Pyro: Generating Domain-Specific Collaborative Online Modeling Environments

Philip Zweihoff(✉), Stefan Naujokat, and Bernhard Steffen

Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
{philip.zweihoff,stefan.naujokat,bernhard.steffen}@tu-dortmund.de

Abstract. We present Pyro, a framework for enabling domain-specific modeling via the internet. Provided with an adequate metamodel specification, Pyro turns your browser into a collaborative, domain-specific, graphical development environment with features reminiscent of desktop IDEs for textual programming languages. The required metamodeling is supported in a high-level, simplicity-driven fashion, and the entire ready-to-run browser-based domain-specific development environment is generated fully automatically. We will illustrate the steps of this development along the realization of a graphical IDE for the Architecture Analysis and Design Language (AADL).

1 Introduction

Domain-specific languages (DSLs) aim at closing the gap between domain knowledge and software development by explicitly supporting the required domain concepts. Graphical domain-specific languages have turned out to be particularly suitable for domain experts without any programming background. The bottleneck in practice is the enormous effort to develop the required domain-specific graphical modeling tools. The Cinco *SCCE Meta Tooling Suite* [26] has been designed to overcome this bottleneck by providing a holistic, simplicity-driven [22] approach for the creation of such domain-specific graphical modeling tools. A key feature of Cinco is that it generates the entire graphical modeling environment (referred to as 'Cinco Products' in the remainder of the paper) from high-level specifications of the defined model structures and functionalities. The (translational) semantics of the specified modeling language is defined in terms of code generation, model transformation, evaluation, and/or interpretation [20]. Cinco Products are Eclipse-based, graphical modeling tools that are realized via a number of Eclipse plug-ins [13]. Thus, setting up a Cinco Product involves some technical aspects that are beyond the competence of typical domain experts, and it becomes even more tedious when one wants to enable a cooperative development.

In this paper, we present *Pyro*, a tool that enables one to generate Cinco Products for collaborative modeling that run in a web browser. Conceptually, Pyro borrows from modern online editors for collaborative work, like Google

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 101–115, 2019.
https://doi.org/10.1007/978-3-030-16722-6_6

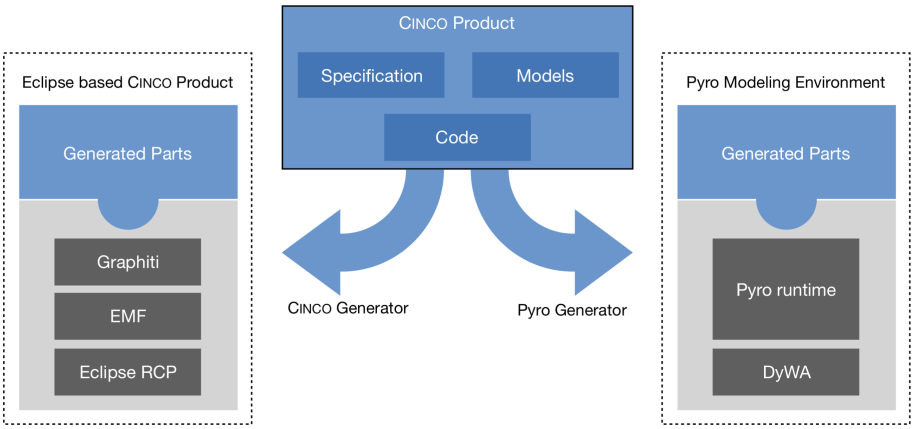

Fig. 1. Cinco generation architecture.

Docs, Microsoft Office 365, or solutions like ShareLaTeX/Overleaf that even free one from maintaining a corresponding build and runtime environment.

Key to the realization of Pyro is that CINCO follows a fully generative approach on the meta level, which allows one to modularly 'retarget' the CINCO Product Generation for the web (cf. Fig. 1). Technically, Pyro web modeling environments utilize *DyWA* [27] (Dynamic Web Application) for data modeling, empowering prototype-driven application development.

In order to achieve this retargeting and to enable collaborative work, Pyro needs to, in particular, compensate for all the required functionality provided by the Eclipse platform, like the EMF framework with GMF or Graphiti for graphical editors. Altogether, this poses the following three key challenges:

- Developing an adequate web solution for the metamodel-based model handling (API, persistence, event system, etc.) that in the Eclipse world is provided by the EMF framework [33] (see *Architecture Backend*, Sect. 3.1).
- Developing a frontend on top of these model structures that feels like a modern integrated development environment with a graphical editor for the models, which in the Eclipse world is provided by the Rich Client Platform (RCP) [24] and the Graphiti editor framework [2] (see *Architecture Frontend*, Sect. 3.2).
- Enabling real-time live collaborative working on models, which is not foreseen in an offline client like Eclipse (see *Collaborative Editing*, Sect. 4).

In the course of this tool paper, Pyro is illustrated along the development of a graphical modeling environment for the *Architecture Analysis and Design Language* (AADL), an SAE standard for modeling the architecture of embedded real-time systems [29]. CINCO was used to develop a graphical AADL modeling tool supporting a subset of AADL's features tailored to be used in teaching [28],

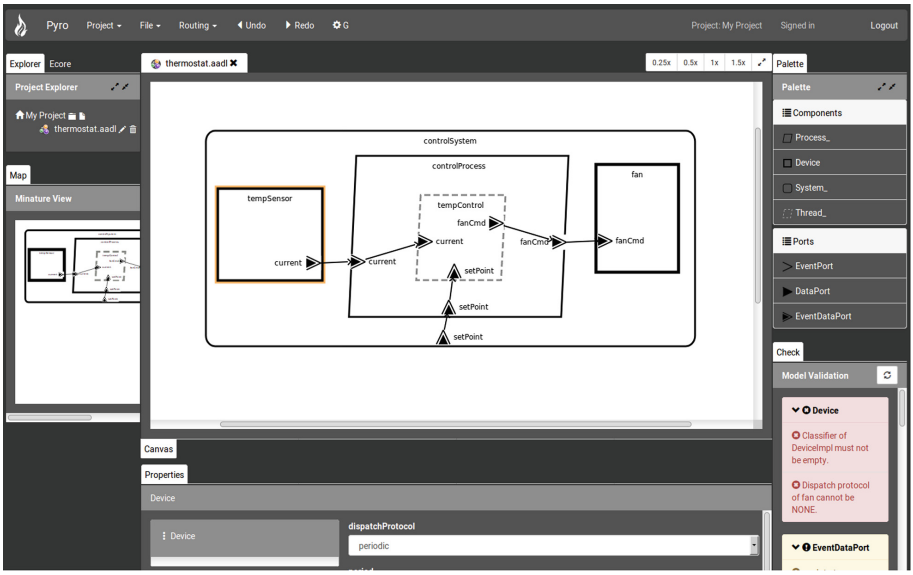

Fig. 2. Pyro web-based modeling environment for the AADL language.

where it replaces the graphical editor of the OSATE tool [8] (AADL's reference implementation). Furthermore, a dedicated code generator was developed to support verification with behavior specified with the BLESS language [17]. Another example for Pyro realizing a DSL for point and click adventures can be found in [21].

Figure 2 shows the web-based graphical AADL editor in Pyro[1]. We will use this editor in the remainder of this paper to illustrate CINCO's and Pyro's core ideas and concepts. The user interface is designed after commonly known concepts from integrated development environments, like Eclipse or IntelliJ. The main area in the center is covered by the *modeling canvas* showing the currently edited model. On the right, there is the *palette* showing the available types of modeling elements. They can be placed onto the canvas just by drag&drop. The attributes of the currently selected element in the editor can be set via the *properties* view at the bottom. The *validation* view (bottom right corner) constantly checks for the syntax and static semantics of the model in the canvas and provides appropriate error or warning messages. Finally, a *project explorer* and a *menu bar* complete the IDE-like appearance.

The remainder of the paper is organized as follows: While Sect. 2 briefly describes the use of CINCO's specification languages to define a sophisticated graphical

[1] The editor is available for experimentation on the Pyro website: https://pyro.scce.info.

modeling language, the generation to a web-based environment and the resulting architecture is explained in Sect. 3. The mechanisms and techniques used to enable simultaneous collaboration are explained in Sect. 4. The paper closes with a summary, related work, and an outlook of the future development in Sect. 5.

2 DSL Development with Cinco

Cinco is a language workbench [11] for the simplicity-driven development of graphical modeling environments that are domain-specific [12], support full code generation [10,15], and easily integrate existing solutions in the form of services [23]. As Cinco is itself a meta-level application of these principles [25], it is specialized to the domain of 'graph-based graphical modeling tools' and fully generates such tools from meta-level descriptions (models) – the key enabling factor for the whole Pyro approach. Primarily relevant in this regard are two Cinco metamodeling languages:[2]

1. The *Meta Graph Language* (MGL) allows for the definition of the abstract syntax of the developed language, i.e., which types of language elements exist and how they can be related. In the context of AADL, this means, for instance, that a *system* model consists of *devices*, *processes* and *threads*, and that all of them have *ports* (of different types) that can be connected with *data/information flow* edges.
2. The *Meta Style Language* (MSL) is used to specify the concrete graphical syntax of those MGL-defined concepts by means of simple hierarchical shapes and their appearance (such as color, line type/width, etc.). As can be seen in Fig. 2, for instance, *devices* are depicted by a black thick line rectangle, while *threads* appear as a grey dashed line parallelogram.

With these meta-level specification files, the Cinco Product Generator (which is part of Cinco) generates plug-ins for the Eclipse Rich Client Platform (RCP) that realize the editor based on the Eclipse Modeling Framework (EMF) and the Graphiti graphical editor framework. Further additions to the editor, which are not covered by these two specification files, can be injected in an aspect-oriented fashion [16]: Cinco provides a so-called mechanism of *hooks* that are triggered on the occurrence of certain events, for instance, when a node is created, moved, or deleted. Hooks are inserted into the MGL file with *annotations* on the model elements defined therein. The effect of a hook can either be modeled in a transformation language [20] or directly be written as Java code using the generated model API. In the context of the AADL editor, e.g., a `postMoveHook` is used to move a port to the nearest border within its container after it has been moved by the user. This results in a very natural 'snapping to the border' effect during modeling.

As Cinco follows a fully generative approach, the very same specification files are utilized by Pyro to generate a web-based modeling editor that runs in

[2] For a more elaborate introduction on how to define a graphical editor with Cinco, as well as other case studies and exemplary modeling languages, please refer to [26].

the browser (cf. Fig. 1). Of course, in this context, the running platform won't be based on Eclipse anymore, but based on common web frameworks like Angular for the frontend and Java EE for the backend. The aspects of a CINCO Product included in a service-oriented fashion via native components written in Java (for instance a code generator or editor-assisting features like the hooks discussed above) can thus directly be run also in the backend of the Pyro editor.

In the following, we will focus on two particularly important aspects of Pyro: After discussing the frontend/backend architecture of the generated Pyro modeling environments in Sect. 3, we will take a deeper look at the communication pattern between the involved components that facilitates synchronous collaborative modeling (cf. Sect. 4).

3 Architecture

In contrast to developing an Eclipse-based modeling environment, for the realization of a web-based solution one nearly has to start from scratch. Eclipse itself is built on a huge amount of plug-ins, developed over the past seventeen years. In particular, the Eclipse Modeling Project provides many frameworks for developing modeling languages based on metamodels and bundling them into a rich IDE. In the context of the web, development of integrated environments has just started, so that only a few best practices, plug-ins, and frameworks are available. This means, even fundamental features often have to be implemented to enable basic functionalities. The main difference between local desktop IDEs and a web-based environment like Pyro is the opportunity to provide distributed access to a centralized instance by multiple users at the same time. This results in new challenges and requirements regarding the synchronization between multiple users and conflict resolution for oppositional modifications.

Thus, the Pyro architecture must be built in a way that adequately substitutes what Eclipse already provides in the desktop application context, but also be prepared for the distributed setting with multiple users – in particular for supporting live collaborative editing on the same models. In this section, the generation of Pyro web-based modeling environments is described in a way that shows how the needed information is collected from CINCO's high-level specification metamodels and where the generated code is placed and distributed in the overall context to build the Pyro architecture.

The previously introduced specification of the AADL modeling language constitutes the source for the tool generation step. After the Pyro generator is triggered, all MGL and MSL files for a CINCO-based modeling tool are collected to gather the required information. At this point, all modeling languages, including their available node and edge types, are visible for the generator.

In the next step, a template of the modeling environment web application is created. The gray parts with dotted borders in Fig. 3 show the static elements independent of the given language specification, whereas the blue parts with solid borders are specifically generated from this specification. The template consists of a *DyWA*-based backend, extended by a specific *Domain Layer*

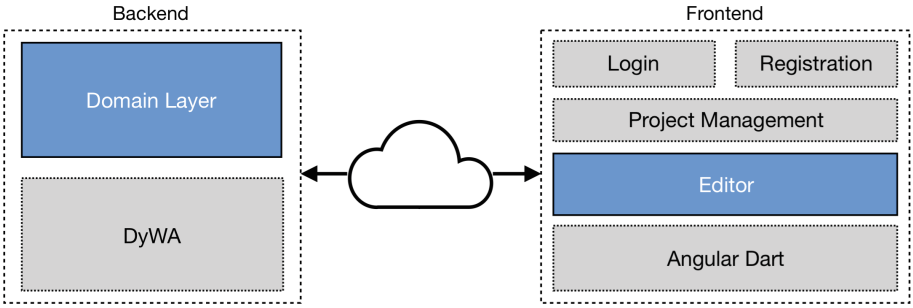

Fig. 3. Overall architecture of the generated web-based modeling environment.

for communication. On the client side, some general parts provide *Registration*, *Login*, and *Project Management*, but the main component is the specific *Editor* generated to handle instances of the graphical modeling language. The underlying single-page web application framework *Angular Dart* [1] is utilized to enable the required features of a rich internet application, like versatile user interaction and asynchronous communication.

Essentially, in the backend, the challenge of providing the metamodel-based model handling (persistence, API, event handling, etc.) is solved, which in the Cinco desktop client world is provided by the EMF framework. The frontend, on the other hand, realizes the rich IDE-like frame application with the graphical editor for the models. In the following, these two parts are explained in more detail to show how the different layers are connected and which parts are generated to establish the entire integrated environment.

3.1 Backend

The backend of a modeling environment generated using Pyro consists of two main layers: One is responsible for the centralized persistence of model instances, the other for receiving and distributing modifications. The lowest level of the web application is the database to store information in a centralized fashion. This layer handles the representation of predefined metamodels for the given domain-specific languages. Pyro modeling environments utilize the *DyWA* as an abstraction layer of a database to store types and objects in a dynamic and loosely coupled fashion [27]. Based on the specified languages' node and edge types, a *Domain Data Plug-in* (see Fig. 4) is generated by Pyro which declares types, associations, attributes, and inheritance. The main reason for using the *DyWA* as model layer is its *Domain Generator*, which generates a specific *DyWA API* providing entities and controllers for the previously given types to handle their instances on a simplified layer above the database. This closely resembles the APIs generated by EMF in the Eclipse world, so that the effort of generating the required *Cinco API* adapters is greatly reduced, which provides functionalities with identical signatures as EMF, so that already

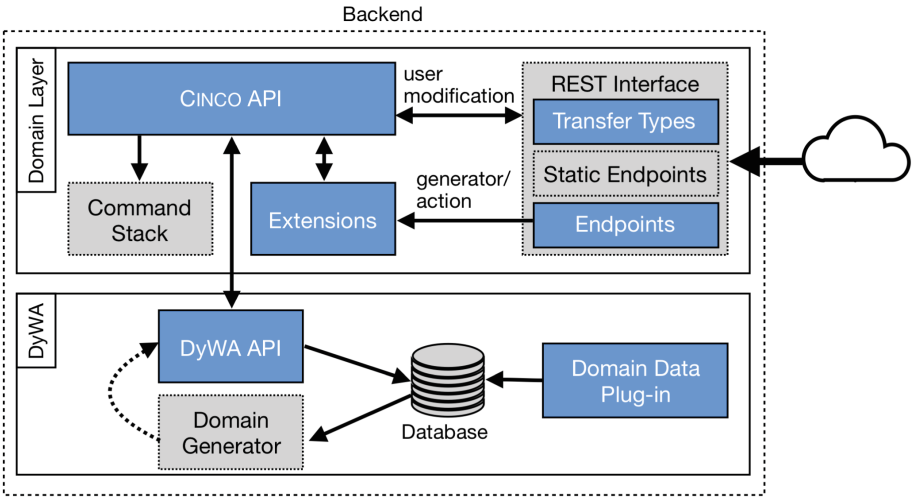

Fig. 4. Backend component architecture and interaction.

existing code can directly be applied (see below). Beyond that, DyWA is prepared for dynamic change of the metamodel, which becomes necessary during modeling language evolution (see [19]).

Since CINCO supports to extend the definition of graphical modeling languages by user-written Java code for hooks, actions, validation checks, and code generators, a holistic reuse mechanism has to be provided in the context of Pyro. To meet this goal, the same CINCO interfaces are rebuilt in the generated web-based modeling environment, providing the same structure and identical signatures. As a result of this, the domain-specific interfaces (see Fig. 4, *CINCO API*) generated by Pyro are compatible to the one CINCO generates for Eclipse and EMF to be used identically by these extensions. In contrast to the desktop-based CINCO Product, a Pyro graph model instance is not persisted in a file on the local system. The Pyro web modeling environment as a distributed system utilizes the DyWA database for storage and centralized access as a server. Thus, the *CINCO API* is internally connected to the corresponding generated *DyWA API* to persist changes in the database, which is hidden from the extensions.

Multi-user collaborative editing with the generated domain-specific modeling languages is one of the main challenges for Pyro. All changes to a centrally held instance of a graph model have to be shared with all participants. For the distribution of the changes performed on a graph model by calling the *CINCO API*) methods, a *Command Stack* is used, to store each individual modification. Since CINCO provides hooks for aspect-oriented extensions, a single action like the movement of a node on the canvas can result in multiple successive commands. As a result, all modifications on a model or any of their elements at runtime are encoded in commands and sequentially stored in the stack. The recorded commands during the *CINCO API* usage are used to synchronize between different

clients looking at the same model as well as the realization of redo and undo functionalities. This synchronization mechanism is described in more detail in Sect. 4.

To use the web modeling environment in a desktop application fashion, an uninterruptible user interaction is necessary. Thus, Pyro utilizes REST-based asynchronous communication for non-blocking data exchange. As a result of this, the outermost component of the generated web application is a *REST Interface*. The interface consists of *Static Endpoints* for project, file, and user management, which are independent from the given modeling languages. These parts are supplemented by generated *Endpoints*, which are based on the CINCO specification and provide methods to create, read, update, and delete (CRUD) a single graph model. In addition to this, the interface contains the central endpoint for commands sent from a client's frontend to the backend. Depending on the used *Extensions*, additional *Endpoints* are generated to fetch and trigger user-written actions or a generator.

3.2 Frontend

To mimic the look and feel of a local desktop modeling environment, the web-based variant generated by Pyro has to provide versatile user interactions. As a result of this, the *Frontend* of the generated web application (see Fig. 5), which realizes the interface for the user, is focused on quick responses and familiar input behavior. To achieve this goal, the frontend part of a web modeling environment is built upon the *Angular Dart* [1] framework, which is used to realize single-page web applications with built-in cross-platform support and comprises an architecture focused on reusable components. In addition to this, it is tailored to asynchronous user interaction and client-side routing, so that it can be used to build rich internet applications, like, for instance, ones resembling integrated development environments (IDEs).

In contrast to a local desktop application, a web application requires additional multi-user focused interfaces. Therefore, the template for the frontend, which is initially created, consists of static user interfaces for *Registration* and *Login* as well as a *Project Management* area to create, edit, and share projects. The specifically generated parts are used by the *Editor*, which comprises domain-specific components. Its user interface is similar to the known Eclipse IDE used by regular CINCO Products (see Fig. 2).

The challenge of preventing delays in the system's response on a user input to enable fluent interaction can be met by avoiding synchronized communication with the backend. The *editor* facilitates this frontend-side computation by two layers used to interact with instances of the graph models. The *Mirror Layer* stores a snapshot of the model present in the database, whereas the *Interaction Layer* is a direct representation of a visible graph which can be modified by the user. This separation enables a delta between the last valid graph, stored in the *Mirror Layer* and the currently visible graph. Thanks to this, generated syntactical validators (e.g., for ensuring lower bounds of given cardinalities) can

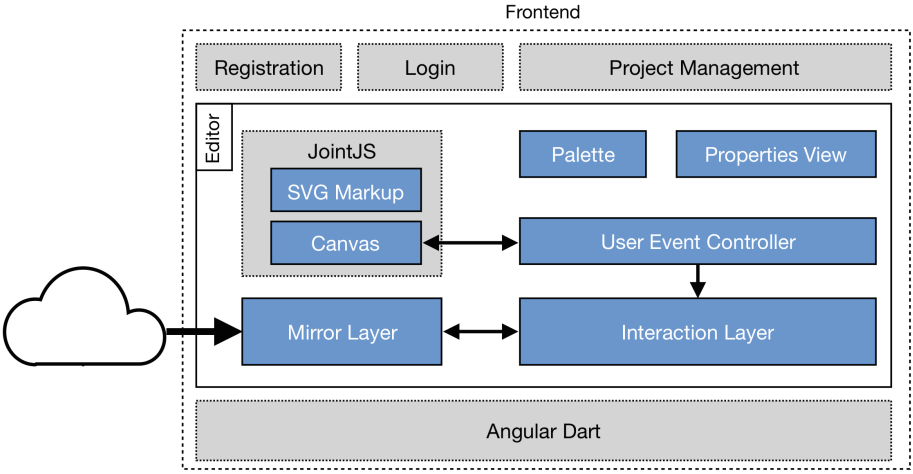

Fig. 5. Front end architecture.

raise errors and the appropriate rollback operation works immediately on the client side without additional communication with the backend.

Pyro specifically aims at supporting users switching from already existing Cinco Products to the web-based modeling environment. Thus, the *Editor*, which is the main part of the frontend, provides multiple components similar to the Eclipse IDE. To not confuse users, functions, behavior and arrangement are recreated. Besides common user interface parts like a project explorer and a menu, specific components for the modeling environment are generated, like the *Canvas*, a *Properties View*, and the *Palette*.

The *Canvas* is based on the *JointJS* framework [9], which in general renders SVGs and adds versatile user interaction for manipulation of nodes and edges via drag&drop functionalities. Using this, it was possible that the web modeling environment running in a browser provides very similar handling to the Eclipse-based desktop application with its Graphiti editor. The exact replication of the node and edge appearance is a central goal of the generated *Canvas*. Ideally, a user cannot distinguish between a Pyro and Cinco visualization of a graph model. This requires the same hierarchical shape structure for the web as in the Graphiti editor, which can be realized by scalable vector graphics (SVGs). The *SVG Markup*, which defines the shapes and styling information of the nodes and edges, is generated based on the concrete syntax specified in the MSL files of Cinco. The *JointJS* framework and *SVG Markup* files are observed by a domain-specific *User Event Controller*, which realizes the listeners and stream handling mechanisms for a single graph model to modify the underlying layers.

Besides the distinct and visible modifications available directly in the *Canvas*, attributes of an edge, node or the graph model (as defined in the MGL metamodel) can be modified using the *Properties View*. It has a generic frame based on a tree view to recursively walk through associated types of the currently

selected element. For every type present in an MGL file, a form for editing the primitive attributes (e.g string, Boolean or integer) is generated. The single fields are tailored to the specified data type of the attribute, to give as much support as possible. Thanks to the two-way data binding of the underlying Angular framework, every change to an attribute is immediately propagated to the underlying layer.

The *Palette* is generated based on the given MGL specifications. It lists all node types available for modeling. In addition to this, the optional annotations of the MGL, e.g. for grouping nodes and dedicated icons for visual support, are considered as well.

4 Collaborative Editing

One of the main features of modeling environments generated by Pyro is the simultaneous editing of graph models by multiple clients at the same time. The continuous synchronization between clients avoids classical revision control repositories for distributed access and instead enables immediate collaboration. To reach the goal of simultaneous synchronization, different aspects have been considered to maintain consistency, scalability and achieve a real-time effect.

In this section, the mechanism used for Pyro web-based modeling environments to communicate is presented and explained. The first part discusses the different challenges of a distributed system with respect to the domain of graphical modeling environments, whereas the second part describes the realization of the command pattern used to exchange modifications on a graph model.

4.1 Simultaneous Synchronization Mechanism

The main communication concept of a generated modeling environment by Pyro as a distributed system is the *optimistic replication strategy* [30]. This concept replicates data and allows the single replicas to diverge, which in the context of Pyro is realized by the separated graph model replicas held in each client. The optimistic replication belongs to the *eventually consistent* consistency model and is furthermore classified as *basically available, soft state and eventually consistent* (BASE) [36]. It benefits from high availability, since it only exchanges updates on given items. In the context of a web-based modeling environment, the updates are based on the modifications a client can do to a node or edge. To enable conflict resolution and maintain consistency regarding commutativity and idempotency, *conflict-free replicated data types* (CRDTs) are represented by commands. CRDT was originally used for text-based synchronization as a simplification of *operational transformation* [34]. It utilizes an additional data structure, based on an identifier of the client, the changed value and the position to create a unique identifier for each changed character of the text. Regarding the graph models handled by Pyro, CRDTs are realized by commands for each type of possible model element modification, which store a unique identifier and the changed properties of the relevant element. In addition to this, the previous

values of the updated properties are stored as well, to enable rollback, undo, and redo functionalities. Thus, Pyro uses operation-based and state-based CRDTs. Thanks to the CRDTs, conflicts of simultaneously editing the same model element at the same time can be detected. In the context of graphical DSLs, conflicts can arise by violating the given static semantics defined in the metamodel. If a conflict is detected, the corresponding command is flagged for rollback and returned to its sender. The client then inverts the modification encoded by the command and applies it to revert the conflicting change.

4.2 Distributed Command Pattern

The distribution of modifications made to a graph model in the Pyro web modeling environment is realized by a *command pattern* [14]. It belongs to the behavioral design pattern, which is used to encapsulate all information needed to perform an update on an object. The commands are sent as HTTP POST requests, combining the graph model and client identifier. An exemplified collaboration of two clients (red and green) modifying the same graph model simultaneously is presented in Fig. 6.

After the initial read from the database, a client only calculates, exchanges and receives commands when a modification is done (see Fig. 6(1)). For every possible change on nodes and edges (e.g., moving a node or bending an edge), a dedicated command encoding the modification is created and sent to the server, extended with a unique identifier of the sender. Thanks to this assignment, all commands can be differentiated (see red commands by client A and green commands by client B in Fig. 6). As an example, the command for the creation of a node consists of the node type, the position and an identifier of the container where it should be instantiated. Other commands, e.g., the move node commands, contain information of the previous as well as the new position, so that they store the delta of the modification.

The *Serializer* (see Fig. 6(2)) is used to parse the received payload and assign the commands to the associated *Command Applier*. Thanks to additional reflective *type* annotations, the received payload can be parsed to recreate the correct command type. The assignment depends on the given graph model type the command belongs to.

The *Command Applier* (see Fig. 6(3)) is the main component of the web server, since it receives, validates and executes the commands. Every modification encoded by a command is initially validated against the syntactical constraints defined by the graph model type. In the case of a constraint violation, the command is inverted based on the given delta, and returned to undo the invalid operation sent from a client. After a successful validation, the modification encoded by the command is applied to the generated domain-specific API, which also triggers the annotated hooks and finally modify the node or edge instances in the central database. Modifications performed on the API itself (e.g., performed by a hook implementation) are again internally encoded as commands for further distribution to other clients. The updates resulting from the hook execution inside the API are combined with the initial command to be

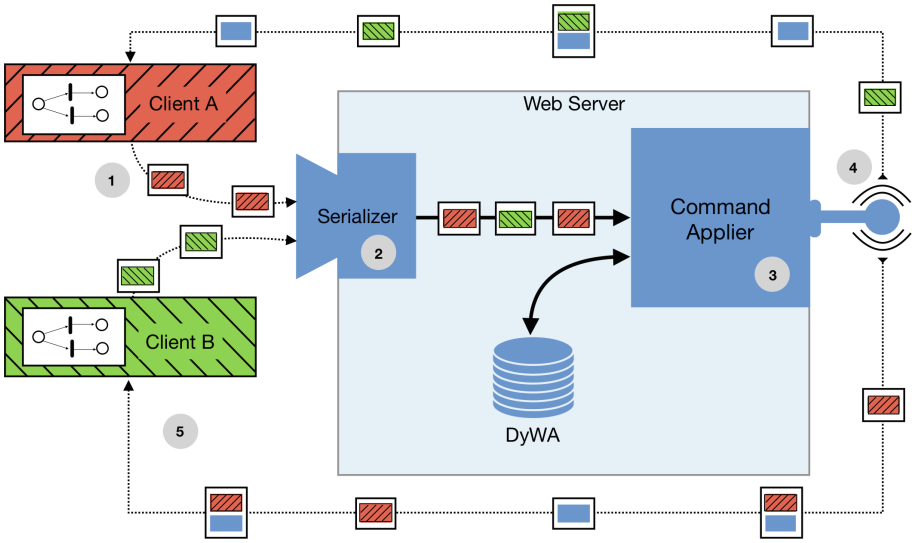

Fig. 6. Concept of the distributed command pattern. (Color figure online)

interpreted as a single transaction shown by the packages of Fig. 6. To ensure the consistency between the sender of a command and the other clients, the initiator is also informed about internally arisen modification based on hook execution. All commands, collected during the execution of the initial modification, are broadcast to other listening clients (see Fig. 6(4)). This mechanism uses bidirectional ongoing connections, so that clients can request to listen on changes made to their currently open graph model.

The commands received by a client (see Fig. 6(5)) are parsed and inspected, to ensure that commands initiated by the client itself are neglected. New changes from other clients are applied to all layers and displayed on the canvas. In addition to this, the client is notified about received changes. Updates caused as a result of self-sent commands (e.g., a modification performed during a hook execution), are only partially applied to guarantee that nodes and edges will not be modified twice.

The command pattern applied to the generated modeling environments is tailored to enable real-time collaborative editing. The main design decisions are focused on scalability and high availability by BASE and CRDT. The operational approach realized with this command pattern is more suitable than a textual language protocol like the *Language Server Protocol* (LSP) [3]. The main difference between the command pattern and the LSP is the way of distributing modifications on the model. In contrast to the presented communication protocol of Pyro, the LSP uses changed regions of a text document for propagation. The intention of the modification has to be evaluated afterwards, whereas in graphical DSLs the commands are used for a direct representation of the occurred change.

5 Conclusion and Perspectives

We have presented Pyro, a framework for enabling domain-specific modeling via the internet. Provided with an adequate metamodel specification, Pyro turns a browser into a collaborative, domain-specific, graphical development environment with features reminiscent of desktop IDEs for programming textual languages. The required metamodeling is supported in a high-level, simplicity-driven fashion: The MGL describes the available node types, edge types, and syntactical constraints, whereas the MSL defines the visual appearance of the modeling artifacts defined in the MGL. Based on these specifications, the entire ready-to-run browser-based domain-specific development environment is generated fully automatically, as has been illustrated along the construction of a graphical development environment for the Architecture Analysis and Design Language (AADL).

The field of web-based development environments is still quite young, so that not many related solutions exist yet. There are the aforementioned collaborative online text editors like Google Docs, Microsoft Office 365 and ShareLaTeX/Overleaf, but in the area of DSLs and modeling, so far we only encountered WebGME [5], an (early stage) online adaption of Vanderbilt University's Generic Modeling Environment [18] and Theia [4], a cross-platform web and desktop IDE for textual DSLs. In addition, itemis (the German company who significantly contributed to the well-known Xtext [6] DSL framework) is currently working on a platform called 'Convecton', which aims at bringing modeling with and execution of domain-specific languages online into the cloud [35]. However, none of these solutions provide a Pyro-like, graphical, collaborative modeling support.

Pyro is still in an early stage of development, and there is a lot of room for improvement, like further enhancing and easing the graphical modeling features, or improving the performance of collaborative modeling by taking advantage of peer-to-peer communication. Pyro is envisioned to enable cross-competence collaboration on a single project in a domain/purpose-specific fashion according to the Language-Driven Engineering (LDE) paradigm [31]. LDE aims at allowing the different stakeholders to formulate their intents in they way they are used to, i.e., in their domain language, and restricted in a fashion that the efforts of the other involved stakeholders are maintained, or as we say, constitute Archimedean points [32] of the considered domain-specific language. Currently, we are starting to explore the impact of the Pyro technology on a larger scale for DIME [7], our framework for developing Web applications.

References

1. About AngularDart. https://webdev.dartlang.org/angular. Accessed 13 Feb 2019
2. Graphiti - A Graphical Tooling Infrastructure. http://www.eclipse.org/graphiti/. Accessed 13 Feb 2019
3. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed 12 Feb 2019
4. Theia - Cloud and Desktop IDE. https://www.theia-ide.org. Accessed 12 Feb 2019

5. WebGME. https://webgme.org/. Accessed 13 Feb 2019
6. Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/. Accessed 13 Feb 2019
7. Boßelmann, S., et al.: DIME: a programming-less modeling environment for web applications. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 809–832. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_60
8. Carnegie Mellon University: Welcome to OSATE. http://osate.org/. Accessed 13 Feb 2019
9. client IO: Joint API. http://www.jointjs.com/api. Accessed 13 Feb 2019
10. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
11. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? June 2005. http://martinfowler.com/articles/languageWorkbench.html. Accessed 13 Feb 2019
12. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley/ACM Press (2011). http://books.google.de/books?id=ri1muolw_YwC
13. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley, Boston (2008)
14. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002). ACM SIGPLAN Notices, vol. 37, pp. 161–173. ACM (2002)
15. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley/IEEE Computer Society Press, Hoboken (2008)
16. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
17. Larson, B.R., Chalin, P., Hatcliff, J.: BLESS: formal specification and verification of behaviors for embedded systems with software. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 276–290. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_19
18. Ledeczi, A., et al.: The generic modeling environment. In: Workshop on Intelligent Signal Processing (WISP 2001) (2001)
19. Lybecait, M., Kopetzki, D., Naujokat, S., Steffen, B.: Towards Language-to-Language Transformation (2019, to appear)
20. Lybecait, M., Kopetzki, D., Steffen, B.: Design for 'X' through model transformation. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 381–398. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_23
21. Lybecait, M., Kopetzki, D., Zweihoff, P., Fuhge, A., Naujokat, S., Steffen, B.: A tutorial introduction to graphical modeling and metamodeling with CINCO. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 519–538. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_31
22. Margaria, T., Steffen, B.: Simplicity as a driver for agile innovation. Computer **43**(6), 90–92 (2010)
23. Margaria, T., Steffen, B.: Service-orientation: conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2297-5_10
24. McAffer, J., Lemieux, J.M., Aniszczyk, C.: Eclipse Rich Client Platform, 2nd edn. Addison-Wesley Professional (2010)

25. Naujokat, S.: Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools. Dissertation, TU Dortmund, Dortmund, Germany, August 2017. http://hdl.handle.net/2003/36060
26. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. Softw. Tools Technol. Transf. **20**(3), 327–354 (2017)
27. Neubauer, J., Frohme, M., Steffen, B., Margaria, T.: Prototype-driven development of web applications with DyWA. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 56–72. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_5
28. Robby, Hatcliff, J., Belt, J.: Model-based development for high-assurance embedded systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 539–545. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_32
29. SAE International: Architecture Analysis & Design Language (AADL), January 2017. https://www.sae.org/standards/content/as5506c/. SAE Standard AS5506C
30. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. (CSUR) **37**(1), 42–81 (2005)
31. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-driven engineering: from general-purpose to purpose-specific languages. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives. LNCS, vol. 10000. Springer, Heidelberg (2019, to appear)
32. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering change. LNCS Trans. Found. Mastering Change (FoMaC) **1**(1), 22–46 (2016)
33. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
34. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW 1998), pp. 59–68. ACM (1998)
35. Voelter, M.: Convecton Presentation at LangDev Meetup at CWI 8–9 March 2018. https://github.com/cwi-swat/langdev/blob/gh-pages/slides/Convecton@LangDev.pdf. Accessed 13 Feb 2019
36. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)