BUMBLE Deliverable D3.2 (Version 3)

Architecture Description for BUMBLE MPS Platform

Edited by: BUMBLE Team
Date: April 2023

Project: BUMBLE - Blended Modeling for Enhanced Software and Systems Engineering

Executive summary

This document describes the MPS-based architecture platforms for the BUMBLE technologies in the various BUMBLE use cases that rely on exploiting Jetbrains MPS as core DSML technology.

First, some of the basic principles and existing features of the Jetbrains MPS technology are presented to enable evaluating to what degree BUMBLE has to extend this base technology in order to satisfy the various requirements identified in Deliverable D2.2.

Second, an architectural view on the identified required extensions is described taking the context of the different use cases into account. These use cases are a subset of all the use cases described in BUMBLE deliverable D2.1: UC4, UC7, and UC12.

The specific, high-level MPS-based architecture platforms for the BUMBLE technologies in these BUMBLE use cases have been described by providing architectural views on the identified required extensions is described, taking the contexts of the different use cases specifically into account.

Contents

Acronyms

AST	Abstract Syntax Tree
B	Blended Syntaxes & Modelling
BCx.y	BUMBLE Common Core Requirement x.y
BTx.y	BUMBLE Common Technical Requirement x.y
C	Collaborative Modelling
Cx.y	(Use Case) Core Requirement x.y
D	Deliverable
E	Evolution
EMF	Eclipse Modelling Framework
GLSP	Graphical Language Server Protocol
GUI	Graphical User Interface
JVM	Java Virtual Machine
LSP	Language Server Protocol
ME	Modelling Environment
N	Model Non-Conformance
T	Traceability
Tx.y	(Use Case) Technical Requirement x.y
DSML	Domain-Specific Modelling Language
IDE	Integrated Development Environment
MPS	Meta-Programming System
MVC	Model-View-Controller
UC	Use Case
VCS	Version Control System

Versions

RELEASE	DATE	REASON OF CHANGE	STATUS	DISTRIBUTION
V1	17/11/2021	FIRST RELEASE OF D3.2	FINAL	UPLOADED TO ITEA PORTAL
V2	23/9/2022	SECOND RELEASE OF D3.2	FINAL	UPLOADED TO ITEA PORTAL
V3	3/4/2023	THIRD RELEASE OF D3.2	FINAL	UPLOADED TO ITEA PORTAL

1 Introduction

This document describes the MPS-based architecture platforms for the BUMBLE technologies in the various BUMBLE use cases that rely on exploiting Jetbrains MPS as core DSML technology. Chapter 2 describes (some of) the basic principles and existing features of the Jetbrains MPS technology to enable evaluating to what degree BUMBLE has to extend this base technology in order to satisfy the various requirements identified in Deliverable D2.2. An architectural view on the identified required extensions is described in Chapter 3 taking the context of the different use cases into account.

Table 1 lists the BUMBLE Use Cases (see BUMBLE deliverable D2.1) that consider Jetbrains MPS as the core DSML technology. It also indicates whether Jetbrains MPS is the only considered core DSML technology. This deliverable focuses on describing the architectures for the use cases that only consider MPS as core DSML technology. The functional concepts of providing blended modeling is covered in Deliverable D3.3, while Deliverable D5.1 focuses specifically on architectures for real-time collaboration across language workbenches where one of these language workbenches is Jetbrains MPS. In this document, Chapter 3 focuses on MPS-specific, implementation architecture-related considerations for UC4, UC7 and UC12.

Table 1. BUMBLE Use Cases that exploit Jetbrains MPS.

Use Case	Description	Lead Partner	MPS	EMF
UC1	Software Open-Source Blended Modeling	MDU	X	X
UC4	Cross-Disciplinary Coupling of Models	Canon	X	
UC5	Reactive and Incremental Transformations across DSMLs	MVG	X	X
UC7	Multi- and Cross-Disciplinary Modeling Workbench	Sioux	X	
UC12	Agile V-model System Architecture	Pictor	X	

Version 3 of this deliverable includes a description of a collaboration solution without central repository as offered by DClare technology in section 3.1.3, and a high level description of the technology demonstrator as developed by the Dutch partners in BUMBLE in chapter 4.

The work and solutions described in this deliverable mainly contribute to the following BUMBLE Technology Bricks and requirements:

Technology bricks	Description of main contributions	Main requirements
Collaboration engine	Real-time as well as non-real-time collaboration is achieved by several, partly complementary solutions	BC5, BC6, BT14, BT16
Blended model access	External access control is integrated with MPS through different user touchpoints	BC10, BC11, BC12, BC13, BT25

2 Jetbrains MPS as Base DSML Technology

MPS (Meta Programming System) is an open-source language workbench developed by JetBrains over the past 15+ years. It is exploited in academia and industry to implement domain-specific languages for real-world applications [1]. A distinguishing feature compared to other language workbenches is the concept of *projectional editing*. It supports almost unlimited language extension and composition possibilities [2] as well as a flexible mix of a wide range of textual, tabular, mathematical, and graphical notations [3]. Hence, it forms a suitable core DSML technology for the BUMBLE project.

2.1 Architecture Overview

MPS relies on (a variant of) the fundamental principle of a Model-View-Controller (MVC) architecture [4], which was first exploited in Smalltalk in the late 1970s [5] and recognized as a generic programming paradigm in the late 1980s [6]. The MVC paradigm allows for one or more views on any specific *model* element. While views can simply be representing a model element to a user in a certain non-editable form using any kind of syntax, such a view can also allow modification (in which case the view is usually called an *editor*). The crux of the MVC paradigm is that all views are automatically updated when the content of the represented model element changes, independent on which editor the modification was initiated from. This is generally realized by using some form of a publish-subscribe mechanism implemented in the Controller aspect of the MVC paradigm. Since there are many different ways to concretely implement such a publish-subscribe mechanism, many concrete MVC architecture variants exist. Abstracting from the Controller aspect, Figure 1 highlights the basic idea of how the MVC paradigm works. Note that, in general, the view from which a modification is initiated is also subscribed to published updates, where the Controller aspect prevents infinite looping (i.e., a request to change the value of the model element to the value it already has is simply ignored).

It is easy to understand from Figure 1 that different views may or may not use different syntaxes to represent a model element. Moreover, different views may in principle serve multiple users at the same time instead of just a single user (as is traditionally the case when exploiting the Integrated Development Environment (IDE) of MPS). Moreover, the MVC paradigm allows to physically distribute where the model and views (and controller) reside. Hence, it matches well with a client-server approach, where the model resides at the server side and the views at the (multiple) client side(s).

Nowadays, many programming languages and IDEs support the development of user interfaces based on (variants of) the MVC paradigm. In particular, web applications exploit the MVC paradigm given the nature of potentially having multiple browser instances (views) open (by a single or multiple users) on the same content (model). Think of Google Docs as an example where multiple users can view and edit the same content. Most frameworks for web applications adopt the MVC paradigm. A key ingredient of frameworks supporting the MVC paradigm are the facilities provided for the Controller aspect. This means that such frameworks generally minimize the amount of code that needs to be written to realize the Controller aspect (think for example also of undo-facilities). In the context of MPS, this allows DSML developers to focus on creating the DSML model and views.
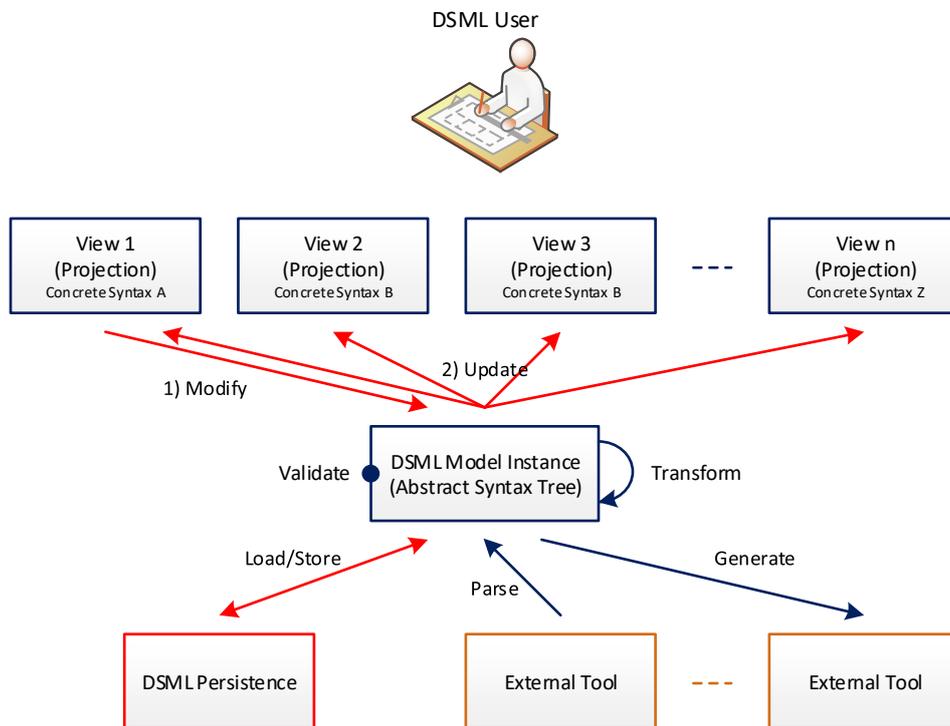
Figure 1. High-level impression of the Model-View(-Controller) paradigm underlying MPS. Red items are provided as part of MPS while a DSML developer may require creation of relevant specifications for the blue items depending on needs for a DSML context. Synchronization between views is exemplified with the modify - update sequence, where update of the view from which the modification originates does not imply further change. Depending on needs for a DSML context, transformations can be initiated explicitly or implicitly (e.g., via modify actions) by the DSML user from any view.

In MPS, views are called *projections*, which explains the term projectional editing. Exploitation of projectional editing in the context of DSMLs also implies a different user experience compared to approaches that primarily rely on a parser-based approach as in many other language workbenches. While projectional editing relies on modifying an Abstract Syntax Tree (AST) via projections (see Figure 1), parser-based approaches rely on (re-)constructing an underlying AST based on (re-)parsing a modified concrete syntax. To exemplify this, copy & paste actions from external tools (such as an ordinary text editor) into MPS do not work by default (as a DSML user might expect). A DSML developer can provide a means to support such actions, but this requires extra development effort compared to language workbenches relying on a parser-based approach where such copy & paste action matches naturally. The atypical user experience explains the steep learning curve often expressed when adopting MPS.

BUMBLE relies on the possibility of combining the MVC paradigm as implemented in MPS with the idea of having multiple concurrent users (instead of just one), possibly accessing the very same model from a different environment (such as a web-based front-end), while exploiting any of the available concrete syntaxes. Based on the requirements reported in D2.2, not all use cases require the full capabilities of this, and the different use cases do have somewhat different requirements to realize this combination. Hence, this deliverable describes specific architectural considerations for each use case.

2.2 Existing Support with respect to BUMBLE Features

This section evaluates to what extent MPS as core DSML technology satisfies the requirements identified in D2.2 and where extensions are needed by reviewing each of the key BUMBLE features. Table 2 gives a high-level summary of the evaluation results as described in this section.

Table 2. Summary of BUMBLE features supported by MPS out-of-the-box.

BUMBLE Feature	Out-of-the-Box Support in MPS
Blended Syntaxes & Modelling (B)	Yes, with textual, tabular, symbolic, and basic graphical concrete syntaxes. Further concrete syntaxes are supported with plugins.
Collaborative Modelling (C)	Yes for off-line collaboration supporting various version control systems (extendable with plugins). **Not for real-time collaboration**
Evolution (E)	Yes for version control of DSML definitions and instances. Yes for co-evolution of DSML instances when their DSML definition changes
Traceability (T)	Yes with respect to version control and in terms of hyper-link like cross referencing between multiple projections of a model element
Model Non-Conformance (N)	Projectional editing automatically disables/prevents non-conformance. **Non-conformance may however prevail for interactions with external tools, in which case the DSML user has to resolve issues manually**.

2.2.1 Blended Syntaxes & Modelling (B)

As discussed in Section 2.1, MPS supports multiple concrete syntaxes out-of-the-box. A DSML model instance resembles an Abstract Syntax Tree (AST) that conforms to the definition of the DSML model. This AST is modifiable by any projection that may exploit different concrete syntaxes. MPS supports textual, tabular, and symbolic (mathematical) and basic graphical syntaxes such as tree-alike and box-edge alike diagrams out-of-the-box. This can be extended by using plugins. An example of such a plugin is the Diagrams plugin [7] by Itemis, which allows for the creation of more sophisticated graphical editors. Since editors are defined on individual model elements, their composition conform the AST allows mixing syntaxes in the composite projections that are eventually shown to DSML users. The combination of these facilities exactly realizes the BUMBLE feature of blended syntaxes & modelling, thereby satisfying core requirements BC1 and BC3 and the semantic aspect of core requirement BC4 identified in Deliverable D2.2. An example from UC4 is shown in Figure 2.

By default, MPS provides a reflective editor for a DSML definition, which will represent an AST conforming to that DSML definition in a simple textual form. A basic editor with a slightly more compact syntax can be generated automatically as well. When multiple editors are defined, the DSML developer can specify which one to use by default. DSML users can always select a different concrete syntax if desired. It is even possible to display multiple projections at the same time (where synchronization between changes to the AST and all relevant projections becomes visible). With the ability to define the composition of editors on model elements separate from the composition of model elements according to the AST, it is possible to not represent model elements in a composite

projection. Having these facilities in MPS out-of-the-box completes satisfying core requirements BC2 and BC4 in D2.2.

Figure 2. Example of blended syntaxes & modelling in MPS (From UC4).

By fully satisfying core requirements BC1, BC2, BC3 and BC4 identified in Deliverable D2.2, the existing facilities provided by MPS do not require extension in the BUMBLE project. The use cases can simply exploit these out-of-the-box facilities to realize their specific requirements. This includes the development and use of plugins when further concrete syntaxes are needed to realize a use case.

Technical requirements BT1 - BT10 in Deliverable D2.2 are also all satisfied out-of-the box by MPS. With respect to BT7, cross-referencing between model elements is always at the level of the AST and not at the level of any concrete syntax. Concerning BT9 and BT10, it is noted that validation occurs also at the level of the AST and not at the level of any concrete syntax, see also Figure 1. Notifications on validation errors are displayed in any projection that does not hide the relevant model elements.

2.2.2 Collaborative Modelling (C)

The BUMBLE feature of collaborative modelling considers two variants, namely off-line collaboration by exploiting a Version Control System (VCS) and real-time collaboration between multiple users. In this section, we focus on real-time collaboration. Off-line collaboration is considered in Section 2.2.3.

The IDE that comes with MPS is a desktop application intended to serve a single user at any time. In other words, real-time collaboration between multiple users is not supported out-of-the-box. Hence, core requirement BC5 and technical requirements BT14 and BT16 are not satisfied without

extension by the BUMBLE project. Extensions for real-time collaboration should also consider the requirements on access control in Deliverable D2.2. As discussed in Section 2.1, MPS is however amenable to extending towards supporting real-time collaboration without changes to the fundamental MVC paradigm, which (to some extent) satisfies core requirement BC6 in Deliverable D2.2. Section 3.1 evaluates possible existing alternative approaches that could be used to extend MPS with real-time collaboration as part of the BUMBLE activities initiated for the various use cases requiring this.

2.2.3 Evolution (E)

BUMBLE considers two aspects of evolution: version control of DSML instances and DSML definitions, where the evolution of the latter may imply breaking conformance of the former to the updated DSML definition. We first discuss relevant aspects of version control.

The IDE for MPS integrates version control out-of-the-box. Supported VCSs include GIT, SVN, mercurial and perforce. Further VCSs can be supported by means of plugins. By relying on traditional VCSs, MPS provides traditional version control facilities including access control, traceability, branching, diffing, and merging. A key feature of MPS is that differencing and merging facilities are provided at the level of the AST and all accompanying editors, validation rules and other typical features of DSML technologies. Differences can be viewed at the level of a concrete syntax of choice (depending on what editors have been defined) as illustrated in Figure 3. This is independent of how a DSML is persisted in the version-controlled files. These facilities satisfy requirements BC7, BC8, BT19, BT20 and BT24 in Deliverable D2.2.
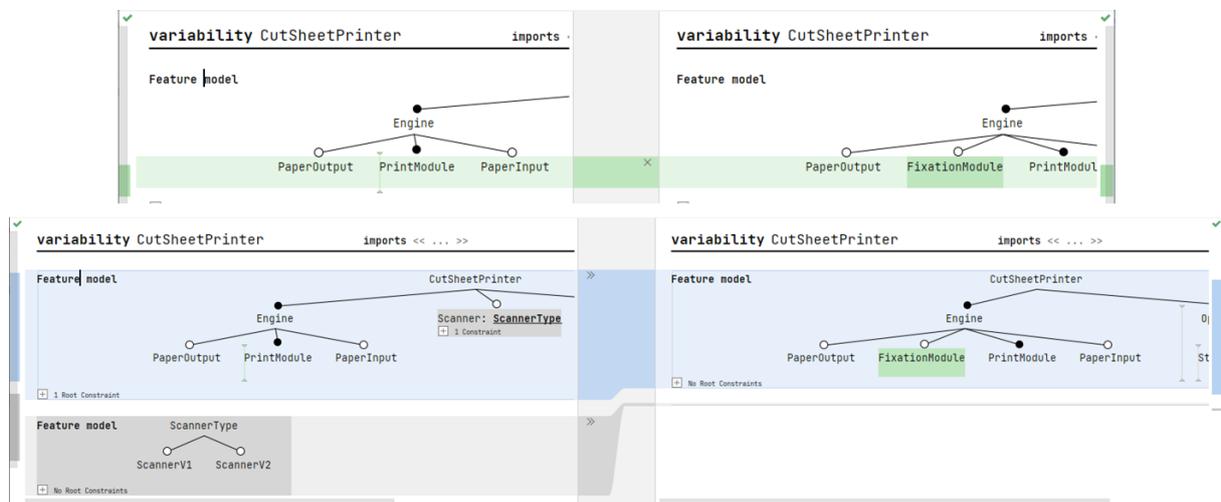
Figure 3. Example of diffing in MPS for a DSML with graphical syntax (From UC4).

Considering evolution of DSML definitions in relation to core requirement BC9 in Deliverable D2.2, the IDE of MPS provides extensive facilities to allow DSML developers to create automated migration of DSML instances that would no longer conform to an updated DSML definition. Although non-conformance may occur, MPS cannot guarantee correct working of relevant DSML facilities in such cases (see also Section 2.2.5) and hence, migration facilities are crucial in MPS. MPS provides migration facilities to a DSML developer that allows implementing migration processes. This means that core requirement BC9 and technical requirements BT21 are satisfied out-of-the-box. A DSML user opening a DSML instance that has not yet been migrated to conform to the corresponding updated DSML definition, will be notified about this (thereby satisfying technical requirement BT22

in Deliverable D2.2). In case of a migration need, DSML users can inspect the migration effect to understand the implied differences, which satisfies technical requirement BT23 in Deliverable D2.2. In some situations, automated migration may not be fully successful in which case the DSML user is informed and must take (manual) action to resolve remaining issues. MPS provides a means to inspect such remaining issues, but in case such a situation occurs, it reflects improper development / testing of the migration process by the DSML developer. In practice, the DSML user might also be able to request the DSML developer for assistance in case such migration issues would occur (not via MPS but via alternative support facilities that should be provided).

Since MPS satisfies all core and technical requirements regarding evolution described in Deliverable D2.2, no extensions are needed by the BUMBLE project. It is however expected that the combination of described facilities for evolution with the need for real-time collaboration in certain use cases may require extensions for which no concrete requirements have been identified yet. These may arise in subsequent versions of Deliverable D2.2 and if so, they will be considered in the next version of this deliverable.

2.2.4 Traceability (T)

Different forms of traceability exist. Deliverable D2.2 does not specify requirements that are specific to traceability alone. Traceability-requirements are always defined in the context of other requirements.

Traceability in the context of version control is supported out-of-the-box in MPS by relying on facilities of traditional VCSs together with the ability to inspect change histories. This means that DSML users and DSML developers can trace what modifications were made when and by whom. Hence, all traceability-related requirements identified in D2.2, except for the context of real-time collaboration, are satisfied by MPS out-of-the-box.

Another form of traceability is the ability to trace from projections in which a model element is used to the source where that specific model element is defined. Because such a model element is not necessarily editable in all projections, it is desirable to be able to trace to its definition. MPS provides hyperlink-like tracing within and across DSLs during model creation, see also Figure 4. In addition, MPS provides a means to execute all kinds of search queries to find model elements.
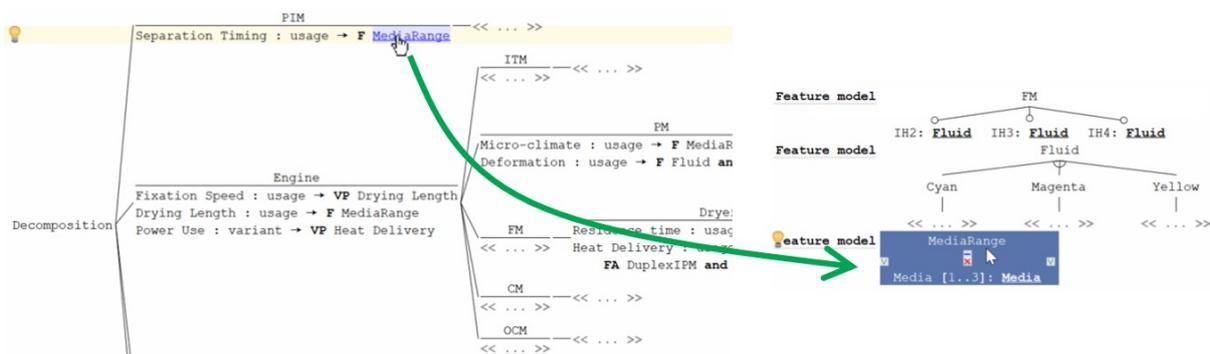

Figure 4. Example of traceability between DSL elements in different projections (From UC4).

2.2.5 Model Non-Conformance (N)

Deliverable D2.2 does not specify common requirements on support for model non-conformance. Nevertheless, it is good to clarify that MPS basically disallows and disables non-conformance of any concrete syntax for an AST due to the nature of projectional editing. This may sound like a limitation, but it is actually an advantage. The main advantage is that all inputs to the model are syntactically correct by definition. In case a need arises for capturing information in a DSML that is not supported by its DSML definition, it means that the DSML definition must be extended. Given testability and continuous integration capabilities of MPS DSMLs, the lead time of extension is very short in practice, which makes this way of working quite feasible in many practical contexts.

In MPS, model non-conformance only plays a role in the context of interactions with external tools. For example, a copy & paste action from an external tool into the language workbench may result in inconsistencies with the DSML definition which subsequently have to be resolved by the DSML user. It is possible to define parsers for this scenario in MPS and this may result in errors that will have to be resolved. With respect to importing from external tools, see Figure 1, MPS provides the ability to exploit the base language (a Java variant) to create parsers where responsibility to ensure model conformance is put on the DSML developer creating the parser. There is no out-of-the-box facility to generate a correct-by-construction parser, for example based on a grammar specification (like the approach of XText). It is however possible to create such a facility as a plugin, e.g., by using ANTLR-grammar based parsing. This could also serve the sketched copy & paste scenario. Individual use cases do require interaction with other tools by parsing files and hence such a facility could be useful. There is however no common requirement identified for this in deliverable D2.2.

As described above, model non-conformance can actually occur in MPS in the context of interactions with external tools. We briefly describe some examples of possible causes for non-conformance that may occur. Consider the flexibility one has for values of identifiers to distinguish items of the same kind (think of giving names to classes in an object-oriented programming language). The use of equally valued identifiers for different items of the same kind poses a problem for parsers as they cannot distinguish between these items when they are referred to by other items (think of instances of those classes) without help of the user. If such a situation would arise (e.g., class A and class B are listed in sequence and then the order of A and B is swapped around in the program listing), parsers may make an arbitrary choice and/or give an error. Another cause of non-conformance may originate from lexers not being able to determine the boundary of tokens when the value of identifiers includes white-spaces or other characters that are part of the concrete syntax. On the other hand, these problems do not exist in an approach where the persistence of a model is completely independent from the way the model is presented to a user in some view / projection as shown in Figure 1 (one could consider persistence as a special view / projection that is not used to interact with the user). In case resolving potential non-conformance cannot be automated in lexers/parsers when creating or modifying DSML instances in MPS by importing specifications in some concrete context (including the copy & paste scenario), a DSML user will have to resolve the non-conformance issues manually. When considering the other direction, i.e., export of a DSML instance in MPS to an external tool, generators require of course to also resolve the mentioned issues if the external tool does not support the described flexibility (e.g., by generating valid identifier values instead of copying those provided by the user).

3 BUMBLE Extensions to Jetbrains MPS

Based on the evaluation of MPS in Chapter 2 in view of the requirements in Deliverable D2.2, this chapter presents possible solution directions and the concrete architectural and technological choices that were made for the various use cases. We focus on three topics:

- Real-time collaboration (core requirement BC5 in Deliverable D2.2)
- Contextual integration (core requirement BC6 in Deliverable D2.2)
- Aspects of model life-cycle management (core requirements BC10-BC13 in Deliverable D2.2)

We remark that each of these topics is relevant for UC4, UC7 and UC12. The requirements in D2.2 imply however different technology choices for UC4, UC7 and UC12 as elaborated in this chapter.

Although UC12 does not express requirements on real-time collaboration and contextual integration, Deliverable D2.2 lists core requirement C12.7 to need access via a webpage. Web-access is not supported out-of-the-box, but a solution direction could well be to exploit a web-based extension for real-time collaboration. Technical requirements T12.1 and T12.2 in Deliverable D2.2 originate from the initial intention to select Eclipse as core DSML technology, which has however been revised after Deliverable D2.2 was submitted. Support for importing EMF models into MPS does exist, although this may not be sufficient in practice. At the time of writing the second version of this deliverable, UC12 is relying on the standard architecture of MPS.

3.1 Real-Time Collaboration

This section describes architectural considerations to realize core requirement BC5 identified in D2.2 for MPS. Section 3.1.1 gives an overview of relevant existing technologies that could be exploited to realize core requirement BC5. The technology choice for UC4 is elaborated in Section 3.1.2. The technology choice for realizing real-time collaboration in UC7 strongly relates to realizing requirement BC6 (contextual integration) and, hence, the architecture for UC7 is described in Section 3.2.2.

3.1.1 Overview of Existing Technologies for Real-Time Collaboration

As part of the BUMBLE activities, we have performed an investigation of available libraries and frameworks in the context of collaborative software engineering without focusing on specific DSML technology contexts. The technologies have been classified according to the features (see Table 3) related to their technical details (dependencies, network architecture and editor type), functionalities provided (diff/merge, offline support, language support, extensibility, data storage and workspace awareness) and applicability in real-world scenario (license requirements and availability of commercial support).

Table 3. Features selected for the classification of collaboration technologies.

Feature	Explanation	Possible Values
Dependencies	Libraries or other frameworks required for the technology to work	Free text
Editor Type	Type of editor provided by the approach for editing the software artifacts	C(ode-editor), R(ich-text), P(lain-text), W(eb-editor)

Network Architecture	The way in which the technology organizes the network devices and services to connect collaboration parties	C(entralized): there is a single server that contains all the updates made by the clients. P2P: clients communicate directly with each other M(ixed) clients exchange messages / editing operations directly with each other, but there is a central server (e.g., used as common repository).
Diff / Merge	Support for diffing and merging	Y(es), N(o)
Offline Support	Are the changes made offline to an artefact preserved and uploaded later?	Y(es), N(o)
Language Support	If the approach is tailored to support collaboration modeling with some specific language(s) or it provides some mechanism to import/define user-defined languages	Free text
Extensibility	If the used approach can be extended with more functionalities (as plugins)	Y(es), N(o)
Data Storage	Does the technology offer an online storage from where the latest updated document can be retrieved later?	Y(es), N(o)
Workspace Awareness	Awareness support to other users' operations while they are working on the same artifact/project	Y(es), N(o)
License	Under which license is the technology copyrighted?	Free text
Commercial Support	Do commercial parties exist that provide support for the technology?	Y(es), N(o)

Table 4 provides an overview of the evaluated technologies (with a weblink), where an entry '?' denotes that we have not been able to identify the appropriate value for the involved technology. Out of the 22 technologies presented in Table 4, ProseMirror, Mondo, AtoMPM and Flame provide dedicated collaboration over models. To summarize the findings of the table, we observed that most of the technologies provide code or textual editors. This shows collaboration is more prevalent during the development/coding phase of software engineering. Centralized network architecture is more prevalent. Out of 22 technologies, seven provide diff/merge to facilitate the versioning of collaborative software artefacts. It is encouraging that attention has been paid to offline support for collaboration with 8/22 technologies already providing this feature. The ability to customize and extend the existing technologies is required so that collaborating engineers can tailor the existing technologies to their needs or use these libraries/frameworks inside their own tools. Half of the studied technologies (11/22) provide this feature. Further, twelve technologies provide data storage support and ten keep users aware of each other's actions in the workspace.

Table 4. Libraries/Frameworks for collaborative software engineering.

Technology	Dependencies	Editor Type	Network Architecture	Diff / Merge	Offline Support	Language Support	Extensibility	Data Storage	Workspace Awareness	License	Commercial Support
DerbyJS	node.js MongoDB	-	?	N	Y	node.js	Y	Y	N	MIT	N
ShareDB	node.js	-	C	N	Y	node.js	?	Y	Y	MIT	N
Convergence	docker	C,P,R	C	N	Y	JavaScript Scala	Y	Y	Y	MIT	Y
Meteor	node.js	?	C	N	N	JavaScript	Y	Y	N	MIT	Y
Ace Editor	None	C	?	N	N	45 Languages	Y	N	Y	?	Y
Slate	Yarn	R	?	N	N	Language Agnostic	Y	Y	?	MIT	?
socket.io	None	-	C	N	Y	Java, C++, Swift, Dart, Python, .NET, Rust, Golang	?	N	Y	MIT	Y
CollaborativeJS	npm or bower or Yarn	-	C	N	Y	JavaScript	N	?	N	MIT	N
RethinkDB	gcc or clang, protocol buffers, jemalloc, ncurses, boost, python 2, libcurl, libcrypto, libssl-dev	-	Cloud (Own or Internet)	N	N	Ruby, Python, Java, JavaScript, node.js, C# / .NET, Go, PHP	N	Y	N	Apache 2.0	Y
ProseMirror	npm	R	C	Y	N	Language Agnostic	Y	N	?	MIT	Y
Automerge	?	-	Network Agnostic	Y	Y	Language Agnostic	?	Stores Change History	N	MIT	Y
yjs	None	R	P2P	N	Y	Language Agnostic	?	N	N	MIT	Y
Fluid	node.js	-	Distributed	N	Y	JavaScript, TypeScript	N	Y	N	MIT	N
WebEditKit	MPS, MPSServer Plugin	Projectional	?	N	N	Language Agnostic	N	N	?	Apache 2.0	Y

TogetherJs	JS Library, HTML	W	C	N	N	JavaScript	Y	N	Y	Mozilla 2.0	Y
Mondo	None	Graphical	C	Y	Y	Language Agnostic	Y	Y	Y	EPL 2.0	N
Collaboro	JRE 1.6 or newer, Eclipse 3.6 or newer, EMF 3.7 or newer, Eclipse Graphviz	Tabular	C	Y	N	Language Agnostic	Y	Y	Y	EPL 1.0	N
AtoMPM	Python, python-igraph, python-socketio, node.js	Graphical	C	Y	N	Language Agnostic	Y	Y	Y	LGPL 3.0	N
Flame	?	Graphical	C	Y	N	XTEAM	Y	Y	Y	MIT	N
Modelix	MPS, docker, gradle	-	C	N	N	Language Agnostic	N	Y	Y	Apache 2.0	Y
Jetbrains RD	.NET framework or gradle 6.2.2, or git, cmake, Visual Studio 2015+ or clang 6.0+	-	?	N	N	Kotlin, C#	N	N	?	Apache 2.0	Y
(Distributed) DClare	Maven	-	-	N	-	-	N	N	N	LGPL 3.0	Y
Saros	Eclipse	P	P2P	N	N	Java	N	N	Y	GPL 2.0	Y

Considering the BUMBLE use cases, Modelix, Jetbrains/rd and (Distributed) DClare are considered as most promising technologies and WebEditKit appears to be a potential candidate to realize real-time collaboration in case of using MPS as core DSML technology:

- **Modelix** is a real-time collaboration extension to Jetbrains MPS developed by Itemis. It consists of a collection of cooperating docker images and a plugin for the IDE that comes with MPS. It offers DSML users the ability to collaborate on MPS projects in real-time. Modelix is still in relatively early stages of development, but far enough to be installed and experimented with. Modelix has been selected to realize the requirements on real-time collaboration in UC4 (MPS only) and UC1 (EMF and MPS). Section 3.1.2 describes the architecture of Modelix in the context of UC4. UC1 exploits Modelix to realize real-time collaboration across language workbenches, which covers the work on Parsafix and its generalization. An architecture description regarding real-time collaboration in this context is provided in Deliverable D5.1.

- **Jetbrains RD** is a Reactive Distributed communication framework for .NET, Kotlin and C++. Inspired by the JetBrains Rider IDE, the framework contains several libraries for single process usage and cross-process communication. For instance, the library Lifetimes provides concurrency and reactive programming, *RdFramework* is a networking library for reactive distributed communication, *RdFramework.Reflection* is a plugin used for defining models

(using C#) and *RdGen* generates stub classes by Kotlin DSL models. Jetbrains RD has not only been selected to address the requirement of real-time collaboration in UC7 but also specifically to address the requirements regarding contextual integration between MPS with Supermodels as explained in Section 3.2.2.

- **(Distributed) DClare** is being developed by the Modeling Value Group B.V. DClare is a general-purpose declarative language based on Java. The aim of DClare is to unlock multi-processor capabilities and reduce program threading. The DClare concepts have been inspired by Object Oriented Programming, functional programming, Object Oriented Modeling and Spreadsheets. Currently, functionality is being added to DClare to synchronize changing models between multiple modeling-environments across the internet. The DClare-based solution will not contain a central repository. The first integration of this functionality will be part of DclareForMPS, enabling multi-user editing of models in MPS. Support for (bidirectional) transformations between different languages in DClare is also being added. Enhancement of (Distributed) DClare is the core of UC5 (covering both EMF and MPS) and is reported on in Deliverables D3.7, D4.2 and D5.1.

- **WebEditKit** is a typescript framework for creating projectional editors, running in browsers, which interact with JetBrains MPS. The editors are then able to get information from MPSServer and present it, react to changes from MPSServer and communicate changes performed by the user to MPSServer. The ultimate goal of WebEditKit is to make all internal MPS APIs available remotely. It brings the MPS to the web in an effort to better facilitate the integration of MPS with other systems, easier installation, and more usability for users. For WebEditKit, MPS needs to be used together with the MPSServer plugin. WebEditKit will be one of the potential target collaboration technologies to be integrated into the generic collaboration solution whose architecture is discussed in Deliverable D5.1.

Apart from the collaboration technologies discussed above that will be experimented with in the BUMBLE project, the Graphical Language Server Protocol (GLSP) will be used as a communication protocol to realize real-time collaboration between client and server in WP5 (discussed in Deliverable D5.1). GLSP defines a Language Server Protocol (LSP) for diagrams and integrates well with existing tool chain and business logic.

3.1.2 Architecture Description for UC4 (Cross-disciplinary Coupling of Models)

MPS comes with a high-end IDE that is well suited for the development and exploitation of DSMLs in an industrial context. This IDE is a stand-alone single-user desktop application. Its users include DSML developers creating DSML definitions with accompanying functionalities and DSML users that create and exploit instances of such DSML definitions. The appearance of the IDE can be customized to ease usage for DSML users (e.g., as a Rich Client Platform). This helps in reducing the learning curve that is often experienced and in only covering the specific functionality relevant for DSML users.

DSMLs at Canon Production Printing often represent domain-specific interfaces between models from different engineering disciplines. This is also the case for the DSMLs in UC4. Such interfaces would benefit from the ability to collaboratively update DSML instances in a similar fashion as Google Docs and Microsoft Office 365 support for office documents. Real-time collaborative modeling would benefit from a web-based front-end for, in particular, DSML users. A server-based

deployment would also ease version control for DSML users unfamiliar with traditional software technologies such as GIT, SVN, etc., and terminology for version control. In addition, it eases updating any involved tools in case of DSML (co-)evolution. This includes, for example, tools for automated code generation, build, and test tool chains, as well as visualization and underlying analysis engines.

Canon Production Printing has selected Modelix by Itemis as means to realize real-time collaborative blended domain-specific modeling. Modelix exposes the views and editing capabilities of a DSML model that is defined in the IDE that comes with MPS [8] inside a webpage. Figure 5 shows a high-level overview of the combination of the desktop IDE of Jetbrains MPS and Modelix [9]. It relies on the DSML model instance being accessible on a central server. The DSML model instance can be accessed with the existing IDE, using the Modelix plugin for Jetbrains MPS. The DSML model instance can, at the same time, also be accessed via a web-based front-end in a DSML user's web browser. The idea is that execution of DSML facilities such as model transformations and code generation run on the server, while viewing and editing models should be as simple as opening a webpage. This is relevant since engineers of all engineering disciplines are expected to exploit the DSMLs for which tooling is realized using MPS. However, engineers with little or no affinity for software development have many difficulties in adopting the IDE, even as DSML users. The exploitation of Modelix allows for highly customized web-based Graphical User Interfaces (GUI) with a domain-specific look-and-feel that is much closer to that of domain-specific tools currently used.
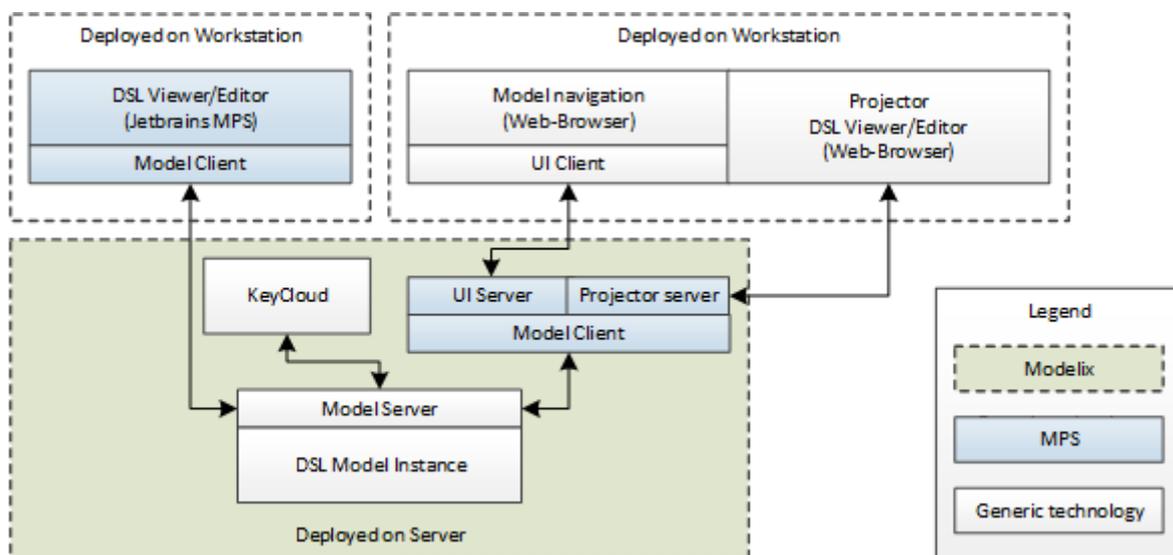

Figure 5. Architecture for UC4 combining MPS and Modelix.

The approach of Modelix shown in Figure 5 has the major benefit that existing DSMLs defined with MPS can be used in a real-time collaborative way without requiring the DSMLs to be redeveloped in a different technology such as JavaScript It uses the Projector technology to create an MPS instance that delegates its GUI to the web-browser. This means that the existing IDE can still be used by DSML users that are already familiar with it as it integrates seamlessly. These considerations led to choosing Modelix to realize core requirement BC5 in Deliverable D2.2 the context of UC4 (i.e., core requirement C4.12).

The user and rights management is delegated to a KeyCloud instance, which keeps track of users from a multitude of sources, including Active Directory. This allows decoupling Modelix from the specific authentication backend that is required to authenticate users.

The DSML Instances are arranged in Workspaces, which are environments that are separated from each other, and can each contain a different set of languages and imported models.

3.1.3 Architecture Description for Collaboration without a Central Repository

Real-time Collaboration can also be achieved without a central repository. Instead, a central exchange service is used to broadcast model changes to all collaborating modeling clients. DclareForMPS realizes this approach for MPS clients. This approach can also be combined with deferred synchronization using version control systems like GIT. This gives flexibility to orchestrate immediate collaboration and separate parallel modeling and merging in any combination you wish for.

The diagram in Figure 6 shows the architecture of this approach if DclareForMPS is used. The diagram is a subset of the diagrammatic notation defined in D3.3. Dclare takes care of incremental and performant synchronisation between different syntaxes (in Figure 6 2 examples are given: on the left between text and table syntaxes, on the right between text and Java syntaxes), between different clients through a real-time DClare sync service (indicated at the bottom), and between different clients through a versioning system (so-called deferred remote synchronization at the top).
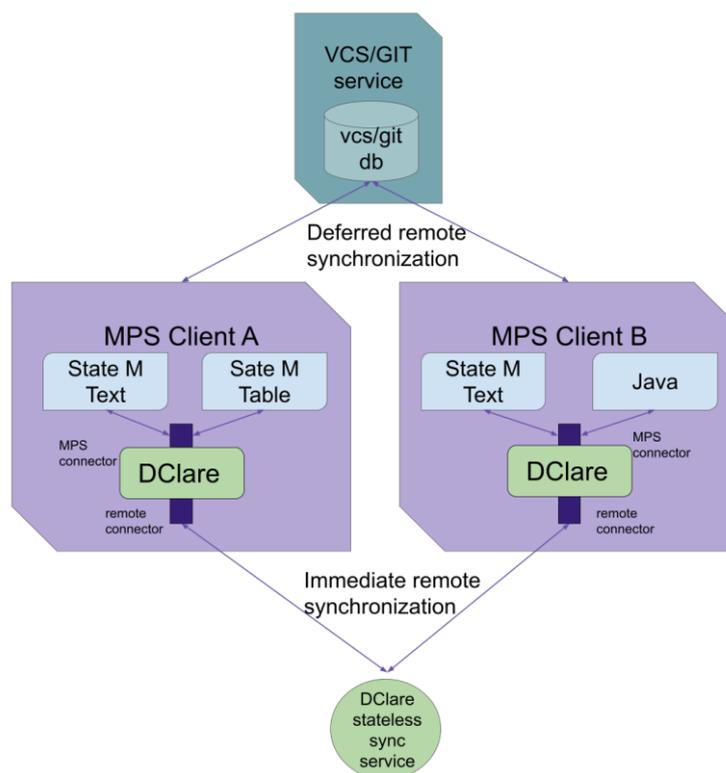

Figure 6. Synchronisation without central repository with DClareForMPS

When deferred and immediate collaboration is combined, all involved transformations need to be incremental. This is to guarantee that parallel changes are not lost while (re)transforming models after synchronization. More information on the dynamic behaviour of the resulting collaboration solution can be found in chapter 4 of deliverable D5.1v2.

3.2 Contextual Integration

It is beneficial to integrate a DSML model with its context, so that changes in the DSML model lead to immediate feedback and updated data in the context. This section describes the architectural considerations for contextual integration in UC4 and UC7 to satisfy core requirement BC6 identified in Deliverable D2.2 (i.e., core requirements C4.23, C4.26 and C7.1).

3.2.1 Architectural Description for UC4 (Cross-disciplinary Coupling of Models)

Contextual integration with external tools is a key ingredient of UC4. However, there are no specific extensions needed to the core DSML technology of MPS when it comes to coupling of models in various external tools. It is considered sufficient to exploit the created import (parser) and export (generator) facilities as expressed in the architectural overview of MPS in Figure 1. On the other hand, Canon Production Printing has various existing development environments that could benefit from exploiting DSML technology. Such environments may rely on custom Graphical User Interfaces (GUI) for which it is often not easy or even infeasible to (completely or partly) replace them in a gradual or disruptive step by the IDE for MPS. Such situations would benefit from the ability to integrate the DSML technologies provided by MPS into the existing development environments as DSML widgets.

Besides the GUIs made available to print professionals (customers of Canon Production Printing) as part of the Professional Digital Printer product families, several tools to develop and maintain these product families are also being created for use within Canon Production Printing. Application of DSML technology to formalize the domain-specific knowledge underlying both such Graphical User Interfaces has major benefits. However, existing DSML technology does not (yet) provide the customization flexibility that traditional software technology provides to develop GUIs. Canon Production Printing envisions the use of Itemis' Modelix to create DSML widgets as part of GUI applications realized with traditional web technology such as Google's Angular [10].

MPS and Angular use a similar component-based approach to compose complex views on related data items from simple views on individual data items. Moreover, the MVC paradigm can be realized fairly easily in Angular. In the Angular context, the controller concept of a MVC paradigm, which is responsible for converting (raw) data into a form that can be displayed to users via a view, is often denoted as ViewModel. Given the similarities, Canon Production Printing envisions that Angular applications could be partly DSLified with Modelix-based components, bringing together the strengths of MPS's DSML technology and the customization flexibility of Angular. This would allow combining the DSML definition with accessible simulation, analysis, and visualization environments. Hence, the chosen combination of in particular the MPS and Modelix technologies in the context of UC4 is expected to also enable satisfying core requirements C4.23 and C4.26 identified in Deliverable D2.2.

3.2.2 Architectural Description for UC7 (Multi- and Cross-Disciplinary Modeling Workbench)

Sioux developed an in-house Modeling Environment (ME) named SuperModels, which provides facilities to create and use graphical DSMLs (with diagrammatic notations) in high fidelity editors. Next to that we use MPS mostly for non-graphical DSMLs that benefit from the MPS facilities described in Section 2.2. We apply both SuperModels and MPS to describe different aspects of a system. Those different aspects concern different engineering disciplines yet they are related to each other.
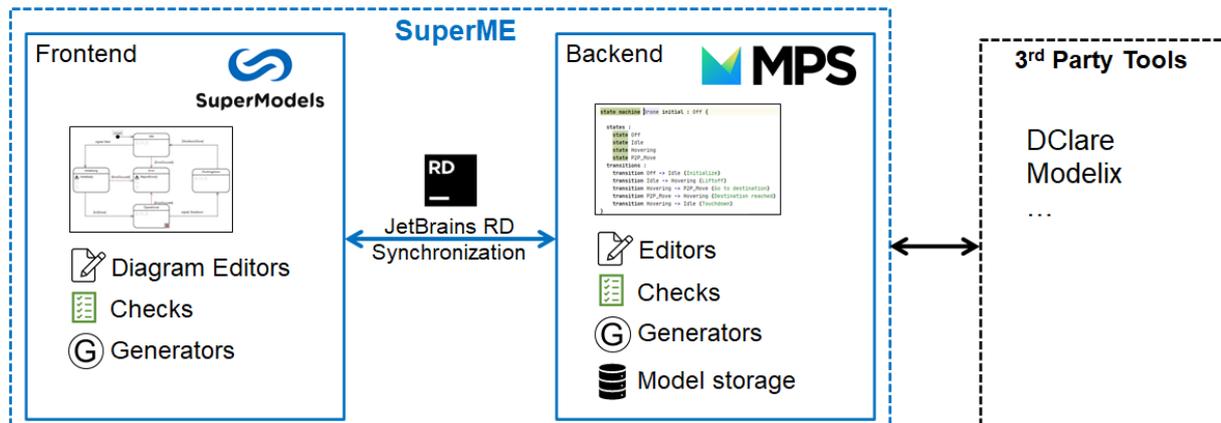

Figure 7. Architecture for UC7 combining SuperModels and MPS.

UC7 is about creating a blended ME that combines the strengths of SuperModels and MPS. It's tentatively called SuperME. The high-level overview of the chosen architecture is shown in Figure 7. As depicted, it combines SuperModels and MPS into SuperME and thus satisfies core requirement C7.1 identified in Deliverable D2.2. SuperModels and MPS are not just mechanically combined, they are deeply integrated. Supermodels was assigned the role of frontend and MPS the role of backend. In that way we can combine the stronger modeling foundations of MPS with the high-fidelity diagrammatic editors of SuperModels (see Figure 8).
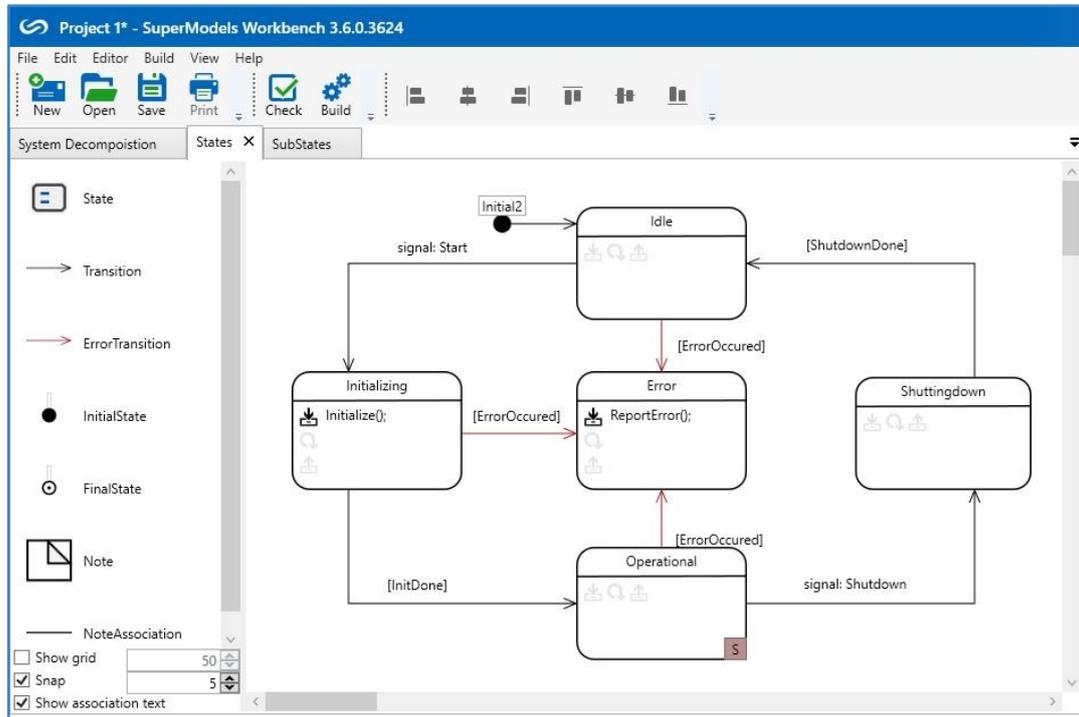
Figure 8. Supermodels high fidelity diagrammatic editor for state machines DSML.

SuperModels and MPS need to exchange data to properly interact. We chose to integrate the two MEs at the abstract model level meaning that the data exchanged is in the form of ASTs. MPS provides a model server that makes the ASTs accessible to a model client in Supermodels for both querying and modification. This allows for Supermodels editors to show and modify the AST and thus to behave like an MPS projectional editor. This also allows for Supermodels generators and model checkers to query (read-only) the AST. The fact that Supermodels has access to the models on AST level makes it possible to preserve existing Supermodels DSMLs. That means we can reuse existing DSMLs editors, generators and model checkers thus satisfying core requirement C7.8 identified in Deliverable D2.2. We can also combine them with existing (and new) MPS DSMLs. In this way, the chosen architecture supports the satisfaction of core requirements C7.2, C7.3, C7.4, C7.5, C7.9, C7.10 in Deliverable D2.2.

JetBrains RD was selected as an interfacing technology that allows inter-process communication between MPS (JVM process) and Supermodels (.NET process). JetBrains RD fits well in SuperME architecture since it provides a server and a client that share data in a common predefined structure, i.e., the instances of the data structure at both sides are kept automatically synchronized in real-time. This capability is applied to synchronize model ASTs between model server in MPS and model client in Supermodels. That enables the fulfilment of technical requirements T7.1, T7.3, T7.4, T7.9. JetBrains RD also allows for remote procedure calls between server and client which is used to propagate user actions like file opening/saving, generation triggering, model checks triggering etc. That enables the satisfaction of technical requirements T7.5, T7.6, T7.10.

It is worth mentioning that the model server and model client are generic, i.e., not DSML specific which makes them applicable in a context broader than SuperME and BUMBLE. The model server is packaged as an MPS plugin and is developed in Kotlin. The model client is developed in C# but it's possible to develop one in C++.

Further, the model storage responsibility was allocated solely to MPS. That means that models expressed in Supermodels DSMLs must be hosted on MPS and accessed via the model server. So, the structural aspect of Supermodels DSMLs must be implemented in MPS but all the other DSML aspects can be reused.

Having Supermodels and MPS models stored at one place (namely the MPS model repository) allows us to exploit the MPS facility of hyperlink-like traceability (see Section 2.2.4) within and across DSMLs. So, we can link between models from both Supermodels and MPS thus the relations between the different aspects of a system can be expressed explicitly which creates the possibility to monitor and/or maintain the consistency between them. As mentioned in Section 2.2.3 the MPS facilities provided out-of-the-box satisfy core requirements C7.6 and C7.7 identified in Deliverable D2.2.

3.3 Model Life-Cycle Management

3.3.1 Architectural Description for UC4 (Cross-disciplinary Coupling of Models)
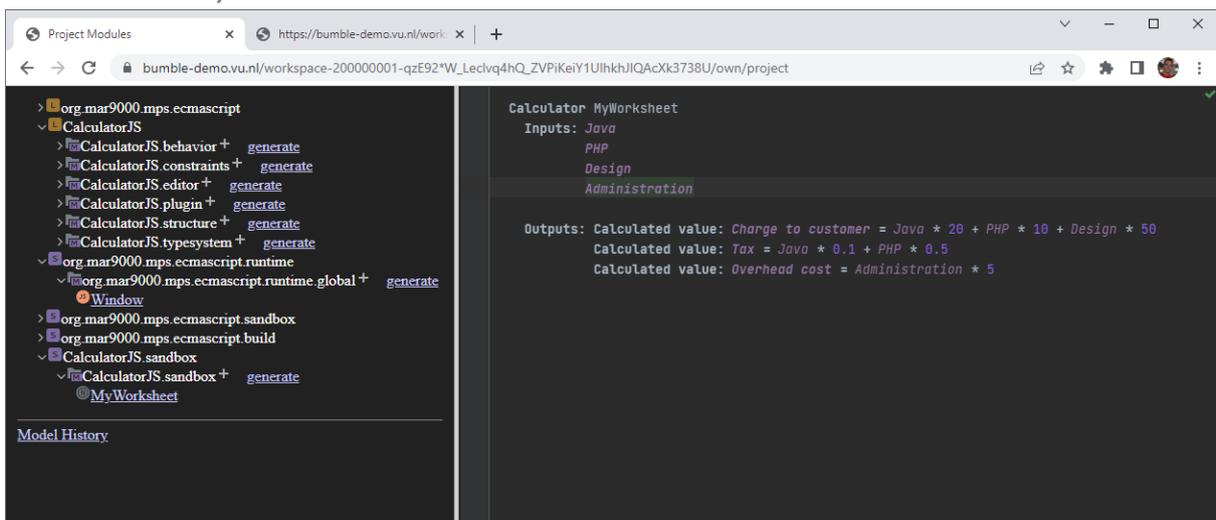

Figure 9. Screenshot of Modelix for UC4.

For UC4, Canon Production Printing intends to realize a low threshold online collaborative modeling environment with Modelix. Modelix has been iteratively developed and deployed in the context of CPP. A screenshot of the BUMBLE Demonstrator workspace on the VU servers is shown in Figure 9. Users are able to log in using their existing user account, for example using OAUTH single-sign-on authentication, see core requirement C4.1 in Deliverable D2.2. In the collaborative modeling environment for UC4, the modeling capabilities are exposed through Projector on a Modelix model store. The artifacts can be generated from the models, are then available for download from the Modelix environment to the user's local workstation (core requirements C4.6 and C4.7 in Deliverable D2.2.

There is also a need to bound the actions and visibility of model (elements) to particular users, to ensure some level of consistency of the models. A snapshot of (the results of) the models needs to

be picked up by downstream development environments. Therefore, models (and/or generated artifacts) need to be coupled to a remote version control mechanism (such as Git).

Multiple data-connectors have been realized and deployed to enable interaction between Modelix databases and other (source code or artifact) repositories (see Figure 10). This allows specifying a workspace's languages, their versions, as well as the deployment of input models and committing changes back to a Git repository. The model versions are explicitly tracked in the Modelix Model Server, therefore achieving a foundation for model management.
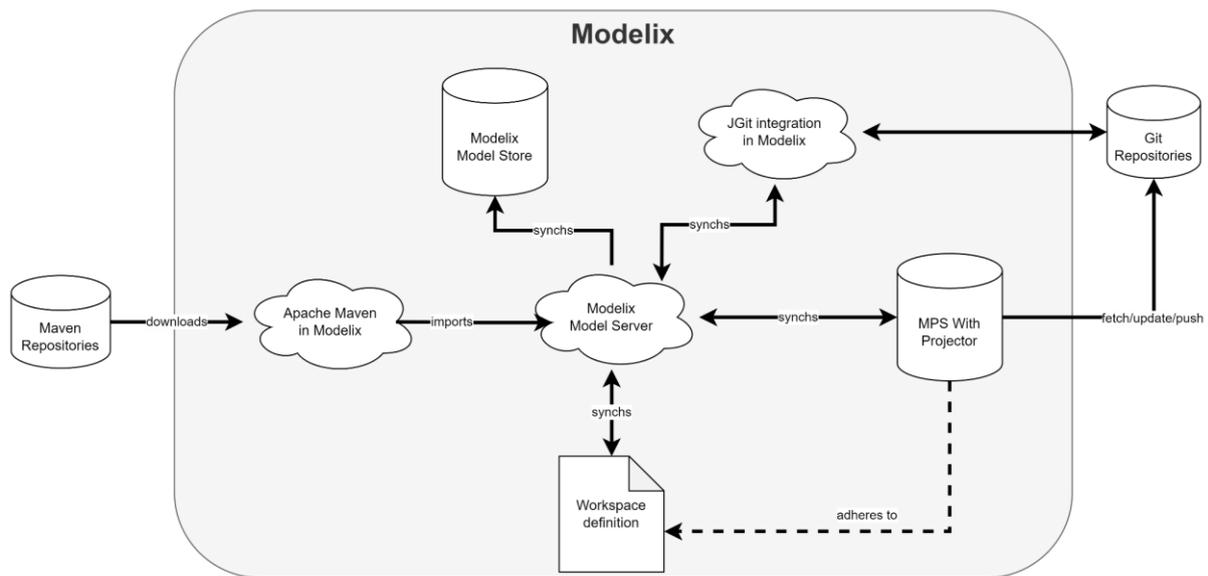

Figure 10. Data source connectors for Modelix.

3.3.2 Architectural Description for UC7 (Multi- and Cross-Disciplinary Modeling Workbench)

For UC7, Sioux intends to exploit existing MPS facilities for Model Life-Cycle Management described in Sections 2.2.3 and 2.2.4. The added value is that through SuperME those capabilities will be made available to existing SuperModels DSMLs. No MPS extension will be developed: all requirements in Deliverable D2.2 related to Model-Life-Cycle Management in UC7 are satisfied out-of-the-box.

4 Architecture description of Technology Demonstrator

The joint technology demo between the university partner VU Amsterdam and OEM partners Sioux and Canon concerns the interaction between multiple collaborative technology platforms. Each platform enables particular collaborative modeling capabilities; i.e., either centralized collaboration (BUMBLE Collaboration Engine, Modelix), decentralized collaboration (Dclare), remote synchronization over multiple clients using the same language and model, and across different technology stacks and models (RService).
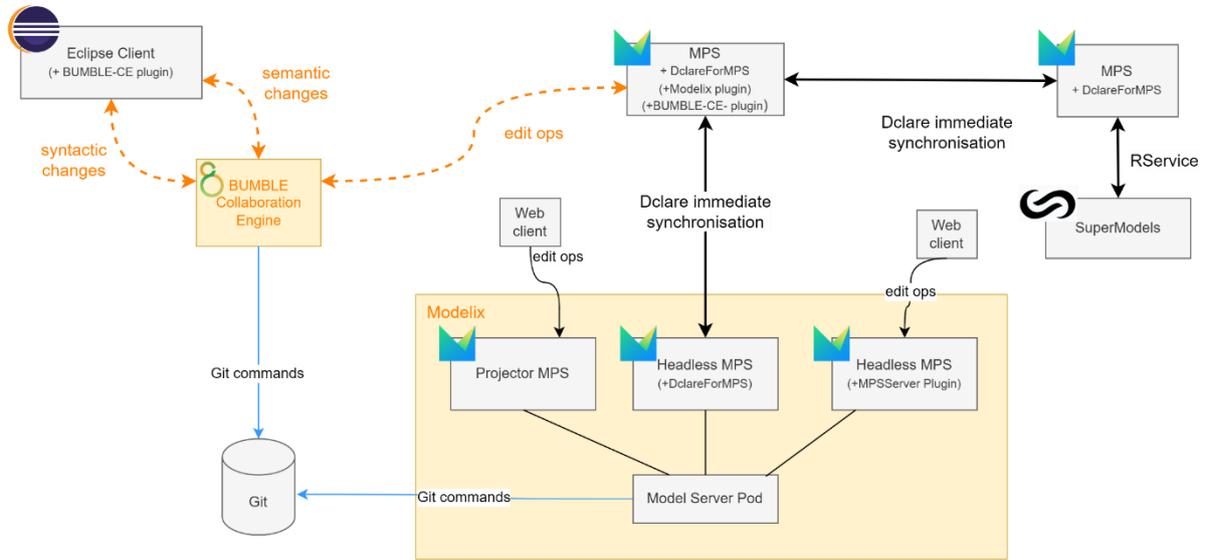
Figure 11. Logical view of joint demo on collaborative modeling capabilities.

Figure 11 shows the connections between the deployed languages and how the model contents are synchronized over different connections. The different clients where users interact with the models in different syntaxes are depicted at the top (MPS, web client, Eclipse, SuperModels), while the model serving parts that take care of synchronization between different stages are in the lower parts of the picture.
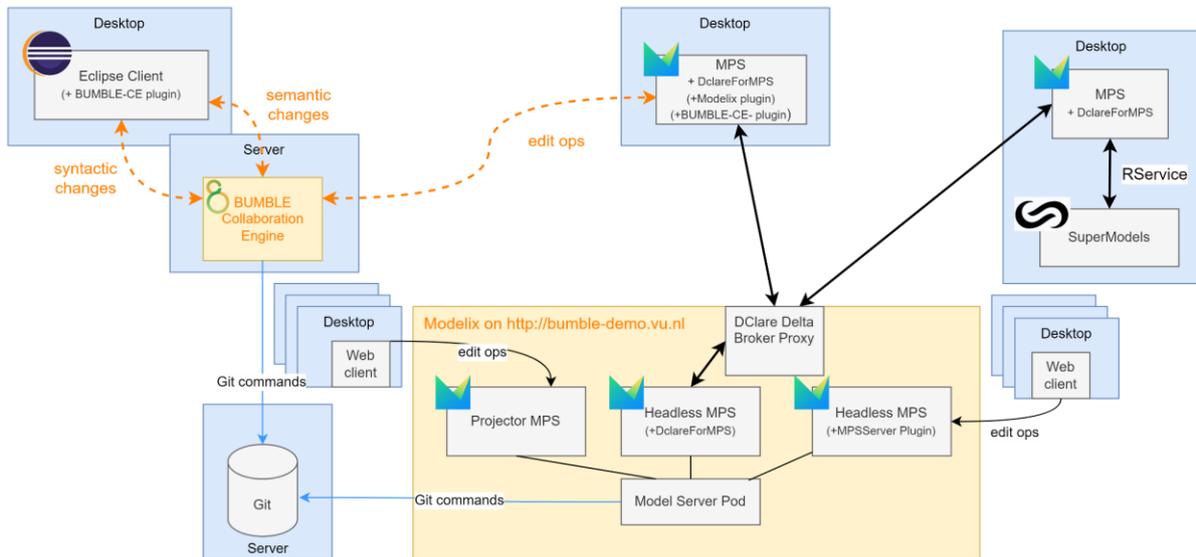

Figure 12. Deployment view of joint demo on collaborative modeling capabilities.

Figure 12 shows how these components are deployed onto different machines. Most deployments consist of an installation of Eclipse or MPS with the corresponding plugins. The Modelix infrastructure is deployed on a Kubernetes cluster using Rancher as a cluster manager. The Modelix workspace definitions specify how MPS instances are created using Projector. The MPS instances are each hosted in their own pod.

The MPS instances which have the DclareForMPS plugin enabled can be connected to the delta broker. The delta broker then distributes and synchronizes all model changes. These model

changes can come from updates of the Modelix centralized Model Server, or from user changes directly. The technology does not distinguish between these changes.

There is one Desktop deployment which consists of an MPS instance (incl. DclareForMPS plugin) and a SuperModels instance that are connected via RService. RService synchronizes (propagates) model changes originating from either side. It doesn't distinguish between user caused changes and tool caused changes.

If all Modelix MPS instances would broadcast and receive the changes from the Dclare delta broker, the performance may needlessly suffer. Because there is only one MPS Headless instance connected from the Modelix cloud, the synchronizations are only sent and received once. These changes are propagated from/to the Modelix Model Server, and from there again propagated to the MPS instances that do not have the Dclare remote synchronization enabled.

Modelers working in an Eclipse environment can also collaborate in real-time with those working in MPS via the cross-platform BUMBLE Collaboration Engine (which runs on its own server). Moreover, the BUMBLE Collaboration Engine supports also non-real-time synchronization, where GIT commits are used as synchronization points for stable versions of the models on which the real-time collaboration sessions are operating. According to such configuration, the GIT repository where the models are stored acts as an Eclipse-MPS reconciliation point for persisted models. Refer to deliverable D5.1v2, section 5.2 for a detailed architectural description of the BUMBLE Collaboration Engine and section 5.3 for its implementation details.

5 Conclusion

In this document, the MPS-based architecture platforms for the BUMBLE technologies in BUMBLE use cases are described. The main basic principles and existing features of the Jetbrains MPS technology are described and used to evaluate to what degree BUMBLE has to extend this base technology in order to satisfy the various requirements identified in Deliverable D2.2. Subsequently, an architectural view on the identified required extensions is described, taking the contexts of the different use cases specifically into account, as the general logical architecture in BUMBLE as well as the architectures for real-time collaboration are covered in Deliverables D3.3 and D5.1, respectively.

References

[1]   Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Domain-Specific Languages in Practice with Jetbrains MPS. Springer (2021), ISBN 978-3-030-73758-0

[2]   Voelter, M.: Language and IDE modularization and composition with MPS. In: Generative and transformational techniques in software engineering IV, pp. 383–430. Springer (2013)

[3]   Voelter, M., Lisson, S.: Supporting diverse notations in MPS' projectional editor. GEMOC@MoDELS 2014, pp. 7–16

[4]   Gamma E., Vlissides J., Helm R., Johnson R.: Design Patterns: Elements of Reusable Object-Oriented Software. Baker & Taylor, ISBN 9780201633610

[5]   Kay A., Ingalls D., Goldberg A.: Smalltalk, see https://en.wikipedia.org/wiki/Smalltalk

[6]   Krasner, Glenn E.; Pope, Stephen T.: A cookbook for using the model−view controller user interface paradigm in Smalltalk-80. The Journal of Object Technology. SIGS Publications. 1 (3): 26–49, 1988

[7]   Diagrams Plugin, https://jetbrains.github.io/MPS-extensions/extensions/diagrams/#diagrams, Itemis

[8]   Birken, K.: MPS Applications in the Browser: Cloud MPS, https://blogs.itemis.com/en/mps-applications-in-the-browser-cloud-mps, 2020

[9]   Lißon, S.: A Next Generation Language Workbench Native to the Web and Cloud, https://github.com/modelix/modelix, 2020

[10]   Google: An Application Design Framework and Development Platform for Creating Efficient and Sophisticated Single-Page Web-Apps (2010–2020), https://angular.io