additional interface which is available within the *Cluster* to provide the workspace pods with an authentication and authorization API. In this way, a DSL module within a workspace requests information such as the view definition to provide a user specific editor. To manage the organizations, projects and members, the environment manager provides a ReST API [Fie00] which can be used by the front-end.

Besides the management tasks, the control of the cluster-based infrastructure is the preformed by the environment manager. Once a user opens a workspace, the *Kubernetes Controller* is utilized to provide a corresponding pod. To create a pod, several Kubernetes APIs are instrumented:

1. **Storage Allocation:** Each pod requires an isolated storage to store the resources of the project. The `etcd` controller provides an abstraction layer to the file system and creates a folder that can be mapped to the pod afterwards, giving the workspace direct access.

2. **Pod Creation:** To create a pod, a corresponding configuration is first created, which combines the DSL modules required for a workspace. For this purpose, the

Registry available in the *DSL Manager* is used to get access to all images previously developed in the CINCO Cloud. Then the new pod can be started by the `kubectl` [15] controller to add the configuration to the scheduler [18].

3. **Pod Publishing:** Once the pod is deployed, the `kube-proxy` [16] controller is used to assign an address to the pod so that users can access the new workspace.

The step-by-step transformation of a DSL model requires all dependent DSLs to be available as a transformation target. Technically, corresponding DSL modules are dynamically deployed as pods during the transformation as needed. For this reason, the environment manager provides an interface that can be used by the *Transformer API* and *Execution API* of each DSL modules. Based on a given description of the required target DSL module, the *Launcher* component of the environment manager launches a matching pod. Subsequently, the transformation is performed using the *Model API* of the DSL module as described in Sec. 5.2. Thanks to this dynamic provisioning of DSL modules, any transformation chains can be performed uniformly inside the CINCO Cloud.

5.3.2 Front-End

The front-end of the environment manager contains the *Management User Interface*. Users of the CINCO Cloud must first register and log in via the front-end's interfaces to gain access to the environment. Within the environment there are different administration pages which can be used depending on the user's permissions to manage organizations, projects and other users. The administration is based on the data model described in Sec. 4.2 for the definition of user specific views.

After successful login, the user gets an overview of all owned or shared organizations. As soon as a user selects an organization all projects with an existing membership are listed. If the appropriate permissions are given, new projects can also be created.

The creation of a new project is done via the interface shown in Fig. A.2, which allows the selection of a workspace type. The corresponding modal dialog lists all DSL modules globally released inside the CINCO Cloud or created within the organization. As soon as a user starts or opens a project, the corresponding workspace pod is instantiated, all required DSL modules are loaded from the *Registry* and the user is redirected to the editor.

From a technical point of view, the front-end is implemented as a single-page web application using Angular [1]. All modifications are transferred to the back-end via ReST API and stored centrally there.

5.4 DSL Creation

The ability to create and refine graphical DSLs plays a crucial role in realizing the language-driven engineering approach. In this way, any user can be supported by a customized DSL at the appropriate level of abstraction, depending on their existing expertise. The challenge

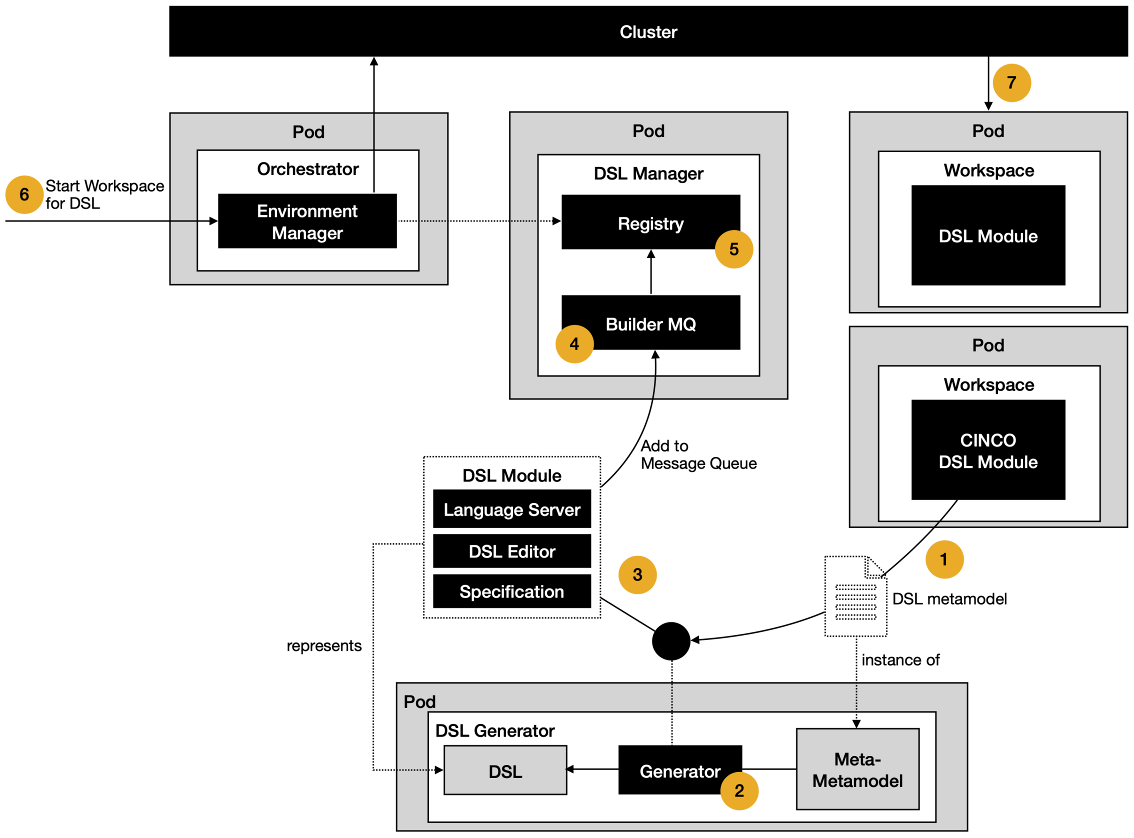


Figure 5.7: DSL generation and release process.

of this approach lies in the multiplicity and variation of the different DSLs, in that efficient creation and direct delivery is required.

Thanks to the unified service-oriented architecture of the CINCO Cloud, this requirement can be met by creating, centrally managing and directly using DSLs in the same environment. Fig. 5.7 illustrates an example process for creating a new graphical DSL up to its usage. The individual steps and involved components of the CINCO Cloud in the creation and use of a new DSL are explained step by step below.

(1) DSL Specification

As described in Sec. 2.2, the specification languages established by CINCO are used as DSL modules to describe the abstract and concrete syntax of a DSL. After the metamodel of the DSL has been developed in this way, the generation can be triggered. The *CINCO DSL Module* contains a corresponding button that is displayed in the editor and starts the language generation process.

(2) DSL Generation

The *DSL Metamodel* is transferred in combination with the language-specific components such as validators, actions, transformers and generators via a corresponding interface to the central *DSL Generator* service. Thanks to the uniform *Meta-Metamodel* of all graphical DSLs within the CINCO Cloud, the same generator can be used across the entire environment. The generator is technically based Pyro [ZS21a] and implements the concepts of CINCO described in [Lyb19].

(3) DSL Module Creation

Depending on the given specification, a customized *Language Server* and *DSL Editor* is created. The *Language Server* includes (as described in Sec. 5.2) a database schema to persist the instances and a multi-user capable interface that processes and distributes operations-based user interactions. All previously implemented components are woven into the *Language Server* so that users can access the additionally implemented features.

In order to build and initialize the DSL module in an automated way, the generator creates a *Specification* file (see Listing A.1 and A.2), which includes the runtime environment all dependencies and compilation steps. Thanks to the unification in a container image, different DSL modules can subsequently be combined to create a workspace pod.

(4) DSL Module Building

The generated DSL module is compressed after generation and submitted as an artifact to the *DSL Manager*. The *Builder Message Queue* (Builder MQ) uses the highly scalable and highly available RabbitMQ [23] framework to build received artifacts depending on the given specification file.

(5) Registry Extension

Then, the build container images are stored in the DSL manager *Registry*. From this point on, the *Orchestrator* service and its *Environment Manager* can access the new DSL, allowing a user to create a corresponding workspace.

(6) Pod Instantiation

As described in Sec. 5.3, users can start a new workspace by the *Environment Manager* interface. Thanks to the *Registry* API of the DSL Manager service, users can combine all previously created and released DSLs by themselves. Once a new workspace is created, the *Environment Manager* utilizes the Kubernetes controllers to start a workspace pod and instantiates the selected DSL modules including the Eclipse Theia editor.

(7) Workspace Execution

After the pod is ready and accessible, the new workspace and all contained DSL modules can be used. In this way, any new DSL can be made available to all users of the CINCO Cloud without the need for manual building, deployment and setup.

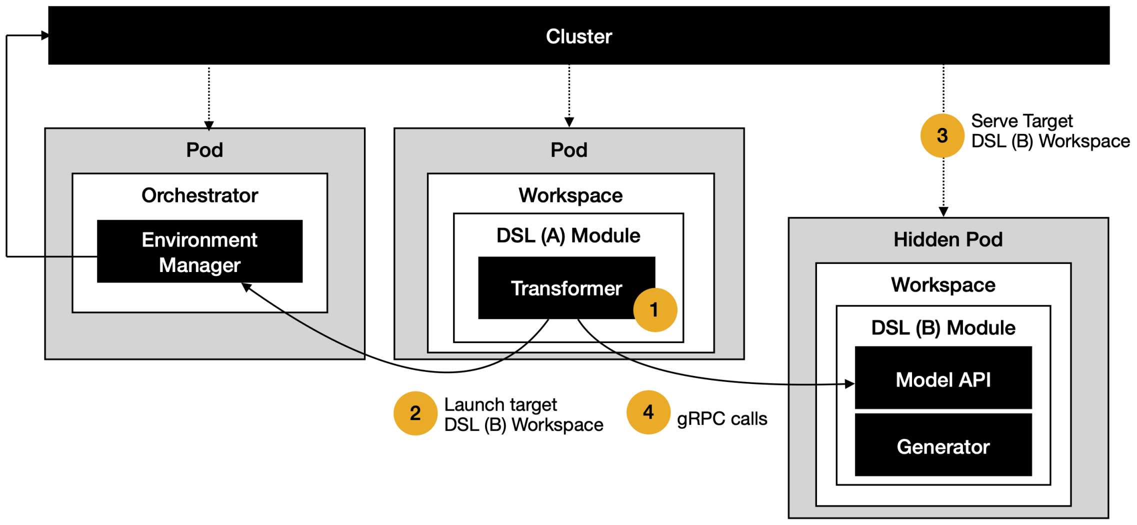


Figure 5.8: Aligned process to transform a model from DSL (A) to DSL (B).

5.5 Transformation

In the context of language-driven engineering, the execution of models is done by the stepwise transformation from a specific to a more abstract DSL up to a generation of executable source code. As described in the alignment concept (see Sec. 3.3), each DSL exists within the environment as a service so that transformation can be performed via successive API calls. Accordingly, each DSL module within the CINCO Cloud contains a *Model API* that can be accessed to create DSL instances in a service-oriented fashion. Thanks to the uniform architecture of all DSL modules and the cluster-based infrastructure, required DSLs can be dynamically integrated and instrumented.

In the following, we describe how the transformation of an instance of the DSL (A) to one of the DSL (B) is performed within the CINCO Cloud. Fig. 5.8 illustrates this process and shows the components and communication steps involved.

① Start Transformation of DSL (A)

As described in Sec. 3.1, a *Transformer* can be added to a DSL module, which can then be started by the user via the editor UI. The *Transformer* implements the corresponding API to specify which target DSL is used and how the transformation is performed. Based on this specification and implementation, the transformation is executed automatically by the CINCO Cloud.

② Launch Target DSL (B)

In order to use the target DSL (B), the corresponding DSL (B) module is instantiated first. To do this, the transformer of the DSL (A) instruments the *Environment Manager*, which reads the *DSL* (B) *Module* from the registry and adds it to the *Cluster*.

③ DSL (B) Instantiation

As already described in Sec. 5.4, the *Environment Manager* creates a hidden workspace pod with the help of the Kubernetes controllers and installs the *DSL* (B) *Module*. In contrast to the DSL release scenario of Sec. 5.4, the pod is only used temporarily for the transformation, so that it does not have to be accessible outside the CINCO Cloud and therefore does not have to be entered in the proxy.

④ Transform Model

Once the pod is ready, the *DSL* (B) *Module* can be reached by the transformer of the DSL (A). For this purpose, the *Model API* is instrumented via API utilizing the standardized gRPC [13] protocol. The transformation is then performed analogously to a user's interactions with an editor by sending the CRDTs described in Sec. 4.1 to the *Model API* of DSL (B). Once the model has been transformed from DSL (A) to DSL (B), further transformations or code generation can be triggered, depending on how the *DSL module* (B) is developed. Thanks to the modular structures of the individual DSL modules, any transformation chains can be performed in this uniform way.

5.6 Generation and Execution

In case a model of a DSL is not executed by an interpreter or further transformation, the CINCO Cloud offers the possibility of generating source code and executing it. As described in Sec. 5.2, generators are an integral part of the DSL modules and can be implemented via a specialized API. The corresponding editor offers a user the option of starting the generator by a simple button press.

The challenge lies in the subsequent execution of the generated code, as a variety of execution environments has to be available. Thanks to container-based infrastructure and publicly available repositories such as the Docker Hub [5], a huge amount of different runtime environment can be used to run the generated code. The runtime environment is defined by an infrastructure-as-code [Hüt12] Dockerfile which is supported by the CINCO Cloud environment. In this way, the generated code can be executed and used by a user without leaving the environment.

In the following, Fig. 5.9 illustrates the process and the components involved in the CINCO Cloud when generating and executing code from DSL models.

① Artifact Generation

After initiation by the user or by a higher-level transformation process, the *Generator* of the *DSL Module* is activated. Depending on the implementation, the model is traversed and textual source code is generated, which together with the specification of the runtime

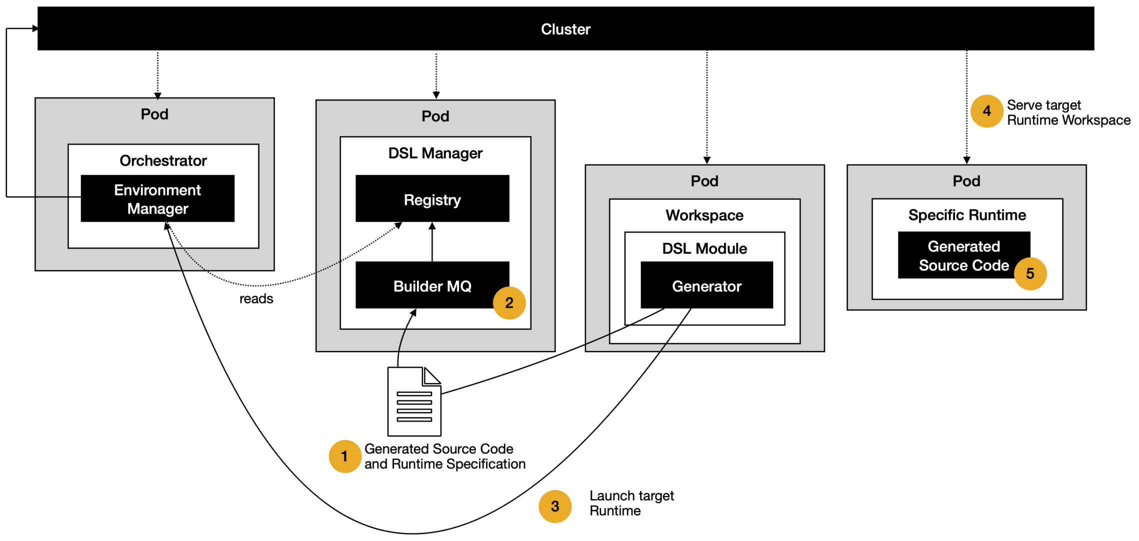


Figure 5.9: Code generation and execution process.

environment forms an artifact. This artifact is transferred to the *Builder MQ* via the *DSL Manager's* API.

(2) Artifact Building

The *Builder MQ* receives the artifact and builds a container image based on the given runtime specification. All dependencies are loaded and the compilation process is executed so that deployment process is prepared. Once the build process is complete, the image is stored in the *DSL Manager's Registry* and can be used inside the environment.

(3) Launch Target Runtime

After the generated artifact could be built successfully, the *Generator* instruments the execution API (see Sec. 5.2) of the *Environment Manager* to start the corresponding runtime environment.

(4) Pod Instantiation

Within the cluster of the CINCO Cloud, a new pod is created by instrumenting the Kubernetes controllers that allocate the required resources. Once the pod is ready, it is published via Kubernetes proxy to provide access to the user.

(5) Runtime Execution

To support a variety of possible runtime environments, the CINCO Cloud provides generic access to the generated application. Users first get access to the logs of the running containers to read the streaming output. In addition, the specification of the generated artifact can include port disclosure so that, for example, web applications and web services can be executed and accessed directly in the CINCO Cloud via browser.

Chapter 6

Conclusions and Future Work

This dissertation showed the realization of the language-driven engineering (LDE) approach through an aligned and collaborative environment. The described CINCO Cloud combines the previously published concepts [ZBS22, ZNS19, ZTS$^+$21, ZS21b] and extends them in the context of LDE.

First, the two basic concepts of alignment and collaboration were explained which clarify the technical challenges of LDE. Alignment refers to a simplified creation, refinement and specialization of DSLs in order to support each user in an optimal and targeted way. In turn, a step-by-step transformation and final code generation should lead to the execution of models by concretizing them.

The collaboration concept illustrated the technical implementation of simultaneous editing of a model by different users. In this way an intuitive collaboration for none technical users without detours over complicated and asynchronous versioning systems is established. At the same time, the collaboration concept includes the definition and management of user-specific views, so that a customized environment can be offered according to the capabilities and focus of a user.

Subsequently, the technical realization of the CINCO Cloud was described by illustrating the system architecture and specific scenarios. The key to the aligned environment is a dynamically expendable and scalable cluster-based infrastructure. By integrating modular language servers, new DSLs can be continuously created, refined and combined to meet the user's needs and expertise.

The goal of instrumenting DSLs for transformation and execution is met by the overarching service-oriented architecture, which was explained using various scenarios. In this way, users and language developers can work closely together and benefit directly from each other without the disadvantages of platform dependency and manual setups.

Since the CINCO Cloud is still under development, not many DSLs are realized yet. As soon as the environment is ready, nothing stands in the way of using DSL tools such as DIME [BFK$^+$16] and Pyrus [ZS21b] inside the environment.

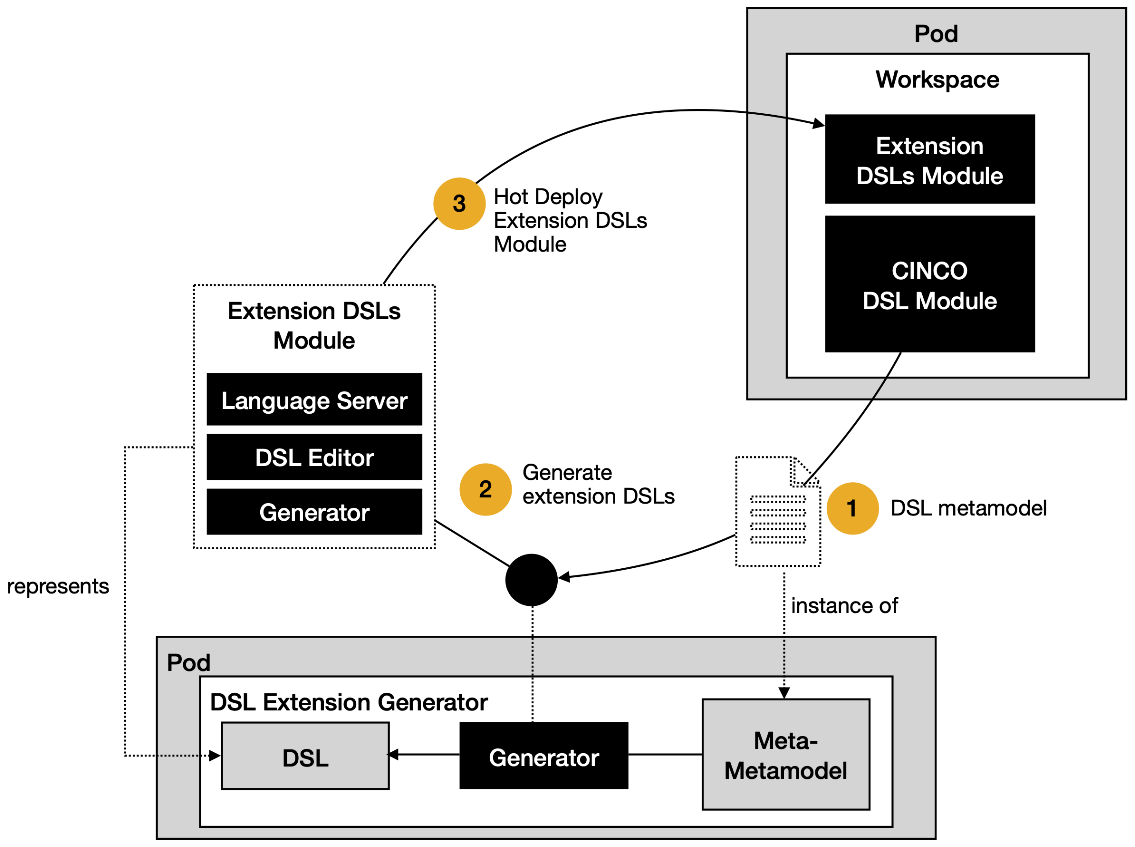


Figure 6.1: Extension DSL generation process.

Due to the novel concept of the CINCO Cloud and the realization as a dynamic and modular service-oriented distributed system, there are many possibilities for extension, two of which are described below.

6.1 No-Code LDE Environment

So far, the development of new and refined DSLs requires the use of general purpose programming languages to implement the extension components such as generators, transformers, validators and actions. To enable non-programmers to create DSLs in the CINCO Cloud, intuitive graphical languages should also be available to describe the extension components. As already described in [Kop19], DSLs for describing extensions should be themselves based on the original DSL and reuse the language constructs declared there, such as nodes and edges. In this way, users get an intuitive access to the extension languages by using the specified elements.

Technically, the extension languages can be generated accordingly on the basis of the abstract and concrete syntax and must then be embedded in the same workspace. For this purpose, the previous DSL generation process (see Sec. 5.4) must be extended as illustrated in Fig. 6.1.

(1) DSL Metamodel

Once the user has completed the abstract and concrete syntax of the DSL, the metamodel is transferred to the *DSL Extension Generator*.

(2) Extension DSLs Generation

The *Extension DSL Generator* reads the metamodel of a DSL to generate the specific languages for describing generators, transformers, validators and actions. The *Extension DSLs Module* is generated according to a uniform pattern and provides the previously defined language elements. In this way, previously defined semantics can be used to describe, for example, validators in combination with the concrete syntax of a DSL.

(3) Hot Deploy Extension DSLs Module

In order to give the user the possibility to describe the extension for the DSL, the automatically generated *Extension DSLs* are embedded in the existing workspace. Technically, the hot deployment requires the instantiation of the container image of the *Extension DSLs Module* within the running pod.

After the hot deployment, the specialized extension DSLs can be used to describe the additional language features. This extension of the CINCO Cloud enables none-technical users to develop DSLs by themselves.

6.2 DSL as a Service

As the approaches [PLDD15, HK09] already showed, DSLs can be used not only to generate executable source code but also to control the behavior of an application at runtime. In this way, for example, the reaction to an event inside an application can be directly influenced by the process model of a DSL. Established tools such as NodeRed [22] illustrate that users are supported by the intuitive and direct reaction of a system to changes in the model during development. However, extending existing systems with an interpretative no-code environment is complicated by the need to develop the appropriate DSL and to make the system respond to model changes.

Thanks to the service-oriented architecture of the CINCO Cloud, it is possible to integrate DSLs as an independent service in an external system (see Fig. 6.2). For this purpose, the DSL can first be developed and published in the CINCO Cloud as already described. Since each DSL module is itself a client server application, the editor of the DSL can be embedded directly within an external web application. This enables users of the external system to ability to use the DSL and allows them to create and edit models.

In order to affect the external system, any change to a model must be synchronized with the running application to influence the behavior. As described in Sec. 3.1, each DSL module within the CINCO Cloud has an event API that is implemented by corresponding

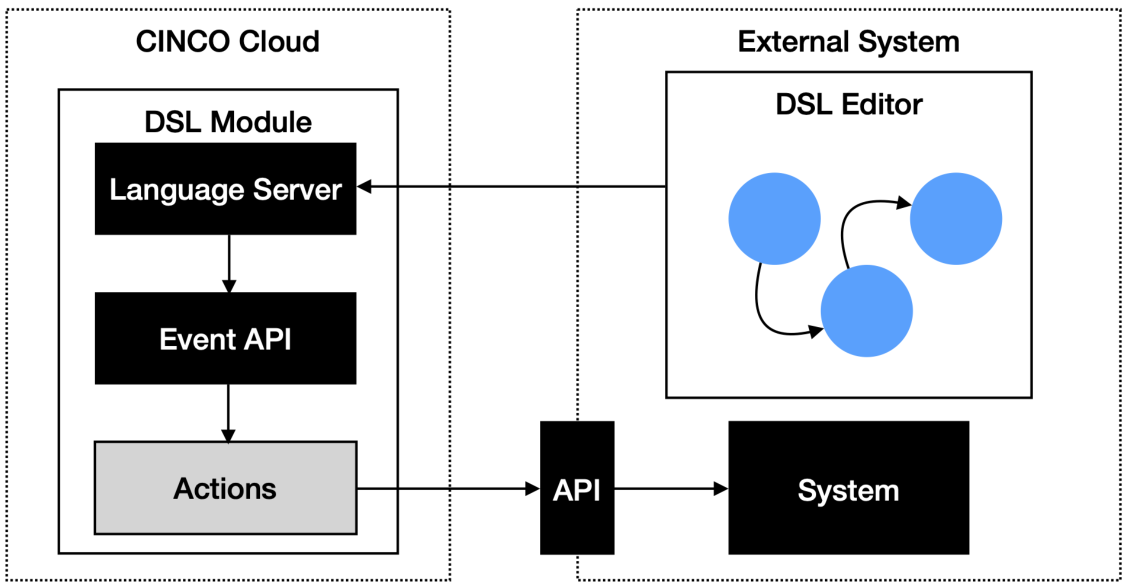


Figure 6.2: Embedding a DSL as a service inside an external system.

actions. In this way, any change to a model can be captured and transmitted to the API of the external system to influence its behavior by so-called web hooks.

This concept can be used not only for external systems but also within the CINCO Cloud, for example by embedding a DSL in an application created with DIME.

Bibliography

[AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[AMU$^+$21] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. An empirical study of developer discussions on low-code software development challenges. *arXiv preprint arXiv:2103.11429*, 2021.

[And15] Charles Anderson. Docker [software engineering]. *Ieee Software*, 32(3):102–c3, 2015.

[BAS14] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 126–140. Springer, 2014.

[BBC$^+$19] Jim-Marvin Bergmann, Tim Berthold, Robin Czarnetzki, Annika Fuhge, Nicole Funk, Christoph Meyer, Benedict Schubert, Fabian Storek, and Jonathan Thöne. *Modellbasierte Entwicklung von Alexa-Skills.* Universitätsbibliothek Dortmund, 2019.

[BED96] Judith S Bowman, Sandra L Emerson, and Marcy Darnovsky. *The practical SQL handbook: using structured query language.* Addison-Wesley Reading, Mass., 1996.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[BFK$^+$16] 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 *International Symposium on Leveraging Applications of Formal Methods*, pages 809–832. Springer, 2016.

[BNS18] Steve Boßelmann, Stefan Naujokat, and Bernhard Steffen. On the difficulty of drawing the line. In *International Symposium on Leveraging Applications of Formal Methods*, pages 340–356. Springer, 2018.

[Bru12] Hugo Bruneliere. Community-driven dsl development with collaboro. In *EclipseCon Europe 2012-Modeling Symposium*, 2012.

[Brz64] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[Bün19] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140, 2019.

[BWLM17] Steve Boßelmann, Alexander Wickert, Anna-Lena Lamprecht, and Tiziana Margaria. Modeling directly executable processes for healthcare professionals with xmdd. In *Service Business Model Innovation in Healthcare and Hospital Management*, pages 213–232. Springer, 2017.

[Cam14] Fabien Campagne. *The MPS language workbench: volume I*, volume 1. Fabien Campagne, 2014.

[Cha06] K Mani Chandy. Event-driven applications: Costs, benefits and design approaches. *Gartner Application Integration and Web Services Summit*, 2006, 2006.

[CHB$^+$19] Chen Chen, Paulina Haduong, Karen Brennan, Gerhard Sonnert, and Philip Sadler. The effects of first programming language on college students' computing attitude and achievement: a comparison of graphical and textual languages. *Computer Science Education*, 29(1):23–48, 2019.

[DKRS21] Ajith Devadiga, Sudhanshu Kumar, Mittal Rachhadiya, and Aarti Sahitya. Integrated development environment on cloud. *Available at SSRN 3867647*, 2021.

[DRFMM17] Davide Di Ruscio, Mirco Franzago, Ivano Malavolta, and Henry Muccini. Envisioning the future of collaborative model-driven software engineering. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 219–221. IEEE, 2017.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[EVDSV$^+$13] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[EVDSV$^+$15] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[FFF$^+$15] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[FG12] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language.* Addison-Wesley, 2012.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* University of California, Irvine, 2000.

[Fow10] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[Fuh18] Annika Fuhge. *Graphische Modellierung von Cinco Produktspezifikationen.* PhD thesis, BSc thesis, TU Dortmund, 2018.

[GCN20] Bikash Gogoi, Unnikrishnan Cheramangalath, and Rupesh Nasre. Custom code generation for a graph dsl. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 51–60, 2020.

[Gho11] Debasish Ghosh. Dsl for the uninitiated. *Communications of the ACM*, 54(7):44–50, 2011.

[GR92] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques.* Elsevier, 1992.

[GvKT$^+$20] Dominik Gloger, Yannick von Kienle, Karen Toben, Sebastian Schröder, Marvin Krause, Lukas Hüwe, Yannik Weber, Fabian Dillkötter, Christian Rathmann, Michelle Zechner, Fabian Eckey, and Jonas Stilling. *Responsive and Knowledge-Driven Business Modeling.* Universitätsbibliothek Dortmund, 2020.

[HK09] Bogumiła Hnatkowska and Krzysztof Kasprzyk. Integration of application business logic and business rules with dsl and aop. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 30–39. Springer, 2009.

[HSCJ09] Michael S Horn, Erin Treacy Solovey, R Jordan Crouser, and Robert JK Jacob. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 975–984, 2009.

[Hüt12] Michael Hüttermann. Infrastructure as code. In *DevOps for Developers*, pages 135–156. Springer, 2012.

[KLNS21] Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, and Bernhard Steffen. Towards language-to-language transformation. *International Journal on Software Tools for Technology Transfer*, pages 1–23, 2021.

[Kop19] Dawid Kopetzki. *Generation of Domain-Specific Language-to-Language Transformation Languages.* PhD thesis, TU Dortmund University, 2019.

[KRKP$^+$16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, volume 2016. 2016.

[LAB$^+$06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and computation: Practice and experience*, 18(10):1039–1065, 2006.

[LJJ07] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. Dsl classification. In *OOPSLA 7th Workshop on Domain specific modeling*, 2007.

[LKS18] Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. Design for 'x'through model transformation. In *International Symposium on Leveraging Applications of Formal Methods*, pages 381–398. Springer, 2018.

[LKZ$^+$18] Michael Lybecait, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat, and Bernhard Steffen. A tutorial introduction to graphical modeling and metamodeling with cinco. In *International Symposium on Leveraging Applications of Formal Methods*, pages 519–538. Springer, 2018.

[LMN15] Anna-Lena Lamprecht, Tiziana Margaria, and Johannes Neubauer. On the use of xmdd in software development education. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 835–844. IEEE, 2015.

[LR01] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings fifth ieee international enterprise distributed object computing conference*, pages 118–127. IEEE, 2001.

[Luk18] Marko Luksa. *Kubernetes in action*. Manning Publications Shelter Island, 2018.

[Lyb19] Michael Lybecait. *Meta-Model Based Generation of Domain-Specific Modeling Tools*. PhD thesis, TU Dortmund University, 2019.

[MdSMAM21] André Menolli, Luan de Souza Melo, Maurıcio Massaru Arimoto, and Andreia Malucelli. An empirical study on the impact of aspect-oriented model-driven code generation. *Sitepress*, 2021.

[MLA10] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[MMBL93] Thomas G Moher, D Mak, B Blumenthal, and L Levanthal. Comparing the comprehensibility of textual and graphical programs. In *Empirical studies of programmers: fifth workshop*, pages 137–161. Ablex, Norwood, NJ, 1993.

[MNS05] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. jeti: A tool for remote tool integration. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 557–562. Springer, 2005.

[MS09] Tiziana Margaria and Bernhard Steffen. Business process modeling in the jabc: the one-thing approach. In *Handbook of research on business process modeling*, pages 1–26. IGI Global, 2009.

[MS10] Tiziana Margaria and Bernhard Steffen. Simplicity as a driver for agile innovation. *Computer*, 43(6):90–92, 2010.

[MS12] Tiziana Margaria and Bernhard Steffen. Service-orientation: conquering complexity with xmdd. In *Conquering Complexity*, pages 217–236. Springer, 2012.

[MS20] Tiziana Margaria and Bernhard Steffen. extreme model-driven development (xmdd) technologies as a hands-on approach to software development without coding. *Encyclopedia of Education and Information Technologies*, pages 732–750, 2020.

[Nau17] Stefan Naujokat. *Heavy meta: model-driven domain-specific generation of generative domain-specific modeling tools*. PhD thesis, TU Dortmund University, 2017.

[NFSM14] Johannes Neubauer, Markus Frohme, Bernhard Steffen, and Tiziana Margaria. Prototype-driven development of web applications with dywa. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 56–72. Springer, 2014.

[NLKS18] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer*, 20(3):327–354, 2018.

[PLDD15] Aleksandar Popovic, Ivan Lukovic, Vladimir Dimitrieski, and Verislav Djukic. A dsl for modeling application-specific functionalities of business applications. *Computer Languages, Systems & Structures*, 43:69–95, 2015.

[PS14] Candy Pang and Duane Szafron. Single source of truth (ssot) for service oriented architecture (soa). In *International Conference on Service-Oriented Computing*, pages 575–589. Springer, 2014.

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380, 2018.

[Ros07] Evan Rosen. The culture of collaboration. *San Francisco: Red Ape*, 2007.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[Sch06] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[Sco93] Roger S Scowen. Generic base standards. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 25–34. IEEE, 1993.

[SGNM19] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. Language-driven engineering: from general-purpose to purpose-specific languages. In *Computing and Software Science*, pages 311–344. Springer, 2019.

[SH13] Michael Stevenson and John G Hedberg. Learning and design with online real-time collaboration. *Educational Media International*, 50(2):120–134, 2013.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit—a flexible graphical environment for methodology modelling. In *International Conference on Advanced Information Systems Engineering*, pages 168–193. Springer, 1991.

[SM20] Salim Saay and Tiziana Margaria. Xmdd as key enabling technology for integration of large scale elearning based on nrens. In *2020 IEEE 20th International Conference on Advanced Learning Technologies (ICALT)*, pages 45–46. IEEE, 2020.

[SMN$^+$06] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jabc. In *Haifa verification conference*, pages 92–108. Springer, 2006.

[SN16] Bernhard Steffen and Stefan Naujokat. *Archimedean Points: The Essence for Mastering Change*, pages 22–46. Springer International Publishing, Cham, 2016.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[SSS21] Ender Sahinaslan, Onder Sahinaslan, and Mehmet Sabancıoglu. Low-code application platform in meeting increasing software demands quickly: Setxrm. In *AIP Conference Proceedings*, volume 2334, page 070007. AIP Publishing LLC, 2021.

[Sub21] Venkat Subramaniam. *Programming DSLs in Kotlin*. Pragmatic Bookshelf, 2021.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 107–119. IEEE, 1999.

[TTP$^+$95] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[WHF$^+$13] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

[WW21] Shouhang Wang and Hai Wang. A teaching module of no-code business app development. *Journal of Information Systems Education*, 32(1):1–8, 2021.

[ZBS22] Philip Zweihoff, Steve Bosselmann, and Bernhard Steffen. Towards tailored and precise conflict detection in real-time collaborative modeling environments. 2022.

[ZNS19] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen. Pyro: Generating domain-specific collaborative online modeling environments. In *International Conference on Fundamental Approaches to Software Engineering*, pages 101–115. Springer, 2019.

[ZS21a] Philip Zweihoff and Bernhard Steffen. A generative approach for user-centered, collaborative, domain-specific modeling environments. *arXiv preprint arXiv:2104.09948*, 2021.

[ZS21b] Philip Zweihoff and Bernhard Steffen. Pyrus: an online modeling environment for no-code data-analytics service composition. In *International Symposium on Leveraging Applications of Formal Methods*, pages 18–40. Springer, 2021.

[ZSD12] Wenyi Zhou, Elizabeth Simpson, and Denise Pinette Domizi. Google docs in an out-of-class collaborative writing activity. *International Journal of Teaching and Learning in Higher Education*, 24(3):359–375, 2012.

[ZTS$^+$21] Philip Zweihoff, Tim Tegeler, Jonas Schürmann, Alexander Bainczyk, and Bernhard Steffen. Aligned, purpose-driven cooperation: the future way of system development. In *International Symposium on Leveraging Applications of Formal Methods*, pages 426–449. Springer, 2021.

Web Resources

[1] Angular. https://angular.io/. [Online; last accessed 13-April-2021].

[2] BitBucket. https://bitbucket.org/. [Online; last accessed 13-April-2021].

[3] Codespaces. https://github.com/features/codespaces. [Online; last accessed 13-April-2021].

[4] Docker. https://www.docker.com/. [Online; last accessed 13-April-2021].

[5] Docker Hub. https://hub.docker.com/. [Online; last accessed 13-April-2021].

[6] Eclipse. https://www.eclipse.org/. [Online; last accessed 13-April-2021].

[7] Eclipse Che. https://www.eclipse.org/che/. [Online; last accessed 13-April-2021].

[8] Eclipse Glsp. https://www.eclipse.org/glsp/. [Online; last accessed 13-April-2021].

[9] Eclipse Theia. https://theia-ide.org/. [Online; last accessed 13-April-2021].

[10] GitHub. https://github.com/. [Online; last accessed 13-April-2021].

[11] GitLab. https://gitlab.com/. [Online; last accessed 13-April-2021].

[12] Gitpod. https://gitpod.io//. [Online; last accessed 13-April-2021].

[13] grpc. https://grpc.io/. [Online; last accessed 13-April-2021].

[14] Joint Js. https://www.jointjs.com/opensource. [Online; last accessed 13-April-2021].

[15] Kube Ctl. https://kubernetes.io/docs/reference/kubectl/overview/. [Online; last accessed 13-April-2021].

[16] Kube Proxy. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. [Online; last accessed 13-April-2021].

[17] Kubernetes. `https://kubernetes.io/`. [Online; last accessed 13-April-2021].

[18] Kubernetes Scheduler. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`. [Online; last accessed 13-April-2021].

[19] Language Server Implementations. `https://microsoft.github.io/language-server-protocol/implementors/servers/`. [Online; last accessed 13-April-2021].

[20] Language Server Protocol. `https://microsoft.github.io/language-server-protocol/specifications/specification-current/`. [Online; last accessed 13-April-2021].

[21] Monaco Editor. `https://microsoft.github.io/monaco-editor/`. [Online; last accessed 13-April-2021].

[22] NodeRed. `https://nodered.org/`. [Online; last accessed 13-April-2021].

[23] RabbitMq. `https://www.rabbitmq.com/`. [Online; last accessed 13-April-2021].

Appendix A

Large Figures and Listings

```
# build the language server
# --------------------------------
FROM docker.io/library/maven:3-adoptopenjdk-11 as backend-builder
WORKDIR /backend
COPY ./backend /backend
# outputs language server to releng/de.jabc.cinco.meta.core.parent/de.jabc.cinco.meta
    .core.ide/target/language-server
RUN cd releng/de.jabc.cinco.meta.core.parent && \
    mvn clean install -DskipTests
RUN mv releng/de.jabc.cinco.meta.core.parent/de.jabc.cinco.meta.core.ide/target/
    language-server /backend/language-server

# build the extension
# --------------------------------
FROM docker.io/library/node:10-buster-slim as extension-builder
WORKDIR /extension
COPY ./cinco-extension /extension
# outputs extension to /extension/cinco-extension-0.0.1.vsix
RUN yarn

# build the theia editor
# --------------------------------
FROM docker.io/library/openjdk:11.0.11-slim-buster
ENV NPM_CONFIG_PREFIX=/home/node/.npm-global
# make readable for root only
RUN chmod -R 750 /var/run/
WORKDIR /editor
RUN useradd -ms /bin/bash theia
COPY --chown=theia:theia web /editor
# install node and yarn
RUN apt update && \
    apt install -y gnupg gnupg2 curl && \
```

Listing A.1: DSL Module Image specification Dockerfile

```
    apt install -y gnupg gnupg2 curl && \
    curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add - && \
    echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list && \
    apt update && \
    apt install -y nodejs npm yarn && \
    apt remove -y gnupg gnupg2 curl && \
    apt autoremove -y
# copy language-server to singleton cinco-language-server-extension
COPY --from=backend-builder /backend/language-server/ /editor/cinco-language-server-extension/language-server/
# install theia
RUN yarn
# copy extension after the build
# otherwise the build will fail because the /plugins directory already exists
COPY --from=extension-builder /extension/cinco-extension-0.0.1.vsix /editor/browser-app/plugins
# integrate favicon
RUN cp /editor/favicon.ico /editor/browser-app/lib && \
  sed -i 's/<\/head>/<link rel="icon" href="favicon.ico" \/><\/head>/g' /editor/browser-app/lib/index.html
RUN mkdir /editor/workspace
VOLUME /editor/workspace
EXPOSE 3000
CMD cd /editor/browser-app && \
    yarn run theia start --port=3000 --root-dir=/editor/workspace --plugins=local-dir:./plugins --hostname 0.0.0.0
```

Listing A.2: DSL Module Image specification Dockerfile cont.

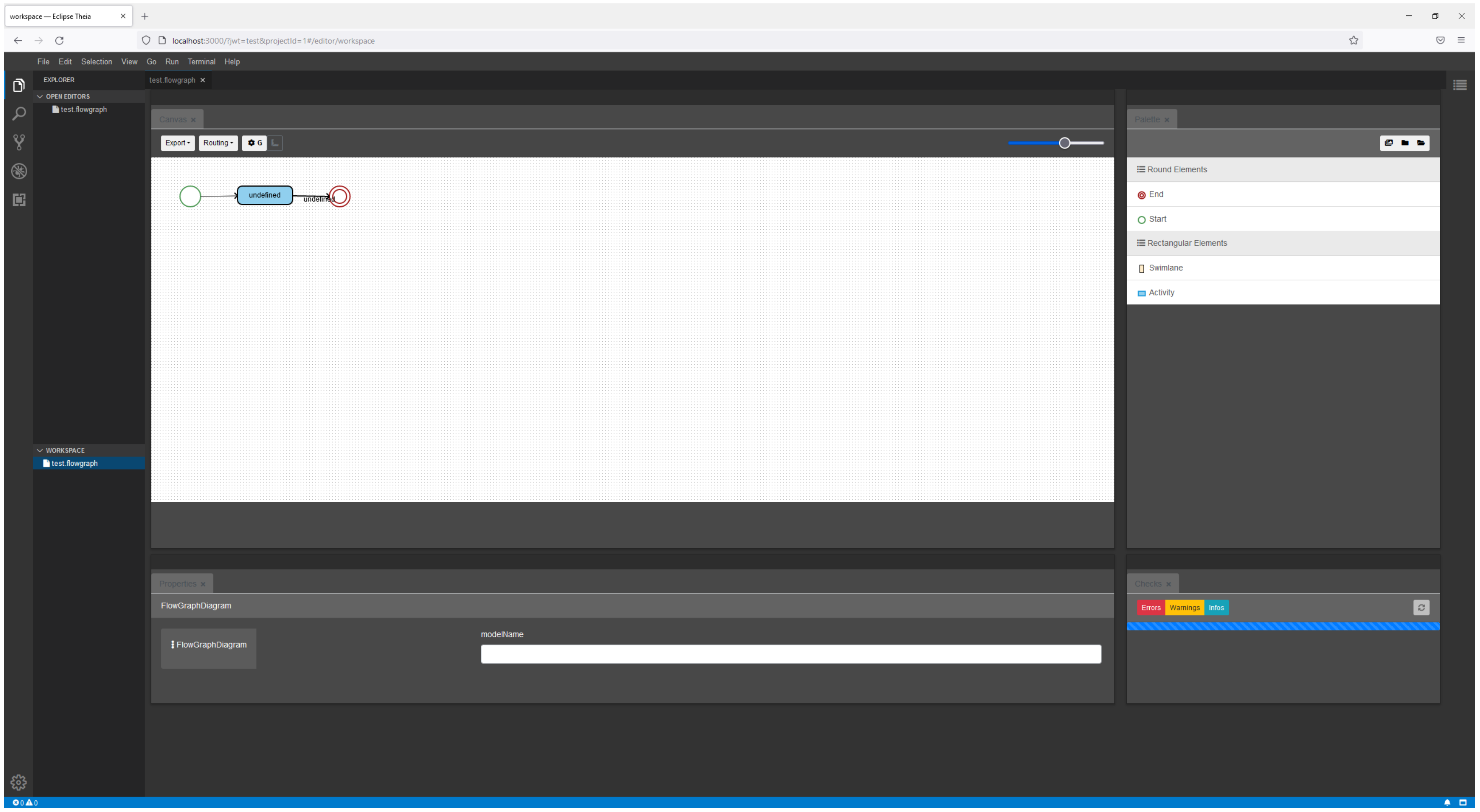
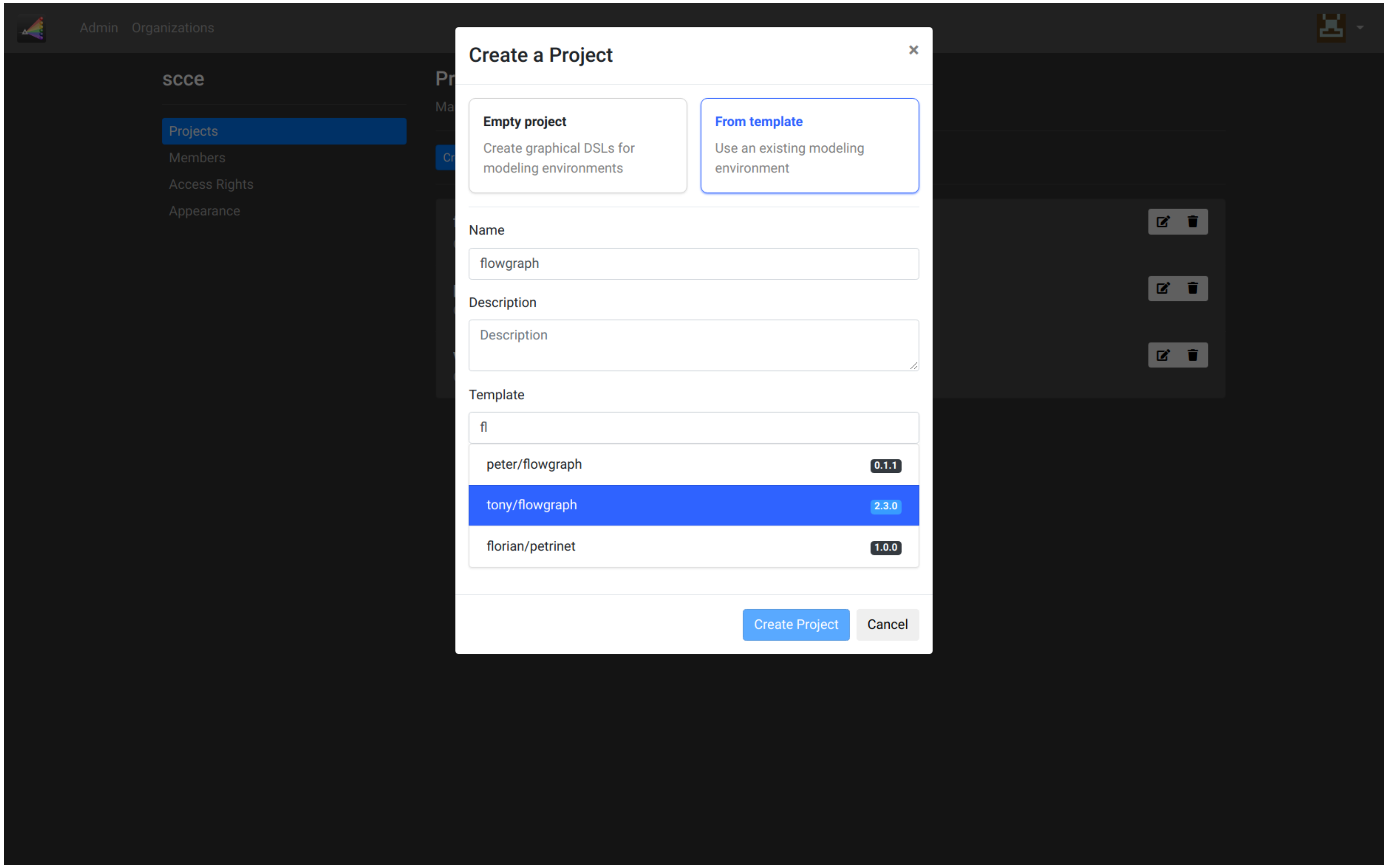


Figure A.1: Screenshot of the CINCO Cloud editor interface inside a browser.



Create a Project

Empty project
Create graphical DSLs for modeling environments

From template
Use an existing modeling environment

Name: flowgraph

Description: Description

Template: fl

peter/flowgraph	0.1.1
tony/flowgraph	2.3.0
florian/petrinet	1.0.0

Figure A.2: Screenshot of the CINCO Cloud Environment Manager interface inside a browser.

Appendix B

Attached Papers

Parts of this dissertation have been published in cooperation with a number of co-authors. This section lists the publications with a classification of my contribution for each.

I Bosselmann, Steve, 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), volume 9953 of LNCS, pages 809–832. Springer, 2016. DOI: 10.1007/978-3-319-47169-3 60 URL: `https://doi.org/10.1007/978-3-319-47169-360`

The presented concepts and technologies were discussed among all authors. Implementations of Dime have been done primarily by Steve Boßelmann, Markus Frohme, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner and myself. Enhancements in Cinco required for the implementation of Dime have been done primarily by Dawid Kopetzki and Michael Lybecait.

II Philip Zweihoff, Steve Bosselmann and Bernhard Steffen. **Towards Semantics-Driven Conflict Detection in Real-Time Collaborative Modeling Environments**. In: Innovations in Systems and Software Engineering (ISSE 2022), under submission. Springer, 2022.

The presented concepts were discussed among all authors. I co-authored all sections and was main author of all sections except 4c.

III Lybecait, Michael, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat and Bernhard Steffen. **A Tutorial Introduction to Graphical Modeling and Meta-modeling with CINCO**. In: Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), volume 11244 of LNCS, pages 381–398.

Springer, 2018. DOI: 10.1007/978-3-030-03418-4_31 URL: `https://doi.org/10.1007/978-3-030-03418-4_31`

The presented concepts and technologies were discussed among all authors. I am the main author of section 3. Implementation of the GCS tool was primarily done by Annika Fuhge and implementation of Pyro was done by myself.

IV PHILIP ZWEIHOFF AND BERNHARD STEFFEN. **A Generative Approach for User-Centered, Collaborative, Domain-Specific Modeling Environments**. In: dblp computer science bibliography, eprint 2104.09948 of CoRR. arXiv, 2021. URL: `https://arxiv.org/abs/2104.09948`

The presented concepts were discussed among all authors. I was main author of all sections.

V PHILIP ZWEIHOFF AND BERNHARD STEFFEN. **Pyrus: an Online Modeling Environment for No-Code Data-Analytics Service Composition**. In: Proc. of the 10th Int. Symp. on Leveraging Applications of Formal Methods (ISoLA 2021), volume 13036 of LNCS, pages 18-40. Springer, 2021. DOI: 10.1007/978-3-030-89159-6_2 URL: `https://doi.org/10.1007/978-3-030-89159-6_2`

The presented concepts were discussed among all authors. I was main author of all sections.

VI PHILIP ZWEIHOFF, STEFAN NAUJOKAT AND BERNHARD STEFFEN. **Pyro: Generating domain-specific collaborative online modeling environments**. In: International Conference on Fundamental Approaches to Software Engineering (FASE 2019), volume 11424 LNCS, pages 101–115. Springer, 2019. DOI: 10.1007/978-3-030-16722-6_6 URL: `https://doi.org/10.1007/978-3-030-16722-6_6`

The presented concepts were discussed among all authors. I co-authored all sections and was main author of sections 1, 3, 4, and 5.

VII PHILIP ZWEIHOFF, TIM TEGELER, JONAS SCHÜRMANN, ALEXANDER BAINCZYK AND BERNHARD STEFFEN. **Aligned, Purpose-Driven Cooperation: The Future Way of System Development**. In: Proc. of the 10th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021), volume 13036 of LNCS, pages 426-449. Springer, 2021. DOI: 10.1007/978-3-030-89159-6_27 URL: `https://doi.org/10.1007/978-3-030-89159-6_27`

The presented concepts were discussed among both authors. I was main author of sections 3, 4 and 5.