JOT
JOURNAL OF
OBJECT TECHNOLOGY

A flexible operation-based infrastructure for collaborative model-driven engineering

Edvin Herac[*], **Wesley K. G. Assunção**[*], **Luciano Marchezan**[*], **Rainer Haas**[†], **and Alexander Egyed**[*]
[*]Institute of Software Systems Engineering, Johannes Kepler University, Austria
[†]Linz Center of Mechatronics GmbH, Austria

ABSTRACT Collaborative model-driven engineering infrastructures are used to concurrently deal with models from diverse engineering domains. However, studies within the industry have shown that existing infrastructures often offer insufficient collaborative support or do not incorporate features to handle multiple engineering domains, which leads to inefficient collaboration and reduced team productivity. In this paper, we propose a flexible operation-based infrastructure for collaborative model-driven engineering that addresses these issues. The infrastructure supports lock-free collaboration within co-existing metamodels of different domains. It also allows engineers to arbitrarily push or pull changes of divergent versions and to deal with appearing conflicts. To achieve those functionalities, the infrastructure relies on operations that represent atomic changes on our simplified version of the Meta-Object Facility, such as creating and deleting new types, instances, or modifying properties. The infrastructure manages those operations within a tree-like structure that grows incrementally. Sequences of operations, going from the root to a leaf, represent different versions of a complete model history (including metamodels). Model versions can be merged by concatenating their corresponding branches and handling occurring conflicts. We evaluated the infrastructure by applying it in practical scenarios in collaboration with an industrial partner. These scenarios demonstrate the feasibility of our infrastructure by complying with our derived requirements for collaborative model-driven engineering infrastructures.

KEYWORDS collaboration infrastructure, model-driven collaboration, collaborative engineering, synchronization, conflict handling, co-existing metamodels.

1. Introduction

Current engineering practices to create complex systems rely on highly interdisciplinary teams, potentially globally distributed, working with heterogeneous artifacts (Bucchiarone et al. 2021). For instance, in a robotics project, collaboration from multiple engineers across different domains such as mechanical, electronic, and software is required (Egyed et al. 2018). However, achieving proper collaboration to correctly and efficiently develop complex systems is not a trivial activity (Zissis et al. 2017; Franzago et al. 2018; David et al. 2021; Tröls et al. 2019). The artifacts developed in each domain, usually represented as models, use different structures (e.g., metamodels) and are managed in different tools, but somehow related to each other (Muccini et al. 2018). Yet, even in the same domain, engineers may have different preferences regarding which tools to use (Torres et al. 2020). For instance, a software engineer may choose to work with Eclipse IDE[1] on source code, whereas another one can decide for using IntelliJ IDEA[2] to work on the same source code or in combination with Visio[3] on a class diagram, using the same or different metamodels.

Engineers can also have different preferences regarding the frequency in which they want to synchronize (push or pull) their changes (e.g., make their changes visible to others or receive changes from others). However, this can lead to time-consuming reservations of artifacts (locks) or to conflicts due

JOT reference format:
Edvin Herac, Wesley K. G. Assunção, Luciano Marchezan, Rainer Haas, and Alexander Egyed. *A flexible operation-based infrastructure for collaborative model-driven engineering.* Journal of Object Technology. Vol. 22, No. 2, 2023.
http://dx.doi.org/10.5381/jot.2023.22.2.a5

[1] Eclipse: https://www.eclipse.org/ide
[2] IntelliJ: https://www.jetbrains.com/idea
[3] Visio: https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software

to concurrent changes (e.g., synchronizing different versions of not locked models). In this context, we argue that engineers lack a collaborative model-driven engineering infrastructure that supports *multi-domains* while providing *synchronization flexibility* so that they can choose how and when to integrate their work lock-free into the project.

To provide the infrastructure mentioned above, however, we need to address several challenges. First, we need a common representation of diverse artifacts such as a meta-metamodel, that enables the generation of a multitude of metamodels. As a result, this meta-metamodel allows us to handle co-existing multi-domain artifacts (i.e., multiple metamodels). Second, such an infrastructure must enable multiple users to share their work, while providing lock-free versioning and conflict detection. Third, a versatile infrastructure must offer engineers the flexibility of choosing when to make their work available to other engineers (i.e., push their changes) and when to get the most recent version of the artifacts (i.e., pull their artifacts with changes from other engineers). We also should not rely on only a fixed set of tools for the infrastructure, as users may have preferred tools that lead them to be more productive. Thus, an interface for users to adapt arbitrary tools for collaboration is desired.

In this paper, we propose a more flexible model-driven infrastructure for collaborative engineering, that supports lock-free multi-domain collaboration due to the handling of co-existing metamodels. The infrastructure is based on operations that enable us fast communication, fine granular conflict detection, and resolution during merges of different versions (Yohannis 2020). Our infrastructure enables engineers to work with adapted tools and arbitrary synchronization triggers, independently of their working domain or the number of engineers. Hence, the infrastructure provides an interface for the possible reuse of familiar single-user applications. Adapting them to become collaborative applications contributes to higher collaboration acceptance and faster delivery cycles, as engineers can still work with their preferred tools. However, adapting tools for collaborative purposes can be a complex task (Sun et al. 2006). The advantage of our proposed infrastructure includes the flexibility of co-existing metamodels and versions while providing generic conflict detection during merges. Additionally, our infrastructure allows engineers to choose their synchronization triggers (e.g., when to push or pull their changes).

To show the infrastructure's applicability, considering its main features, plugins (extensions of single-user tools) and metamodels for a set of engineering tools were created. We describe the use of the adapted tools in four practical scenarios, including two with an industrial partner. These practical scenarios constitute cases, with a different number of domains, engineers, tools, and configurations. The infrastructure was able to provide the co-existence of metamodels while providing synchronization flexibility among multi-users, addressing the limitations of existing work.

The remainder of this paper is structured as follows: Section 2 presents an illustrative example and the main requirements desired in collaborative systems. Section 3 describes how the infrastructure is designed to address the requirements. The scenarios to evaluate the infrastructure are presented in Section 4. Section 5 shows different attempts to implement those main requirements. Lastly, Section 6 concludes our work, presenting the future steps of our research.

2. Motivation and Problem Statement

Based on studies conducted in industry (Liebel et al. 2018; Cicchetti et al. 2016; Jongeling et al. 2019, 2022), reports from the literature (Franzago et al. 2017; Torres et al. 2020), and our experience with industrial partners,[4] we argue that existing collaborative engineering tools/systems have insufficient support or even missing collaboration features. To detail these problems, this section describes an illustrative example of a collaborative engineering scenario and the core requirements (Req) that should be implemented by a collaborative infrastructure. The example relies on the project of a heartbeat monitor device used in hospitals to inform the medical staff in case a patient's heart stops beating.

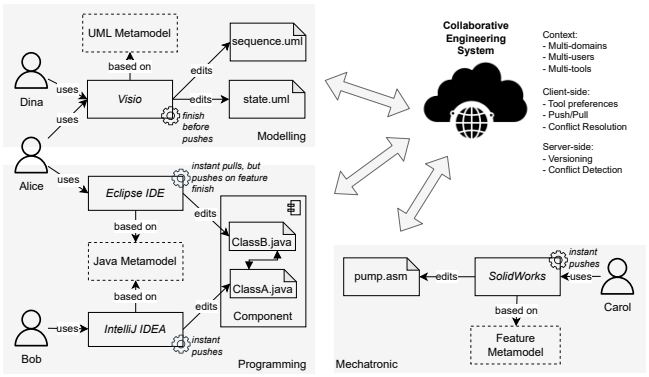


Figure 1 Collaboration context for the development of the Heartbeat Monitor Project.

Figure 1 presents the overall context of the project development. The heart monitoring device is composed of both software and hardware components. More specifically, the project consists of three different domains:

Modeling: Devoted to the specification and designing of the structure and behavior of the heartbeat monitor. In this domain, the main concerns are to reason on how to design functional requirements (e.g., notifying medical staff in case a patient's heart stops beating) and non-functional requirements (e.g., 24/7 operation and elimination of external interference such as mobile phone networks). The stakeholders of the project decided to rely on the Unified Modeling Language (UML), thus, the design models are based on the UML metamodel.

Programming: Focuses on the realization of the structure and behavior defined in the modeling domain. The goal is to create a concrete piece of software for the heartbeat monitor. For that, the stakeholders chose to use the Java programming language, creating the source code according to the Java metamodel.

[4] Especially with Linz Center of Mechatronics GmbH, Austria. https://www.lcm.at/en/

Mechatronic: Has the goal of designing the mechatronic plan of the heartbeat monitor. It describes which electric components (e.g., resistors, transistors, and opamps) are used or how the machine parts look like (e.g., the features of a machine part) and how they are assembled. The stakeholders decided to use SolidWorks, which creates artifacts following the feature metamodel.

We can see that the heartbeat monitor is a *multi-domain* project, in which practitioners have to deal with artifacts/models in three co-existing domains in accordance with the UML, Java, and Schematic metamodels. Thus, a requirement is:

> **Req1**: *Infrastructures for collaborative model-driven engineering must support work in multi-domains by coping with co-existing metamodels.*

In addition to multi-domains, to achieve the goal of the collaborative project, engineers have to work together on related artifacts. For example, in Figure 1, three engineers, namely Alice, Bob, Carol and Dina, work collaboratively. We can observe that even in the same domain (i.e., programming), Alice + Bob and Alice + Dina, work in the same artifacts (i.e., Java source code or UML Models) concurrently without any reservations or locking each other out. This configures a *multi-user* collaborative environment. In addition to multi-domain, supporting multi-user is a functional requirement of a collaborative model-driven infrastructure. Hence, we can define another requirement:

> **Req2**: *Infrastructures for collaborative model-driven engineering must support concurrent work with multi-users by allowing versioning and lock-free work.*

In collaborative environments, engineers may have different collaborative work styles. For example, Alice and Dina, with Visio, may prefer to first finish their models related to one feature before pushing them (making their changes visible to the other engineers). Nevertheless, when Alice is working with source code, her preference is to always pull from the remote to keep her code in sync (locally) with visible model changes from her colleagues. Alice still, however, only pushes her code after one feature is finished. Similarly, Bob prefers to always push his work instantly to stay in sync with changes because a push involves always a preceding pull. Carol may desire to configure her tool to also always stay in sync like Bob. Hence, we argue that providing the possibility to work lock-free (no waiting time) while allowing arbitrary pull and push is important to cope with engineers' collaborative preferences. This enables users to have instant pulls while only sporadically pushing their work. Most tools, however, focus on one single type of collaborative work style (e.g., only synchronous or asynchronous work (Pietron et al. 2021)). Thus our third requirement:

> **Req3**: *Infrastructures for collaborative model-driven engineering must allow engineers to pull and push their models anytime to cope with engineers' collaborative work style preferences along the engineering cycle.*

Despite having three different metamodels, to work on the artifacts, engineers have a myriad of tools available. For example, the engineers responsible for the modeling domain, Alice and Dina, work with Visio for the UML models. In addition to dealing with UML models, Alice also works with Java source code, preferring to use Eclipse IDE as a source code editor. Bob is also an engineer working with Java code, but he prefers the editing support offered by IntelliJ IDEA. Finally, for dealing with the mechatronic models, Carol decided to use SolidWorks.[5] This situation illustrates that in collaborative engineering, users should be able to choose which tool they prefer or are familiar with. It would be hard to define a guideline for how a multi-tool environment or a tool interface should look like, regarding the vast amount of tools and applications on the market. Thus, we do not define it as a requirement. The underlying idea of a tool interface is to allow the infrastructure to be used in an already existing tool landscape. It allows engineers to develop new tools and adapt their preferred tools for their own purpose. Therefore, we defined an interface for our infrastructure to cope with a multitude of possible scenarios which we consider sufficient.

3. Proposed Infrastructure

Our proposed solution is an infrastructure that supports co-existing metamodels. This is achieved by providing our own simplified Meta-Object Facility (MOF) (Object Management Group 2022) and representing it with a history of change operations that are managed in a tree-like incrementally growing structure, as described in Section 3.3. Due to the lock-free concurrent multi-user environment, model versioning is essential (Altmanninger et al. 2009). Each engineer has to modify their own replicated version of the model until the versions are merged and the change conflicts are presented in Section 3.4. Our infrastructure provides flexibility for collaborative work style preferences. It allows engineers to arbitrarily push or pull changes (i.e. merge model versions), as explained in Section 3.4.1. Also, different types of tools, applications, standalone products, and services can be integrated into our infrastructure at any time in the engineering cycle, explained in Section 3.1.

3.1. Overview

We configured the infrastructure as a two-tier client/server system (Oluwatosin 2014). The server handles all sessions of the clients and their requests. Hence, acting as the central node for handling all connections, versions, and changes (i.e., acts as a central global unified state). The client/server configuration allows us to centralize versioning and conflict handling, reducing the infrastructure's overall complexity. The server uses the paradigm of workspaces to handle model versions.

A *Workspace* acts as a container for a specific state of co-existing metamodels and model versions, and all their not pushed local changes. Its state is represented by a token on a node of the operations handling tree-like structure. All data can be held in an arbitrary number of private workspaces and one central public workspace (parent of all private workspaces, shared by all engineers). All connections from clients to the server are aiming for a chosen private workspace. Only one engineer can be connected to a private workspace simultaneously,

[5] SolidWorks: https://www.solidworks.com/

in order to prevent inconsistencies and unmanaged conflicts. The public workspace is a central node (i.e. parent workspace) that allows the private workspaces to share their changes with each other through pull and push commands, visible in Figure 2.

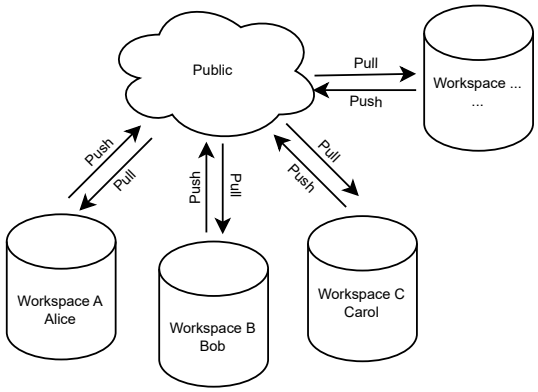


Figure 2 Workspaces collaboration star topology.

In contrast to the server, all clients are based on and connected through our infrastructures software development kit (SDK), see Figure 3. The SDK provides clients with a connection handler, an invisible layer for the synchronization to the server, and an object-relational mapping (ORM) of our simplified MOF for any kind of data modifications (Keith & Schnicariol 2009).

A client can connect to an existing private workspace or create a new workspace at any given time. That connection starts as a stream where the client replicates the private workspace's model. In the case of a new workspace, the current public workspace's model version is replicated, allowing us a lock-free work environment for the new private workspace. Due to the incremental growing history tree, described in Section 3.4, no locks on the public workspace are required during replication. Consequently, private workspaces grow to branched versions of the public workspaces model. Branching branches (e.g., private workspaces with child private workspaces) is future work and is omitted in this approach due to complexity reasons.

We use gRPC[6] for the communication between the server and the clients. The server uses a first in, first out (FIFO) connection, which ensures the right order of incoming requests. However, every engineer works on his/her own workspace, thus no conflicts can be created due to local changes. Conflicts can only appear during pushes or pulls, in which case the conflict handling intervenes. Client change requests are executed on the server, and the resulting operation sequence is sent over to the client. This creates a stream of changes from the server to the connected client to keep them consistent with the central source of truth.

The infrastructure is composed of the server (where an instance of our implemented infrastructure is running) and the connected clients with their adapted engineering tools, illustrated in Figure 3.

[6] gRPC - Remote Procedure Call Framework. https://grpc.io/

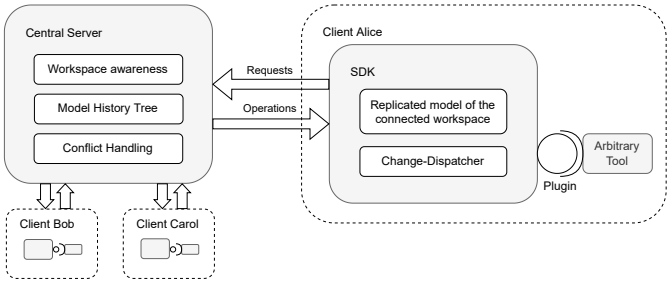


Figure 3 Overview of the main components.

Workspace awareness: The server is aware of all existing workspaces, their current version state in the model history tree, and their connected clients.

Model History Tree: Persists all model changes as a history of operations in a tree-like incrementally growing structure. Complete branches of this tree represent different variants of a model with workspace states as tokens typically at the leave nodes. Every node on the branch represents a model version.

Conflict Handling: When two branches are merged (one private and the public workspace) through a pull command, a conflict handling analyzes the corresponding branches' operations to detect possible conflicts and resolves them by creating resolution operations. These operations are added to the leaf node of the private workspace's branch in the model history tree (i.e., new private changes).

SDK: The SDK uses ORM for converting the model data to the heap of an object-oriented programming language. This allows us to use a virtual object database within Java or C# to modify metamodels and models. The client SDK receives changes (i.e., operations) that are executed on the locally cached model and notifies the plugins of its changes. In other words, it provides an invisible layer between the server and the client. The SDK can be used to create plugins for arbitrary tools. These plugin implementations can combine various domains and functionalities like implementing scenario logic or combining co-existing metamodels to a multi-view. Thoroughly showcased in Section 4.

Replicated model of the connected workspace: Every client has its own replicated model of the connected workspace locally cached. The server is the central source of truth for its state. So any changes to the model on the server are reflected in the client by receiving and executing the received operations locally.

Change-Dispatcher: The change-dispatcher executes all incoming operations from the server to local changes on the cached model. It also transforms the changes on the locally cached model (by the ORM and the plugin) into requests to the server.

3.2. Workflow

In this section, we illustrate the workflow of our infrastructure, to understand the basic behavior of the components and how engineers may interact. As shown in Figure 4, Alice and Bob are connected to their private workspaces with arbitrarily

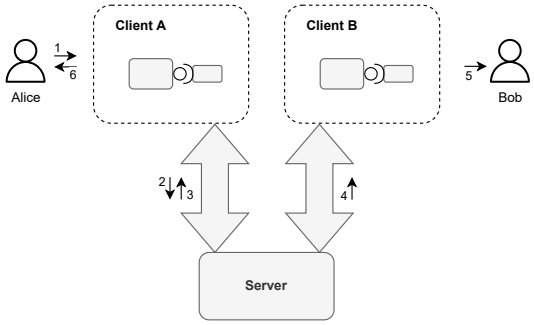


Figure 4 The workflow of collaborating engineers.

adapted engineering tools. They are working synchronously (pushing and pulling as soon as there are changes). Next, we describe a simple workflow that may happen when considering this scenario.

Step 1: Alice connects to her workspace and starts changing an artifact inside her tool. Alice's tool reacts to the change and sets the value inside of *Client A* cached model through the SDK.

Step 2: The SDK transforms the model changes into a request and sends it to the server. It is also important to execute the change locally too, without waiting for the response operations. This leads to a faster response time. The server receives the change request, creates operations, and adds those changes to the leaf of the corresponding workspace's branch of the model history tree.

Step 3: The connected *Client A* is notified of the new operations. It receives the same operations that are already cached. In case something went wrong, those operations could change the model back to the old state. Alice pushes those changes instantly to the public workspace as intended by the plugin.

Step 4: *Client B* pulls as soon as there are new visible changes on the public workspace and receives an operation sequence which is executed on its local model. However, in case of conflicts, the conflict handling activates, described in Section 3.4.

Step 5: The adapted tool of *Client B* pulls the internal artifacts corresponding to the pulls local model for Bob.

Step 6: Alice keeps working on her tool.

3.3. Model Management

Our data structure is persisted as a change-based model history. That technique is called change-based persistence (CBP) (Yohannis et al. 2017). This makes all model changes be represented and persisted as a finite sequence of changes, instead of saving their current model state (state-based) like EMF (Steinberg et al. 2008) uses (e.g., Ecore files). State-based persistence saves and loads the whole model. Thus it is computationally expensive and can specifically affect large models and collaboration. CBP enables faster detection of changes and speeds up the merging of model versions (Ráth et al. 2012; Ogunyomi et al. 2015). However, CBP can still build up large and ever-growing change history files, where loading times get significantly higher and the current model state has to be reconstructed from the model history (Yohannis et al. 2017).

The persisted metamodels and models can be handled in different ways (e.g., MOF or EMF). Meta-Object Facility (MOF) is a standard specification developed by the Object Management Group (OMG). It defines metamodels in the field of software engineering by providing a framework for creating, manipulating, and exchanging metamodels and models (Object Management Group 2022).

By utilizing our simplified MOF, we provided students with a more accessible and understandable approach (but not as expressive) to designing metamodels for various tools. The simplified MOF still offers sufficient expressive power to implement a wide range of metamodels for our evaluations and industry cases. Our primary goal is to strike a balance between simplicity and expressive power (to our experience) to facilitate metamodel design and usage in our infrastructure and to show feasibility of combining the three requirements for collaborative engineering infrastructures.

All changes on the metamodels and models generated from our simplified MOF are represented by operations defined by our operation model, illustrated in Figure 5. The proposed simplified MOF can be considered an explicit definition of how a domain-specific metamodel is built. It is the key data structure to our domain flexibility (**Req1**). We created a simplified MOF to be capable of representing a wide variety of software engineering artifacts, as shown in our evaluation scenarios. A more complex MOF would lead to a more complex operation model and henceforth, to a higher overall system complexity. It is an individual design decision between MOF completeness and complexity to handle our evaluation scenarios and still be capable to comply with all our infrastructure requirements.

Every client can create, change or delete a metamodel and instantiate corresponding models within their private workspace at any time. Each workspace can contain its own metamodels and models, which can be shared by pushes and can be received by pulls from the public workspace. Meta-metamodel classes illustrated in Figure 5 are defined as follows.

Element: Is the basic abstract type of the metamodel and cannot be instantiated. Every *InstanceType* and *Instance* is an *Element*. It contains a unique *id* that is generated from the server. Any request (e.g., create *InstanceType*) is sent to the server and executed there. Thus the support of a centralized *id* generation.

InstanceType: The clients can create metamodel with *InstanceTypes*. Multi-inheritance is allowed. In this case, inheritance means to inherit all *PropertyTypes* of all supertypes recursively. If multiple *PropertyTypes* exist with the same name, only the first one will be inherited.

Instance: The instantiated *InstanceType* containing *Property*s transformed from the *InstanceType*s *PropertyType*s.

Property: Contains the property name and object values, matching the *Cardinality* and the *PropertyType*.

PropertyType: The type that represents an instantiated *Property*. It may be an object value or a reference to another *Instance* (REF). It is used to define which properties an *Instance* gets, from a particular instantiated *InstanceType*

Cardinality: The cardinality of a *Property*. Any *PropertyType*

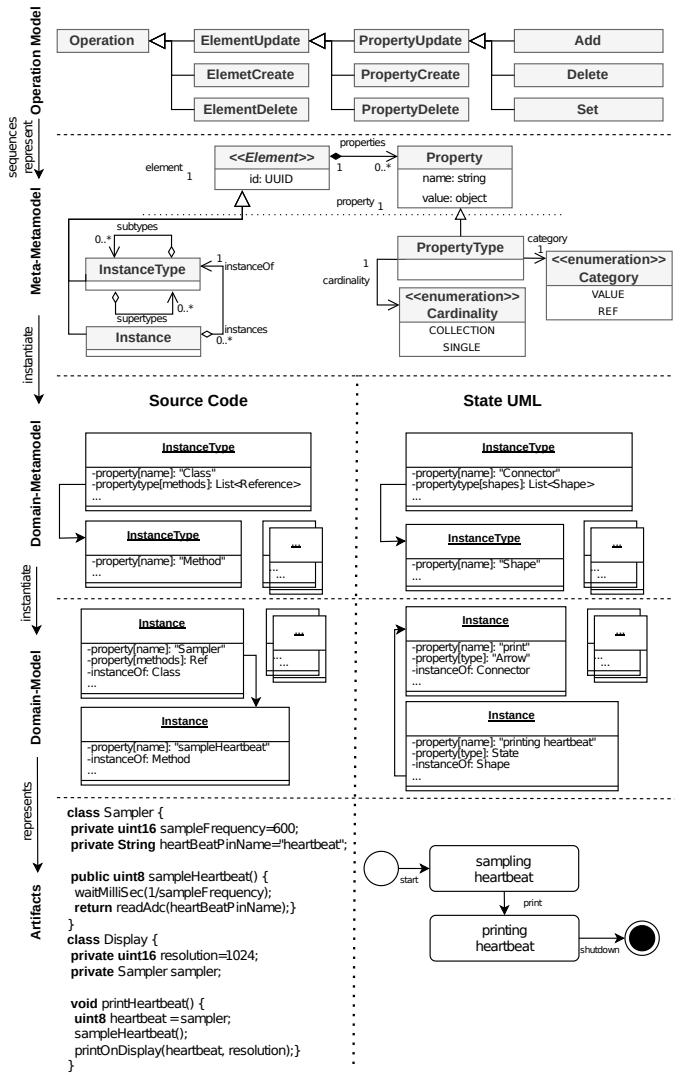


Figure 5 Operation-Model and our simplified Meta-Object-Facility (MOF) with its representation of artifacts.

can be mapped either a single value or a collection of values with different constraints.

The instantiation of an *InstanceType* is the most important aspect of the metamodel. Instantiating an *InstanceType* means generating a real *Instance* of an abstract *InstanceType*. During instantiation, all *PropertyType*s of an *InstanceType* and its supertypes are transformed to properties and related to the *Instance*. Thus, an *InstanceType* can have its own properties and *PropertyType*s. *InstanceType*s and *Instance*s can be created arbitrarily during runtime.

3.4. Versioning and Conflict Handling

Enabling parallel development is crucial in large-scale software development (Perry et al. 2001). To achieve this, a common approach is to permit the modification of one artifact concurrently and merge the distinct modifications into a new version of the artifact. Consequently, version control systems (VCSs) must facilitate a merge process for concurrent evolved software engineering artifacts. Thus, our infrastructure uses operations to represent versions of a model, transmit, and cache them. Any possible state in the model history could be replicated at any time. Our operation model contains a fixed amount of operations (see the bottom part of Figure 5). Those operations give us the possibility to change models generated from our simplified MOF and merge models with conflict detection and resolution.

We use incremental sequences of operations in a tree-like structure to represent diverging model versions. Simple lists cannot illustrate competing versions (i.e. branches) because concurrent changes on the same list cannot be persisted. Therefore, we use a model history tree where every node is an operation, and every branch acts as an alternative change sequence, i.e., another model version (**Req2**). The model history tree supports branches and merging of branches, i.e. merging of model versions, similar to the proposed approach of Berlage et al. (Berlage & Genau 1993). A branch can be created by a new change on a predecessor node independent of other siblings (e.g., branching nodes). When a new private workspace is created, it receives the same node state in the model history tree as the public workspace. A new change adds a new child node to the current workspace's state node.

3.4.1. Push & Pull

Engineers might prefer sometimes to not make their changes visible to others, thus they keep their changes in their private workspace or to make them visible for others by pushing them to the public workspace. Engineers can select their preferred collaborative work style flexibly (**Req3** in Section 2). For that purpose, we offer a flexible lock-free pushing and pulling.

To receive new changes from the public workspace, the private workspace has to pull. To make changes available to the public workspace, the private workspace has to push. In the case of a pull and already existing changes on the private workspace, all operations will be conflict detected and resolved on the central server. Necessary corrections will be sent to the users' private workspace. Every push executes a pull beforehand to synchronize the private workspace with the public including conflict resolution. The resolving operations can finally be pushed to the public. At that point, the private and public workspaces are in the same model state.

Figure 6 shows the three possible state exchanges between the public and a child workspace after arbitrary pull and push commands. The left push command executes a nonconflicting push to send its state to the public workspace. The middle pull command executes a nonconflicting pull to the private child to get into the same state as the public workspace. The right push is executed in conflicting states of the public and private workspaces. It executes a preceding pull to solve the conflict locally on the private workspace which creates a new nonconflicting state. That state is pushed to the public. A pull command on conflicting states is executed in the same way with the difference of not pushing the resolved state to the public.

3.4.2. Model History Tree

Figure 7 represents the behavior of a visually simplified model history tree for a collaborative scenario. Every node is an operation and the edges are pointing to the previous change operation. A branch is a different model

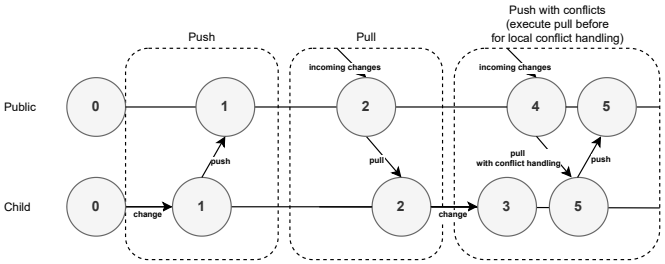


Figure 6 Model state behavior during push and pull commands.

history sequence and every blue circle represents a state token of the labeled workspace.

To get a specific model state, the SDK executes all operations of the corresponding workspace from the root to its state node. For example, if we take Figure 7a, the public workspace *P* operation sequence would be (P1, ..., Init). For the workspace *A*, the operation sequence would be (P1, ..., Init, A1, A2, A3, A4).

Depending on the arbitrary changes, pulls, and pushes of the engineer the model history tree behaves differently. Figure 7a, shows the initial operations of the public workspace. They contain all operations for the creation of our simplified MOF (e.g., *ElementCreate(InstanceType)*, *ElementCreate(Instance)*, etc.). Following the additional three branches to the final state nodes of workspaces A, B, and, C we can extract the corresponding sequences for a complete model state recreation.

After a push from workspace B, we receive Figure 7b. Only the public workspace state token switched to the same state as the workspace B token. Displaying the same model state. The public workspace was in no unknown state for workspace B, thus no conflict handling or change propagation was necessary. Subsequently, workspace A decides to pull the new visible changes from the public workspace. However, the changes (A1, A2, A3, A4), e.g., creating the "Shape" InstanceType, and (B1, B2, B3), e.g., creating the "Class" InstanceType, had no conflicting operations (e.g., just created a new metamodel for the uml diagram and a new metamodel for java source code). The whole branch of workspace A is reattached to the state node of the public workspace. The changes (B1, B2, B3) are propagated to the connected clients of workspace A. However, workspace A finally adds a new change (A5) to the newly received model from workspace B visible in Figure 7c, e.g., changing the name of a "Class".

After that workspace C decides to also pull the new changes from the public workspace and add a new change to it (C5), e.g., changing the name of the same "Class". The tree behaves just in the same way as in the pull from workspace A but we receive Figure 7d.

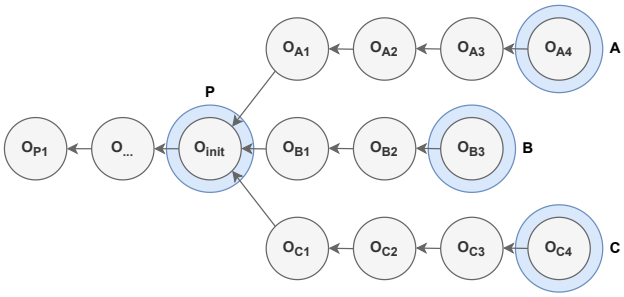
Figure 7e can be achieved, when workspace A pushes all changes to the public workspace and workspace C executes a subsequent push. The public workspace receives the same state as workspace A and afterward tries to conflict handle the two new changes (A5) and (C5). The conflict is recognized by the implemented conflict detection (detailed in Section 3.4.3) and resolved automatically by adding a new resolution operation (CR) (see Section 3.4.4) to the private workspace C. Workspace C instantly pushes the (CR) to the public workspace, where workspace A can pull it again. Finally, workspaces P, A, and B receive the same state of the model (P1, ..., Init, B1, B2, B3 A1, A2, A3, A4, A5, C1, C2, C3, C4, C5, CR) representing the engineers' metamodels and models. This scenario describes how changes are transmitted from private workspaces over to the public workspace, and then to other private workspaces including conflict handling.

3.4.3. Conflict Detection

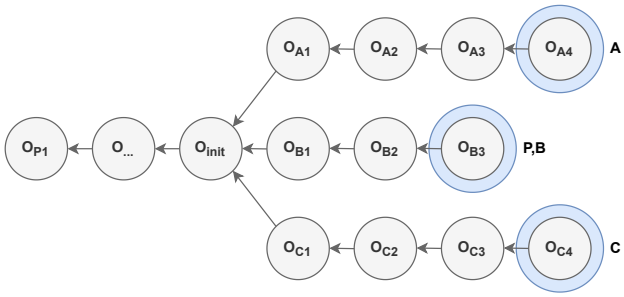
On every pull command of a private workspace, we detect conflicts between private and public workspace change operations. Our conflict detection mechanism is similar to how Yohannis et al. (Yohannis 2020) compare operation sequences. It iterates two operation sequences (branches) with a common predecessor (branching) node, typically the public state node. For instance, let us take the Figure 7d and try to merge A and C. The conflict detection scans the operation sequences of (A1-A5) and (C1-C5) for conflicts. Based on our meta-metamodel (Section 3.3) conflicts can happen when operations change the same element or the same property of the same element. If such a case occurs, our detection logs the corresponding operations from both branches, because they are indicating possible conflicts. Conflicts are defined by our conflict specifications in three cases: **Case 1**: The same element is deleted in both branches; **Case 2**: An element is deleted in one branch and the other branch creates or changes a property of that same element; and **Case 3**: The same property of the same element is changed or deleted in both branches. Those cases are delivered to conflict resolution, which decides how they are handled. Currently, our infrastructure does not include a feature to display conflicts to the users. This is primarily because our focus has been on providing a solid foundation for collaborative modeling and addressing the core requirements of tool integration and conflict management. We have utilized existing conflict detection algorithms (Prakash 1999) as part of this work.

3.4.4. Conflict Resolution

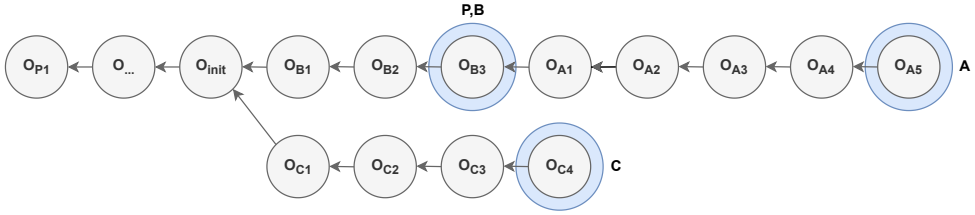
The conflict resolution computes overwriting changes for the private workspace, as visible in Figure 7e. Typically engineers can decide what to overwrite during a merge in a collaborative system. Single values are overwritten by repeating an operation and collection values are overwritten by undoing all changes and repeating them in the correct order as mentioned in (Prakash 1999). In the evaluation of our infrastructure, we employed a simple approach where a flag was used to determine which user's changes would overwrite the conflicting changes. This automatic conflict resolution was implemented in the backend, without exploring different conflict awareness or notification modes. The focus of this proposed infrastructure is not on interactive conflict management. It is simply to show, that conflicts can be detected and handled. However, future work will delve into how users can actively engage in conflict resolution and explore various conflict management strategies.



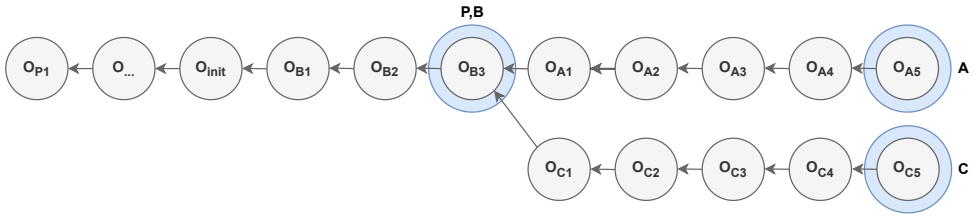
(a) Initial tree of the model history and its workspace states.



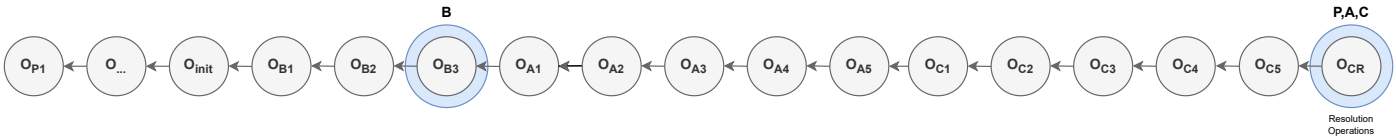
(b) After a push from workspace B.



(c) After a subsequent pull from workspace A and a new change.



(d) After a subsequent pull from workspace C and a new change.



(e) After a subsequent push from workspace A and afterward a pull from workspace B with a local conflict resolution of the new previous changes.

Figure 7 Behavior of a visually simplified model history tree for a collaborative scenario.

4. Evaluation

In this section, we present practical scenarios and also discuss the complexity and performance of our approach. Every engineer can work on single or multiple metamodels or models with different interests reflected by different plugin implementations. Henceforth, we want to give an insight into the challenges we observed and had to solve while implementing the scenarios and adapting the engineering tools.

As mentioned in Section 2, adapting tools can be a complex and tedious task (Sun et al. 2006). Nevertheless, engineers like their accustomed environment of tools so providing an interface for them can be beneficial. Engineers can work with their artifacts with any tool they want. To provide such a feature, the required tools must be transparently adapted to the collaborative environment with a technique called *Transparent Adaptation* (Sun et al. 2006). Such an adaptation can be faulty and take a long time due to the complexity of the artifacts and the aspects of collaborative work. Assessing the exact effort required for plugin implementation can be challenging due to various factors, including the complexity of the tools' APIs, the specific scenarios we considered, and the available documentation for the tools. Each tool may have its own intricacies and learning curve, which can impact the development effort. However, it is important to note that our focus in this work was to demonstrate the feasibility of our approach rather than providing a comprehensive analysis of the effort involved in plugin development. We acknowledge that developing plugins for complex tools can be a challenging and time-consuming task.

However, we implemented some transparent adaptation plugins for different tools to use our infrastructure. During the implementation, we observed many obstacles regarding the tools SDKs and the adaptations that were not mentioned by Sun et al. (Sun et al. 2006).

Our scenarios are more complex and have flexible collaborative styles that caused us implementation challenges. Those challenges could give some insights into the effort that is required for the transparent adaptation.

4.1. Challenges

A major challenge was the *Granularity of events* where the SDKs could only deliver coarse granular events. For example, when a "method name" is changed, a "class changed" event is fired. Such problems could lead to performance issues. On the other hand, an even more complex problem is too fine for granular events. For example, when a "method name" is changed, around twenty independent, possibly not ordered, events for each character changed are fired. Furthermore, problems like the *Application state*, where the user is in some unpredictable state (e.g., OpenFileDialog), cause the application to lock and some SDK calls will not work leading to crashes. The most difficult problem was the *Thread safety* of the tools or SDKs. Since the plugin listens asynchronously to incoming operations of the infrastructure in its own thread, the MainThread thread must be thread-safe. Otherwise calls and changes from the plugin thread lead to exceptions and crashes. This is especially important for CAD solutions since they are developed for single users and were never required to run asynchronously or with multi-users.

We were able to address these problems and adapt the infrastructure to provide thread-safe access to the tools and the required object model space of the application. The following section shows you engineering scenarios where we utilize the adapted tools.

4.2. Practical Scenarios

In this section, we present four scenarios. Two of them were applied in collaboration with an industrial partner. These scenarios are used to demonstrate how the main features of our infrastructure are used for different domains, tools, and a number of engineers, allowing different preferences for collaborative work styles.

Single domain/tool with multi-user concurrent work: One of the simplest scenarios (used only for demonstrative purposes) is when a single tool needs to be used by multiple users. The scenario, presented in Figure 8, uses two instances of Visio that work in a collaborative environment. In this scenario, both instances are applying pushes and pulls as soon as there are changes available either on the public or private workspace. This means that any change performed in one instance is propagated to all other instances almost instantly. This configuration allows two engineers to work on the same project and see the changes from their colleagues in real-time. In our project repository, we also demonstrate other scenarios applying our infrastructure using multiple instances of 4diac and IntelliJ.[7] This simple scenario already demonstrates how our infrastructure addresses (**Req2**), multi-users and allows synchronous work (**Req3**) if preferred.

Collaboration in multi-domain/user/tool with co-existing metamodels: One scenario in cooperation with our industrial partner[8] where our infrastructure is applied allows collaborative

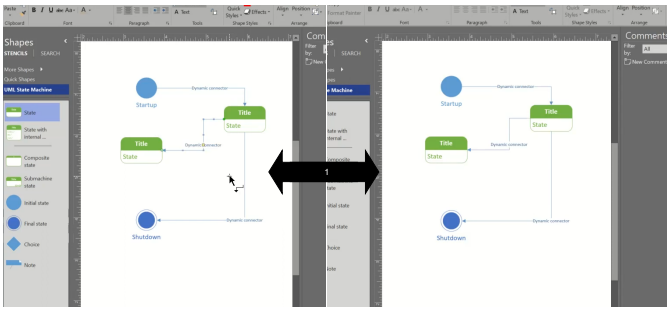


Figure 8 Collaboration between two instances of Visio working on a UML state machine.

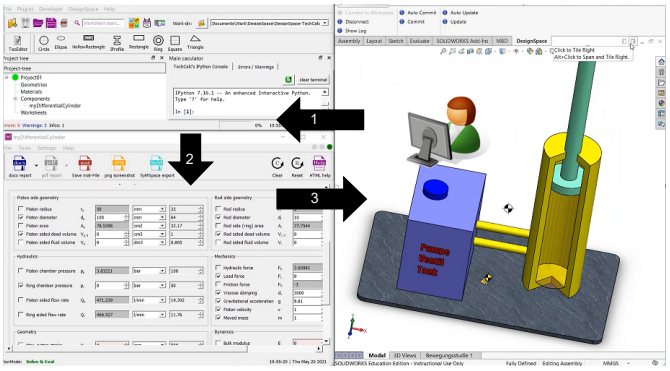


Figure 9 Collaboration using Techcalc and SolidWorks with two different metamodels.

work with tools from different domains. In this scenario, SolidWorks is used in collaboration with Techcalc.[9] The tools are being used by different engineers working in different domains with different metamodels, addressing requirements (**Req1**) and (**Req2**) mentioned in Section 2. Figure 9 demonstrates an environment where both tools (TechCalc on the left and SolidWorks on the right) are used for collaborative work. More specifically, in this scenario, changes performed in one tool, i.e., Solidworks, are propagated to Techcalc (arrow number 1) when the engineer pushes them (asynchronous). This propagation leads to Techcalc receiving as input the changes from the Solidworks model and analyzing them to pull values on Techcalc's model side (arrow number 2). In the same scenario, it is also possible to propagate changes in the opposite direction, e.g., from Techcalc to Solidworks (arrow number 3). By allowing asynchronous propagation of changes both for pushes and pulls, our infrastructure can give engineers the flexibility of deciding when their changes should be shared with other tools.

Collaboration in multi-domain/user/tool with two different metamodels: Figure 10 demonstrates another multi-domain, multi-tool collaborative use case (for demonstrative purposes). In this scenario, our infrastructure was applied in an architecture project, allowing collaborative work between three tools: Excel for financing management, Visio for sketching, and SolidWorks for 3D representation. To understand the information exchanged

[7] All our use cases demo videos are available at: https://isse.jku.at/designspace/index.php/Demos

[8] Industrial partner: Linz Center of Mechatronics GmbH, Austria. https://www.lcm.at/en/

[9] Techcalc is a tool used in mechatronics. Available at: https://www.lcm.at/en/project/softwaretool-techcalc/

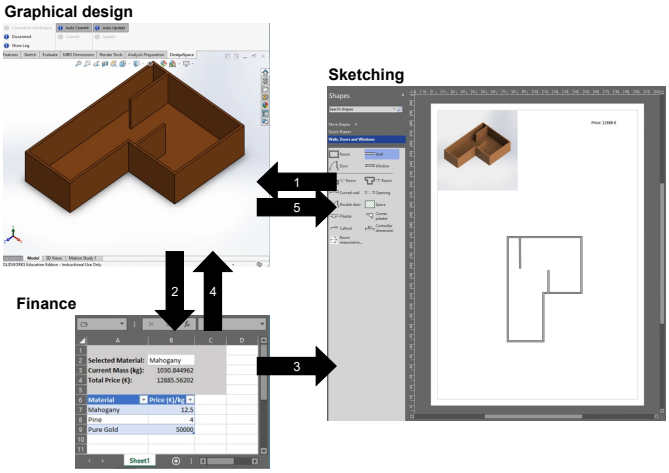


Figure 10 Collaboration using SolidWorks, Visio and Excel.

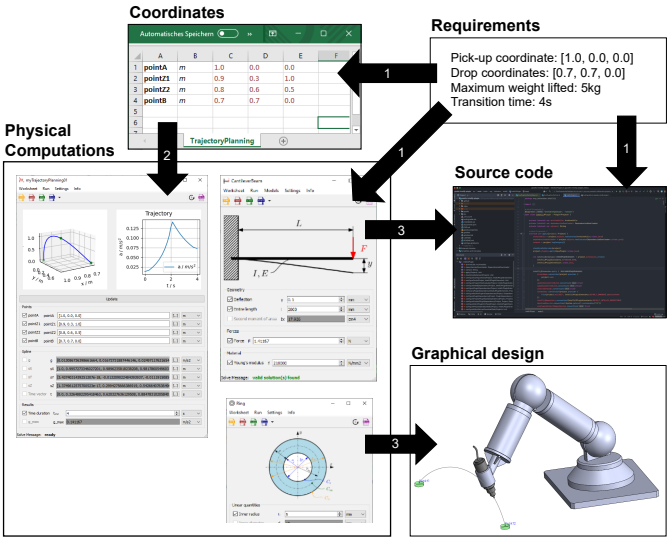


Figure 11 Collaboration using a Requirements tool, Techcalc, SolidWorks, Excel and IntelliJ.

between the tools, we consider a scenario where a house is being designed in the following way: I) a first sketch of the house is created in Visio within its metamodel (right side of Figure 10) which affects the graphical design in SolidWorks (arrow number 1); II) based on the initial sketch, SolidWorks plugin reacts on the Visio model changes and creates a 3D representation of the house. This leads to a pull in Excel (arrow number 2); III) the person responsible for the finances, applies a change in the Excel model, e.g., pulls the price of the material used to construct the walls of the house or the material itself. This change impacts both the Visio sketch (arrow number 3) and the 3D representation created in SolidWorks (arrow number 4). IV) the last change is the 3D rendering calculated by SolidWorks, the image is propagated to Visio, giving real-time feedback on the sketch including the color of the selected material on Excel.

Similar to the first case study, in this scenario our infrastructure can be used to propagate changes from/to tools of multi-domains (**Req1**) while connecting the work of multiple users (**Req2**). Moreover, in this example, all tools were working synchronously as pulls were being propagated automatically (**Req3**). We argue that supporting different preferences of how to propagate changes is important as, in some scenarios, engineers may want the flexibility to decide their preferred way of collaboration (Techcalc and SolidWorks scenario). Whereas, in other scenarios (SolidWorks, Visio, and Excel) synchronization between tools is mandatory to keep the artifacts consistent.

Collaboration in multi-domain/user/tool with various metamodels: Figure 11 depicts another case study (extracted from (Raţiu et al. 2022)), with our industrial partner, where five tools, namely a Requirements tool (implemented as a multi-user tool), Techcalc, SolidWorks, Excel, and IntelliJ, are used in a robot arm project. In this configuration, the environment is hybrid, considering how pushes and pulls are propagated. Firstly, every time a requirement is created or modified, changes are propagated to Excel, Techcalc, and IntelliJ synchronously (arrow number 1). This is necessary because changes in the requirements must be immediately informed to other engineers. The engineer responsible for Excel pulls the metrics for the robot arm, which is also propagated to Techcalc synchronously (arrow number 2). The engineer working with Techcalc, however, will only propagate changes once several changes are manually performed to pull the drawing of the robot arm. Thus, changes from Techcalc are propagated manually (asynchronous push). Once these changes are propagated, they will automatically pull the graphical design in SolidWorks (synchronous pulled), represented by arrow number 3 connecting Techcalc and SolidWorks in Figure 11. However, IntelliJ is not listening to changes from Techcalc. This means that the engineer working with source code decided to perform manual pulls when changes are coming from Techcalc (asynchronous pull). In this case study, we have all possible collaboration commands described Section 3.4.1, thus addressing (**Req3**). This case study also deals with multi-domains (**Req1**) and multi-users (**Req2**).

4.3. Complexity and Performance

Computational complexity and performance are important for collaboration systems, as they should be able to scale fast and easily to be practically applicable. Thus, it is crucial to avoid or at least minimize any delays when pulling visible changes and pushing local changes. Those delays have a limit of 100 ms under which users still have the feeling that the system reacts instantly (Nielsen 1994). Thus, we apply all operations instantly to the local model, before sending them to the central server. The central server can decide to send correction operations back to the local workspace afterward if necessary. The change request concurrency problem (e.g., two concurrent push calls) can be handled by model timestamps. The not up-to-date request gets denied. Pull and push commands are reattaching branches of the model history tree and notifying the local model of the new n operations that are executed in $O(n)$.

To measure the local model execution time of our infrastructure, we loaded multiple persisted operation sequences. The model caching was executed on an Intel-Core i7-10700 pro-

Table 1 Number of elements, operations and execution time

UML models	#elements	#operations	lt* (ms)
RobotArm	1227	25193	387
ATM	1322	32705	491
WordPad	9193	535784	2904
TaxiSystem	11626	517871	2773
QuickUML	54530	2352647	10433

*loading time

cessor with 2.90 GHz on Windows 11 x64-based. To obtain the results, we loaded different UML models[10] into the infrastructure and represented them as elements. Table 1 shows the number of model elements ranging from 1,227 to 54,530. As shown, the number of operations created ranged from 25,193 to 2,352,647. The execution time is given in milliseconds, ranging from 387 to 10,433 ms.

Regarding our purpose and evaluation, the infrastructure works efficiently and without visible delays. Furthermore, the model history tree is designed to support multi-core scalability. Every private workspace on the server, could work on its own core and mitigate the server bottleneck for incremental change requests if necessary (new operations are just added to the end of the workspace branch). However, too many pulls and pushes (reattaching branches) could lead to a bottleneck on the model history tree, leading to many denied commands. This case never occurred in our scenarios. We have highlighted the loading of operations as the most significant bottleneck in the current prototype of the infrastructure. Thus the server and the client were executed on the local network in the same room. The memory limitations of our prototype prevented the loading of larger models, which indicates the need for technical improvements in terms of memory management that we are actively working on. While the current prototype may have limitations and memory bottlenecks, it is essential to acknowledge that these are natural hurdles encountered in the early stages of development. It does not invalidate the main contribution of the proposed approach.

5. Related Work

Existing solutions for collaborative model-driven engineering are mainly implemented for specific scenarios or environments (Torres et al. 2020). Therefore, extending them with new domains, additional features or tools gets difficult. Furthermore, despite the engineers' preferred collaborative work styles, such systems usually offer only synchronous or asynchronous (mostly lock-based) sharing of produced artifacts or changes. As reported in literature reviews, there is a lack of approaches combining the requirements (**Req1**), (**Req2**), (**Req3**) mentioned in Section 2 together (Franzago et al. 2018; Torres et al. 2020; Sharbaf et al. 2022). Hence, in the following, we present related works classifying them by their main active field. Approaches that come closer to addressing the first three requirements are described next.

Multi-domain collaborative systems: Collaboration for multi-domain is explored in many approaches (Torres et al. 2020; Kanagasabai et al. 2018; Kuryazov & Winter 2015; Sharbaf et al. 2022). Koshima et al. (Koshima & Englebert 2015) present an approach that is applied for conflict detection, reconciliation, and merging while collaboratively editing EMF models. Debreceni et al. (Debreceni et al. 2018) present a collaboration framework dealing with the adoption of secure views for working in collaborative modeling. Such views use rule-based access control on models. The authors address the protection of intellectual property across heterogeneous teams as one of their main concerns. Langlois et al. (Langlois et al. 2014) present their approach Kitalpha. They propose to give engineers the possibility of focusing on the system architecture for Model-Based Engineering. This involves the consideration of heterogeneous model artifacts of different domains. These approaches allow engineers to model artifacts from both native or custom tools that may be integrated. They also give support to conflict handling. However, the flexibility with synchronous and asynchronous collaborative work styles is still not explored, limiting the flexibility regarding engineers' collaborative work style preferences. We argue that the use of co-existing meta-models for multi-users with arbitrary pulling and pushing is still missing from current approaches.

Collaborative flexibility in engineering environments: Collaborative engineering environments that allow flexibility regarding users' preferences are related to our study (Franzago et al. 2018). Basic approaches like Git (Loeliger & McCullough 2012) are missing the collaborative work style flexibility and can only handle text-based artifacts. Pietron et al. (Pietron et al. 2021) acknowledged this problem and proposed a collaborative engineering system for graphical modeling tools with synchronous and asynchronous offline collaboration modes. Their approach, however, is designed for graphical modeling tools only, limiting its applicability in practice. Yohannis et al. (Yohannis 2020) proposed an approach that combines state-based and operation-based model transformations to deal with the changes. Their approach, however, only gives support to asynchronous collaboration as it requires the use of version-control systems (VCS). The editor variEd (Kuiter et al. 2021) also deals with synchronous and asynchronous pushes and pulls. VariEd, however, was designed to support collaborative and real-time feature modeling, limiting their work to this domain. Schneider (Schneider 2007) also delivers the same support as Pietron et al. (Pietron et al. 2021), but it is not capable of branching versions and merging. Debreceni et al. (Debreceni et al. 2017) also presents a framework for collaboration called MONDO that provides a web-based modeling front-end and asynchronous offline collaboration. MONDO is focused on secure control access and is based on Subversion. It is not lock-free and does not provide flexible switching between synchronous and asynchronous work. With our infrastructure, however, we define collaborative work styles as different timings

[10] The UML models used are available in an online repository (Herac et al. 2022)

for pushes and pulls, neglecting the possibility of offline work as a mandatory requirement for collaborative engineering from the user perspective.

Operation-based collaborative systems: Artifacts are often described as models and serialized into textual files. A simple way to allow collaboration among those files is to use a VCS. However, understanding the changes and conflicts in such a system could get too complex for human users as the models can get illegibly large and hard to compare. EMF compare (Brun & Pierantonio 2008) is a popular tool that elevates those conflicts to a graphical level where users can understand and work with them. It is a state-based approach that instantiates each model version and compares the entire models for conflicts, which can be very time intensive. Yohannis et al. (Yohannis 2020), however, showed that an operation-based approach is faster for version comparison. There are also other operation-based collaborative systems, like CoObra (Schneider 2007) that supports synchronous and asynchronous collaboration modes but misses conflict handling. The Kotelett approach (Appeldorn et al. 2018) makes use of a Difference Language that is used for representing deltas between two different versions of a model. Their approach gives support to synchronous and asynchronous collaboration, however, it also does not support conflict handling.

Summary: Nevertheless, Franzago et al. (Franzago et al. 2018) and Sharbaf et al. (Sharbaf et al. 2022) show that no work has solved all our requirements together while focusing on a more versatile collaborative infrastructure with multi-domains where the engineers can work with their preferred collaborative work style. We conclude that a lock-free operation-based infrastructure for collaborative MDE composed of support for conflict detection, arbitrary pulling and pushing among concurrent users, and co-existing meta-models of various domains is still missing. Our infrastructure provides all those features together, explained in the following section.

6. Conclusion

In this work, we presented an infrastructure that aims to support model-driven collaborative engineering for multi-domains while being lock-free and allowing users to flexibly decide their collaboration style. The infrastructure also allows engineers to work concurrently with any desired adapted tool. This is achieved by designing the infrastructure to be operation-based. The created change operations are centrally managed in an incrementally growing tree-like structure which enables us to merge and conflict detection of changes. The main benefits of our infrastructure include flexibility when dealing with multiple domains (e.g., co-existing meta-models), arbitrary versioning for multi-users, and the ability to flexibly trigger pulls and pushes to cope with the engineers' preferred collaborative work styles. The combination of those benefits in one infrastructure is our main contribution.

Furthermore, we demonstrated the applicability of our infrastructure with four practical scenarios, including two industrial cases. The collaboration environment and tool variety in these practical scenarios aids us to show our proposed capabilities regarding its support for multi-domain and multi-users while adopting different collaborative work styles based on the engineers/company preferences. Thus complying with all three requirements mentioned in Section 2.

Nevertheless, it is also important to mention that in our current infrastructure, we do not yet support the co-evolution of models like other tools (Di Ruscio et al. 2011; Di Rocco et al. 2018). While conflicts arising from metamodel changes can be resolved, the co-evolution of instances is not automatically handled. As a result, when metamodel changes occur, only new instances created after the changes are guaranteed to be valid with respect to the new metamodel. However, our ongoing research and development efforts focus on implementing services within our infrastructure that will enable the co-evolution of models.

Future work plans also include the extension of our infrastructure with more flexible and intuitive conflict resolution strategies, allowing engineers to select their pushing and pulling models and their overwriting changes in a finer granularity during run-time or to switch to automatic modes. We also plan to apply our infrastructure in additional industrial scenarios, deriving further case studies to evaluate our infrastructure for all requirements together. Furthermore, fine granular access control and grouping mechanisms are still in research.

Acknowledgement

This work is funded the Austrian Science Fund (FWF), grand no. P31989; and supported by the FFG-COMET-K1 Center "Pro^{2}Future" (Products and Production Systems of the Future), Contract No. 881844.

References

Altmanninger, K., Seidl, M., & Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems*, *5*(3), 271–304.

Appeldorn, M., Kuryazov, D., & Winter, A. (2018). Delta-driven collaborative modeling. In *Models workshops* (pp. 293–302).

Berlage, T., & Genau, A. (1993). A framework for shared applications with a replicated architecture. In *Proceedings of the 6th annual acm symposium on user interface software and technology* (pp. 249–257).

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34.

Bucchiarone, A., Ciccozzi, F., Lambers, L., Pierantonio, A., Tichy, M., Tisi, M., … Zaytsev, V. (2021). What is the future of modeling? *IEEE software*, *38*(2), 119–127.

Cicchetti, A., Ciccozzi, F., & Carlson, J. (2016). Software evolution management: Industrial practices. In *10th workshop on models and evolution co-located with 19th international conference on model driven engineering languages and systems (models)* (pp. 8–13).

David, I., Aslam, K., Faridmoayer, S., Malavolta, I., Syriani, E., & Lago, P. (2021). Collaborative model-driven software engineering: A systematic update. In *2021 acm/ieee*

24th international conference on model driven engineering languages and systems (models) (p. 273-284). doi: 10.1109/MODELS50736.2021.00035

Debreceni, C., Bergmann, G., Búr, M., Ráth, I., & Varró, D. (2017). The mondo collaboration framework: secure collaborative modeling over existing version control systems. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 984–988).

Debreceni, C., Bergmann, G., Ráth, I., & Varró, D. (2018). Secure views for collaborative modeling. *IEEE Software*, *35*(6), 32–38.

Di Rocco, J., Di Ruscio, D., Narayanankutty, H., & Pierantonio, A. (2018). Resilience in sirius editors: Understanding the impact of metamodel changes. In *Models (workshops)* (pp. 620–630).

Di Ruscio, D., Lämmel, R., & Pierantonio, A. (2011). Automated co-evolution of gmf editor models. In B. Malloy, S. Staab, & M. van den Brand (Eds.), *Software language engineering* (pp. 143–162). Berlin, Heidelberg: Springer Berlin Heidelberg.

Egyed, A., Zeman, K., Hehenberger, P., & Demuth, A. (2018). Maintaining consistency across engineering artifacts. *IEEE Computer*, *51*(2), 28-35. doi: 10.1109/MC.2018.1451666

Franzago, M., Di Ruscio, D., Malavolta, I., & Muccini, H. (2017). Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, *44*(12), 1146–1175.

Franzago, M., Ruscio, D. D., Malavolta, I., & Muccini, H. (2018). Collaborative model-driven software engineering: A classification framework and a research map. *IEEE Transactions on Software Engineering*, *44*(12), 1146-1175. doi: 10.1109/TSE.2017.2755039

Herac, E., Assunção, W. K. G., Marchezan, L., Egyed, A., & Haas, R. (2022, October). *A flexible operation-based infrastructure for collaborative model-driven engineering - Evaluation Data.* Zenodo. Retrieved from https://doi.org/10.5281/zenodo.7198117 doi: 10.5281/zenodo.7198117

Jongeling, R., Carlson, J., & Cicchetti, A. (2019). Impediments to introducing continuous integration for model-based development in industry. In *45th euromicro conference on software engineering and advanced applications (seaa)* (pp. 434–441).

Jongeling, R., Ciccozzi, F., Carlson, J., & Cicchetti, A. (2022). Consistency management in industrial continuous model-based development settings: a reality check. *Software and Systems Modeling*, *21*(4), 1511–1530.

Kanagasabai, N., Alam, O., & Kienzle, J. (2018). Towards online collaborative multi-view modelling. In *International conference on system analysis and modeling* (pp. 202–218).

Keith, M., & Schnicariol, M. (2009). Object-relational mapping. In *Pro jpa 2* (pp. 69–106). Springer.

Koshima, A. A., & Englebert, V. (2015). Collaborative editing of EMF/Ecore meta-models and models: Conflict detection, reconciliation, and merging in DiCoMEF. *Science of Computer Programming*, *113*, 3–28.

Kuiter, E., Krieter, S., Krüger, J., Saake, G., & Leich, T. (2021). varied: an editor for collaborative, real-time feature modeling. *Empirical Software Engineering*, *26*(2), 1–47.

Kuryazov, D., & Winter, A. (2015). Collaborative modeling empowered by modeling deltas. In *3rd international workshop on (document) changes: modeling, detection, storage and visualization* (pp. 1–6).

Langlois, B., Exertier, D., & Zendagui, B. (2014). Development of modelling frameworks and viewpoints with Kitalpha. In *Proceedings of the 14th workshop on domain-specific modeling* (pp. 19–22).

Liebel, G., Marko, N., Tichy, M., Leitner, A., & Hansson, J. (2018). Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, *17*(1), 91–113.

Loeliger, J., & McCullough, M. (2012). *Version control with git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.".

Muccini, H., Bosch, J., & van der Hoek, A. (2018). Collaborative modeling in software engineering. *IEEE Software*, *35*(6), 20–24.

Nielsen, J. (1994). *Usability engineering*. Morgan Kaufmann.

Object Management Group. (2022). *OMG Metaobject facility*. https://www.omg.org/mof/.

Ogunyomi, B., Rose, L. M., & Kolovos, D. S. (2015). Property access traces for source incremental model-to-text transformation. In *European conference on modelling foundations and applications* (pp. 187–202).

Oluwatosin, H. S. (2014). Client-server model. *IOSRJ Comput. Eng*, *16*(1), 2278–8727.

Perry, D. E., Siy, H. P., & Votta, L. G. (2001). Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *10*(3), 308–337.

Pietron, J., Füg, F., & Tichy, M. (2021). An operation-based versioning approach for synchronous and asynchronous collaboration in graphical modeling tools. In *Proc. of the 1st international workshop on foundations and practice of visual modeling, bergen* (pp. 21–25).

Prakash, A. (1999). Group editors. *Computer Supported Cooperative Work, Trends in Software Series*, *7*, 103–133.

Raţiu, C. C., Assunção, W. K. G., Haas, R., & Egyed, A. (2022). Reactive links across multi-domain engineering models. In *25th international conference on model driven engineering languages and systems* (pp. 76–86). ACM. doi: 10.1145/3550355.3552446

Ráth, I., Hegedüs, Á., & Varró, D. (2012). Derived features for emf by integrating advanced model queries. In *European conference on modelling foundations and applications* (pp. 102–117).

Schneider, C. (2007). *Coobra: Eine plattform zur verteilung und replikation komplexer objektstrukturen mit optimistischen sperrkonzepten* (Unpublished doctoral dissertation).

Sharbaf, M., Zamani, B., & Sunyé, G. (2022). Conflict management techniques for model merging: a systematic mapping review. *Software and Systems Modeling*, 1–49.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: eclipse modeling framework*. Pearson Education.

Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., & Cai, W. (2006).

Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *13*(4), 531–582.

Torres, W., van den Brand, M. G. J., & Serebrenik, A. (2020, oct). A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, *20*(3), 897–916. doi: 10.1007/s10270-020-00834-1

Tröls, M. A., Mashkoor, A., & Egyed, A. (2019). Multifaceted consistency checking of collaborative engineering artifacts. In *2019 acm/ieee 22nd international conference on model driven engineering languages and systems companion (models-c)* (pp. 278–287).

Yohannis, A. (2020). *Change-based model differencing and conflict detection* (Unpublished doctoral dissertation). University of York.

Yohannis, A., Kolovos, D., & Polack, F. (2017). Turning models inside out. In *3rd flexible mde workshop at the 20th international conference on model driven engineering languages and systems* (pp. 430–434).

Zissis, D., Lekkas, D., Azariadis, P., Papanikos, P., & Xidias, E. (2017). Collaborative cad/cae as a cloud service. *International Journal of Systems Science: Operations & Logistics*, *4*(4), 339–355.

About the authors

Edvin Herac is a PhD student at the Institute of Software Systems Engineering (ISSE) at the Johannes Kepler University Austria, supervised by Prof. Dr. Alexander Egyed. He received his master degree in Computer Science from the Johannes Kepler University Linz (JKU). His research interests include Model-Driven Software Engineering and Automated Software Engineering. You can contact the author at e.herac@hotmail.com.

Luciano Marchezan is a PhD student at the Institute of Software Systems Engineering (ISSE) at the Johannes Kepler University Austria, supervised by Prof. Dr. Alexander Egyed. He received his master degree in Software Engineering from the Federal University of Pampa (Unipampa - Brazil). His research interests include Model-Driven Software Engineering, Automated Software Engineering, Software Reuse and Empirical Software Engineering. You can contact the author at lucianomarchp@gmail.com.

Wesley Klewerton Guez Assunção is currently a University Assistant at Johannes Kepler University Linz (JKU) - Austria, and Post-Doctoral researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Brazil. Wesley received his M.Sc. in Informatics (2012) and Ph.D. in Computer Science (2017) both from Federal University of Paraná (UFPR) - Brazil. His areas of interest are Software Modernization, Variability Management, Collaborative Engineering of Complex Systems, Software Testing, and Search Based Software Engineering. You can contact the author at wesleyklewerton@gmail.com or visit https://wesleyklewerton.github.io/.

Rainer Haas is a senior researcher for Machine Design and Hydraulic Drives at the Linz Center of Mechatronics GmbH company. You can contact the author at rainer.haas@lcm.at.

Alexander Egyed is Professor for Software-Intensive Systems at the Johannes Kepler University, Austria. He received his Doctorate from the University of Southern California, USA and worked in industry for many years. He is most recognized for his work on software and systems design – particularly on variability,consistency, and traceability. You can contact the author at alexander.egyed@jku.at or visit http://www.alexander-egyed.com/.