ITEA Office
High Tech Campus 69 - 3
5656 AG Eindhoven
The Netherlands

T + 31 88 003 6136
E info@itea3.org
W www.itea3.org

ITEA 3 is a EUREKA strategic ICT cluster programme

BUMBLE Deliverable D3.6

Integration with external tools

Edited by: BUMBLE Team

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

Release	Date	Reason of change	Status	Distribution
V0.1	20/12/2022	First draft, Internally-revised draft, chapter structure in place	Draft	WP Leaders
V0.2	04/02/2023	Polished on chapter structure, added introduction, Executive summary	Draft	Contributors
V1.0	24/03/2023	Finalization	Final	Uploaded to ITEA portal

Executive summary

This deliverable includes the software results of task T3.4 related to the integration with external tools (pre-existing tools/software not developed as part of the BUMBLE project). The results from various integration activities are presented. The purpose of the deliverable is to illustrate the investigations done to mainly integrate different solutions that already exist, how to make workflows smoother, migrate/re-use data from one solution to another, and so on. The external tools do not necessarily have to be modeling software, but regular software identified as being good options to integrate with to improve the overall user experience.

There are no previous versions of this document, so this is the only and final version of the D3.6 document.

Contributors

Kousar Aslam, Ivano Malavolta	VU
Martin Axelsson, Mattias Mohlin	HCL
Federico Ciccozzi, Malvina Latifaj	MDU
Luca Berardinelli	MJKU
Roelof Hamberg, Joost van Pinxten	Canon
Ruud Meeuws, Alexander Darmonski	Sioux

Reviewers

Detlef Scholle	Pictor
Jörg Holtmann	GU

Acronyms

AST	Abstract Syntax Tree
DSML	Domain Specific Modelling Language
EMF	Eclipse Modelling Framework
MPS	Meta Programming System
UC	Use Case
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

1. Introduction

This deliverable focuses on the integrations with different software, either within the BUMBLE project or to external tools (pre-existing tools/software not developed as part of the BUMBLE project). The external tools do not necessarily have to be modeling software, but regular software identified as being good options to integrate with to improve the overall user experience. The below chapters will illustrate some of the tool integrations used or created as part of the BUMBLE project. The idea is to describe and document the integrations used. The following applicable contributors were identified and their integrations are described in the following chapters of this document.

- Chapter 2 - Integrating existing modeling editors into a real-time collaboration engine
- Chapter 3 - Migration from HCL RTist to RTist in Code – prototype
- Chapter 4 - Integration of multiple open-source frameworks for metamodelling and modelling
- Chapter 5 - Integration of JSONware and Modelware via JSONSchemaDSL
- Chapter 6 - Modelix + MPS
- Chapter 7 - SuperModels + MPS

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

Technology bricks	Description of main contributions	Main requirements
Platform Integration	We have integrated multiple Eclipse-based technologies as well as bridged Eclipse and MPS for blended (meta-)modelling	BT5, BT6, BT9, BT10, BT17, BT18

In chapter 2, VU elaborates on how they have designed the architecture and built a prototype implementation of a cross-platform real-time collaboration engine, based on the requirements elicited from the use-cases provided by the partners in the BUMBLE project. The software architecture and prototype implementation of this architecture have been discussed in D5.1-v2. To demonstrate the functioning of the BUMBLE collaboration engine – BUMBLE-CE, VU describes how to apply it in the context of collaborative modeling between Eclipse EMF and JetBrains MPS. The example is used as a common basic use case within the BUMBLE project, which involves the usage of a simple DSML for representing state machines. In chapter 2, VU explains the setup of collaboration on both EMF and MPS side, the State Machine metamodel and the TrafficSignal.statemachine conforming to this metamodel which is our main model for collaboration. The real-time collaboration is supported in both directions, that is, EMF to MPS and vice- versa.

In chapter 3, MDU elaborates on how they investigated the migration of models from HCL RTist to RTist in Code. Chapter 3 explains why such a migration is useful and how it is designed and implemented.

In chapter 4, MDU elaborates how they have investigated how to integrate several disjoint tools for metamodeling and modelling purposes. Note that the technical details of each integration are not provided, instead the focus is on the benefits of the integrations. Technical solutions and descriptions can be found in other deliverables for WP3 and WP4. The sub-chapters to chapter 4 covers e.g. integration of Sirius and Xtext, integration of EMF and MPS and integration of Draw.io and TextX.

In chapter 5, JKU elaborates on how to integrate JSONware and Modelware via JSONSchemaDSL, where JSONSchemaDSL is a model-driven approach to bridge two unrelated technical spaces (TS), namely JSONware and Modelware. The JSONware TS includes JSON (JavaScript Object Notation) and JSON Schema. Modelware TS refers to software tools and platforms that are used to develop, maintain, and manage models as cornerstone artifacts of a (software) engineering process. JSONSchemaDSL leverages the Eclipse Modeling Framework (EMF)-based software tools and platforms, like Xtext, Sirius, and GEMOC.

In chapter 6, Canon elaborates on different integrations between Modelix and MPS, for example Discuss on how to handle user authentication, Modelix Deployment, Authorization and User Management as well as Integration with other Front End technologies

In chapter 7, Sioux elaborates on their integration between SuperModels and MPS, where SuperModels serves as the frontend and MPS the backend. This chapter illustrates the architecture of this integrations and the technologies used.

2. Integrating Existing Modeling Editors into a Real-time Collaboration Engine

VU has designed the architecture and built a prototype implementation of a cross-platform real-time collaboration engine, based on the requirements elicited from the use cases provided by the partners in the BUMBLE project. The software architecture and prototype implementation of this architecture are part of the activities of WP5 and are presented in detail in deliverable D5.1. For the sake of understandability, we show the architecture of the BUMBLE generic cross-platform real-time collaboration engine in Figure 2.1 below.

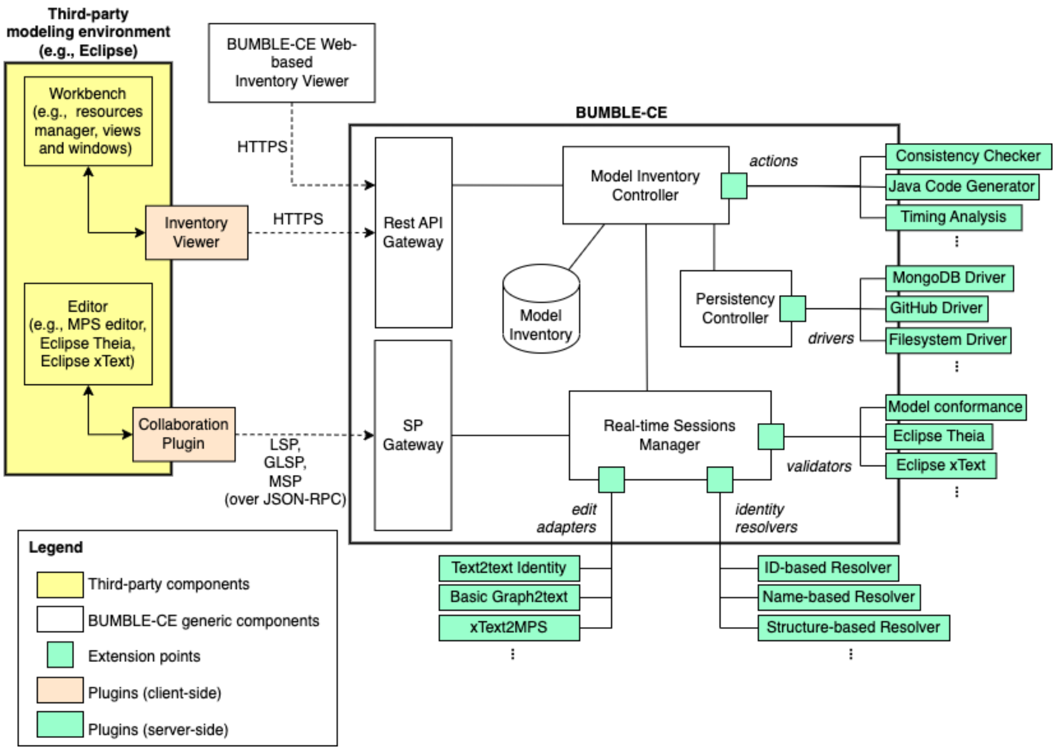


Figure 2.1. Detailed architecture of the BUMBLE cross-platform real-time collaboration engine (presented in detail in D5.1)

To demonstrate the functional principle of the BUMBLE collaboration engine – BUMBLE-CE, we describe how to apply it in the context of collaborative modelling between Eclipse EMF and JetBrains MPS. The example is used as a common basic use case within the BUMBLE project, which involves the usage of a simple DSML for representing state machines. Below, we explain the setup of collaboration on both EMF and MPS side (i.e., *Collaboration plugin* from Figure 2.1) , the *State Machine* metamodel and the *TrafficSignal.statemachine* conforming to this metamodel which is our main model for collaboration. The real-time collaboration is supported in both directions, that is, EMF to MPS and vice- versa. In this chapter, the term *Modeller* is used to refer to a modelling tool user.

2.1 Collaboration Plugin for EMF

On the EMF side, the *Collaboration Plugin*[1] uses an activator to control its lifecycle. It activates a new window of a run-time editor and initiates the server client when the modeller launches the collaboration application and terminates the plugin when the modeller exits the editor window. The run-time model editor allows the modeller to import new models from various resources or generate model instances from existing metamodels (i.e., an Ecore model).

In this prototype, we use the publish-subscribe pattern for data communication. The Eclipse *Collaboration Plugin* listens to the changes happening in the editor domain and sends them to the server. The EMF cloud server acts as the publisher that receives messages from clients (i.e., subscribers) and propagates the changes to all of its subscribers. The changes that were received by the client are then applied to the local model in the editor domain. In the example application, we assume the Ecore model already exists in the server so that modellers can post models generated from it into the server. Once the collaboration button is clicked, the plugin first checks if the local model has a copy in the server by sending a GET request to the server. If the server confirms that the model exists, then the model is pulled from the server and the local version stored in the editor's resource set is replaced with the server version. If the model is not in the server, the plugin will POST the local model to the server. At this stage, the communication between the client and server is done via REST API. Once the initiation is done, the client subscribes to the model in the server through WebSocket.

Listener

In the context of EMF, the model instance itself and its contents can all be represented using EObjects. The EMF library provides a *ChangeRecorder* class that can listen to changes in EObjects and returns notifications that include the details of the changes. In this prototype, we only listen to semantic changes. This means that the listener will only be notified if a property change is fully done, e.g., a modification in the name of a node. The collaboration plugin parses the notification and converts it into a JSON patch based on the operation type of the change (i.e., replace, remove, and addition). The JSON patch includes the operation type, the path that locates the node and the new value.

Propagation

The JSON patch is sent to the server if a change is detected. Once it is received, the server first applies the change to the model on its side and publishes it to the subscribers. The published message is again in the format of JSON patch which resembles the JSON patch it receives from clients. In practice, the JSON patch sent to and received from the server are not always exactly the same. Operations such as removal and addition may cause position changes in other nodes. Thus, a single JSON patch sent from the client may eventually lead to a list of JSON patches published from the server to its subscribers and each JSON patch in the list states a single step of change. The propagation function is provided by the EMF cloud server and it ignores the platform of the JSON patch sender and model subscribers. As long as the message the client sends to and receives from the server is in the format of JSON patch, the platform of the client is not restricted.

[1] https://github.com/Yunabell-VU/nl.vu.cs.bumble.emfcollaborationplugin

Subscription

The Eclipse collaboration plugin subscribes to the model in the server via WebSocket. It receives JSON patches from the server and applies the changes to the local model in the modeller's editor domain. This is achieved by parsing the JSON patches one by one and for every single patch. The parser locates the position of the node to be changed through the path value in the patch, creates a new EObject with the new value, and attaches it to the position located. From the modeller's perspective, the model in the editor domain changes with new properties on certain nodes in real-time if other clients made a change on their side.

2.2 Collaboration Plugin for MPS

MPS provides an architecture for setting up standalone plugins to achieve particular functionalities. To enable real-time collaboration on the MPS side, we needed to implement two functionalities: enable/start a collaboration session, and disable collaboration. Each functionality comprises a sequence of action coded to initiate different sets of logic. These actions are combined into a group called *Collaboration* which is configured to display these actions in the context menu for nodes. A model in MPS is stored in a data structure called node, specifically root node. Thus in order to begin collaboration, a user right-clicks on the root node and clicks on *Enable collaboration*. This action from the modeler will launch the *Collaboration plugin* in MPS, if it is not already running. Upon the launch of this plugin, three components are fired up one after another, which we describe in the following.

Synchronizer

This component is responsible for ensuring that all aspects of the selected node in MPS are equivalent to that of the model in the EMF model server. This implies that the language structure of the given node complies with the Ecore logic of the EMF model, as well as the content at the start of the collaboration. This is achieved with the following sub-components:

1. **Validator**: Ensures that the selected node is present in the EMF model server by performing a GET request with the name of the selected node. If available, the *validator* checks whether the language structure of the selected node conforms to the metamodel of the corresponding model in the model server. This is achieved with the help of a *mapper* component. If the validation process fails for any one of the two steps mentioned above, the collaboration is automatically disabled.
2. **Mapper**: Since the logic of the language structure of MPS and Ecore are packed differently, this component cross-checks the *EStructuralFeatures* and *ESuperTypes* of Ecore with the language structure in MPS. Each root node of the language structure of MPS represents a *concept*, which might or might or might not extend and implement other concepts. A *concept* is referred to as an *EClass* in EMF terminology. In order to store and read Ecore data, the Ecore is fetched from the server via GET request in XMI format, packed as a JSON string, and stored in data classes with Jackson's Object Mapper[2]. In order to read data from the root nodes of MPS, one of the MPS's platform languages, SModel[3], was used. After ensuring that all EClasses from the EMF metamodel are present in the language structure of MPS, the EStructural features and ESupertypes of each EClass are compared.

[2] https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html
[3] https://github.com/JetBrains/MPS/tree/master/core/openapi/source/org/jetbrains/mps/openapi

3. **ContentSynchronizer**: After mapping is performed between MPS node and EMF models, the *ContentSynchronizer* component compares and synchronizes the *content* of the selected node in MPS to that of the EMF model in the model server. The *ContentSynchronizer* performs structural comparison of the values for all the attributes and other properties between EMF and MPS models. If there is any inconsistency found, then the content of the selected node is overwritten to that of the model present in the model server.

Communication between EMF Model Server and MPS

The communication between MPS and EMF is established via the external library, emfcloud-modelserver[4]. Originally, the emfcloud-modelserver has been designed to accommodate EMF logic so we designed our own *mapper* component discussed earlier to interpret and exploit this logic for MPS. Models stored on the model server can be attained via GET requests in various formats, BUMBLE-CE uses JSON format. In order to propagate the changes made during the collaboration session on a given model, the modeller is subscribed to the model via websocket. These edit operations on the MPS side are received as JSON patches. Once subscribed, patch operations can be performed on the models using MPS's SModel language to reflect any change made to the model in the model server locally in MPS.

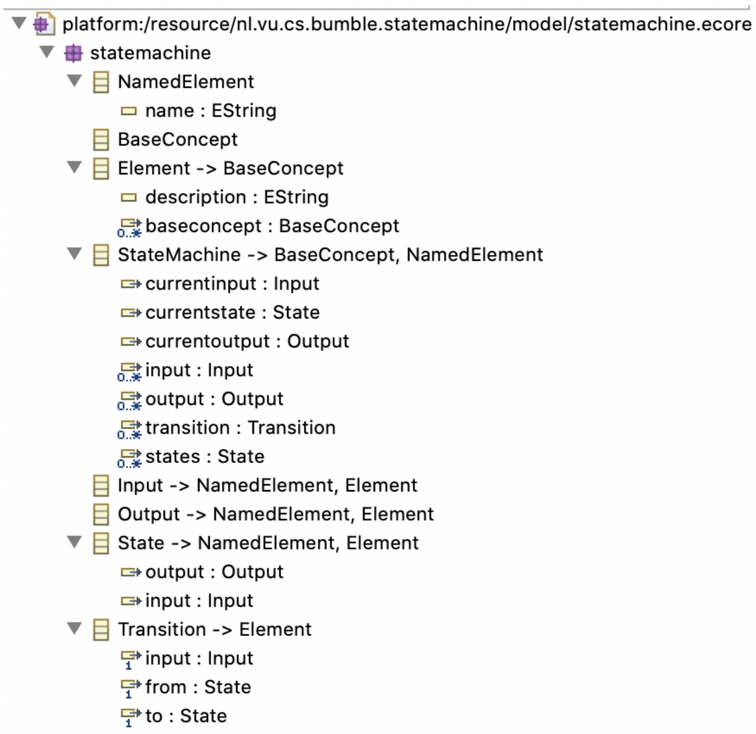
Listener

MPS provides a library called Open API[5] which provides controlled access to a given model and also provides interfaces in order to provide a custom implementation in various aspects, in our case for the listener. Among the listeners provided by OpenAPI, we are using SNodeChangeListener in our work. When an edit operation is performed on a given node in MPS, SNodeChangeListener is notified via MPS's message bus and the operation is propagated to the EMF model server. For now, the listener is configured to report changes character by character on MPS side in the BUMBLE-CE. In the future, we will refine the implementation to receive and propagate the change as a whole.

When the user decides to end the collaboration session, the user right clicks the node involved in their collaboration session and clicks *Disable collaboration*. With this action, the *Listener* and *EmfModelServer websocket client* are disabled one after another.

[4] https://github.com/eclipse-emfcloud/emfcloud-modelserver
[5] https://github.com/JetBrains/MPS/tree/master/core/openapi/source/org/jetbrains/mps

2.3 State Machine DSML



```
platform:/resource/nl.vu.cs.bumble.statemachine/model/statemachine.ecore
  statemachine
    NamedElement
      name : EString
    BaseConcept
    Element -> BaseConcept
      description : EString
      baseconcept : BaseConcept
    StateMachine -> BaseConcept, NamedElement
      currentinput : Input
      currentstate : State
      currentoutput : Output
      input : Input
      output : Output
      transition : Transition
      states : State
    Input -> NamedElement, Element
    Output -> NamedElement, Element
    State -> NamedElement, Element
      output : Output
      input : Input
    Transition -> Element
      input : Input
      from : State
      to : State
```

Figure 2.2: Ecore Metamodel of *State Machine*

The *State Machine* DSML was initially designed by the Modeling Value Group (MVG) in MPS[6]. We translated the *State Machine* DSML from MPS to EMF manually. The Ecore metamodel of the *State Machine* DSML is shown in Figure 2.2. According to thismetamodel, a *State Machine* extends three metaclasses: *BaseConcept*, *NamedElement* and *Element*; also, a state machine can contain sets of *Input* and *Output* elements, *States*, and *Transitions*.

The *State Machine* keeps track of the current input, output and state through the references *currentinput*, *currentoutput* and *currentstate* to the metaclasses *Input, Output* and *State*. The current *State* receives a trigger defined by the metaclass *Input*, transitions to a new *State* and produces a new *Output*. The *Transition* metaclass handles the switching from one state to another state on receiving an input through references to *Input* and *State* metaclasses. The metaclasses *Input, Output, and State* have attributes *name* and *description*. The metaclass *State* has references to the metaclasses *Input* and *Output*; and the metaclass *Transition* has references to the metaclasses *Input* and *State*. In this way, our metamodel illustrates essential concepts to define a state machine. BUMBLE-CE considers the modeller to be responsible for storing the metamodel of the used DSML (*StateMachine* in our case) in the server. We expect that the metamodel of the used DSML is not subjected to changes frequently and storing the metamodel on the server is only a one-time action performed by the modeller.

[6] https://github.com/ModelingValueGroup/statemachines

2.4 TrafficSignal.statemachine: Model for Collaboration

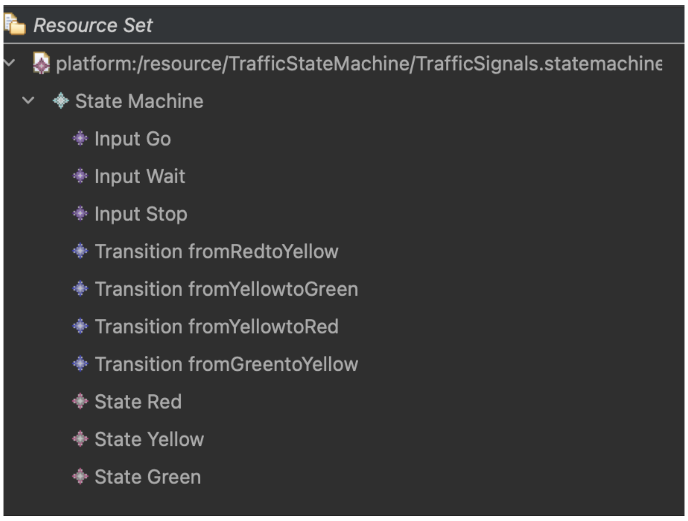
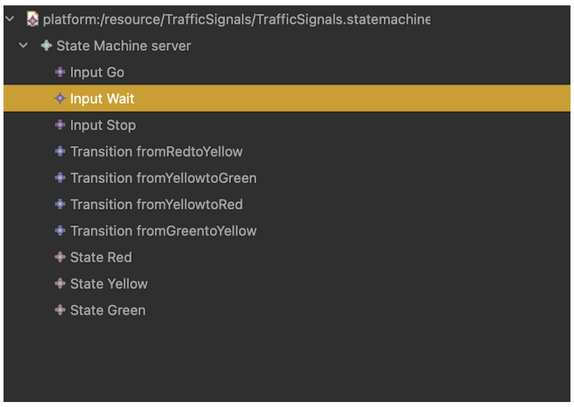

Resource Set
platform:/resource/TrafficStateMachine/TrafficSignals.statemachine
State Machine
Input Go
Input Wait
Input Stop
Transition fromRedtoYellow
Transition fromYellowtoGreen
Transition fromYellowtoRed
Transition fromGreentoYellow
State Red
State Yellow
State Green

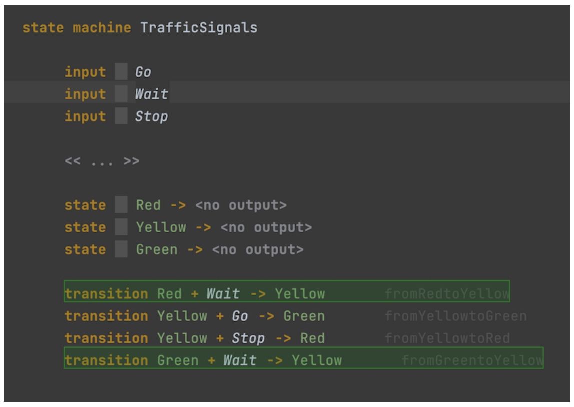
Figure 2.2: Model *TrafficSignals.statemachine* that conforms to the metamodel shown in Figure 2.1

We have generated a model *TrafficSignal.statemachine* conforming to the *StateMachine* metamodel shown in Figure 2.1. The *StateMachine* metamodel is imported into the editor at run-time through the file system. The *TrafficSignal.statemachine* model represents the working of a traffic signal with three inputs *Go, Wait and Stop* and three states *Red, Yellow and Green*. When in state *Red* or *Green* and on receiving inputs *Go* or *Stop* respectively, the *TrafficSignal.statemachine* will transition to the state *Yellow*. When in state *Yellow*, if input *Stop* is received, the *TrafficSignal.statemachine* transitions to the state *Red* and if input *Go* is received, the *TrafficSignal.statemachine* transitions to the state *Green*. The *TrafficSignal.statemachine* will be our subject of collaboration between EMF and MPS.

To start the collaboration, a modeller right clicks on the *TrafficSignal.statemachine* in their respective editor and clicks on the *Start Collaboration* menu item provided by the BUMBLE-CE. The *Session Manager* uploads the *TrafficSignal.statemachine* on the server so that other modellers can also join the collaboration session. Once a modeller performs an operation on the model (for instance, adds or deletes a node, edits the name of an attribute, or changes the position of a node in the tree), the change will be propagated on the other side to the subscribed models through web sockets. Figure 2.3 shows the collaboration scenario for *Replace* operation between EMF and MPS. The *TrafficSignal.statemachine* is shown for both EMF and MPS before starting the collaboration in Figures 2.3(a) and 2.3(b) respectively. The modeller on the EMF side replaces the name of Input *Wait* with *Hold* and the change is immediately reflected on the MPS side, see Figures 2.3(c) and 2.3(d). Figure 2.3(e) presents the log history for events happening on MPS side showing that the *Replace* operation is successfully received and corresponding changes are made to the model.



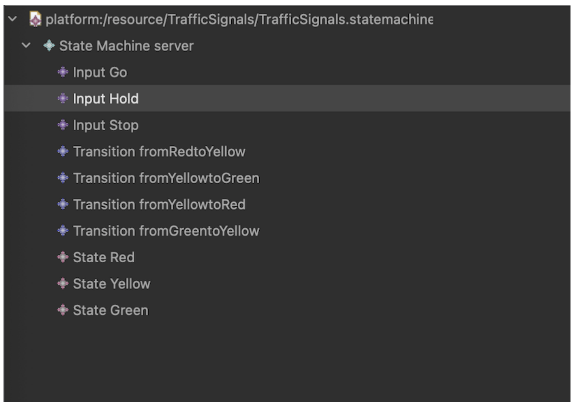
```
platform:/resource/TrafficSignals/TrafficSignals.statemachine
  State Machine server
    Input Go
    Input Wait
    Input Stop
    Transition fromRedtoYellow
    Transition fromYellowtoGreen
    Transition fromYellowtoRed
    Transition fromGreentoYellow
    State Red
    State Yellow
    State Green
```



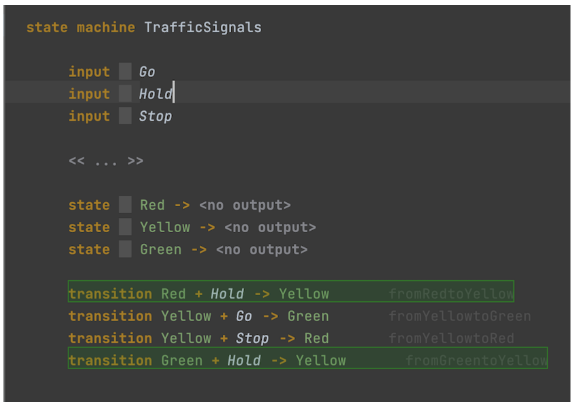
```
state machine TrafficSignals

    input  Go
    input  Wait
    input  Stop

    << ... >>

    state  Red -> <no output>
    state  Yellow -> <no output>
    state  Green -> <no output>

    transition Red + Wait -> Yellow        fromRedtoYellow
    transition Yellow + Go -> Green        fromYellowtoGreen
    transition Yellow + Stop -> Red        fromYellowtoRed
    transition Green + Wait -> Yellow      fromGreentoYellow
```

(a) TrafficSignals.statemachine in EMF before any edit operation

(b) TrafficSignals.statemachine in MPS before any edit operation



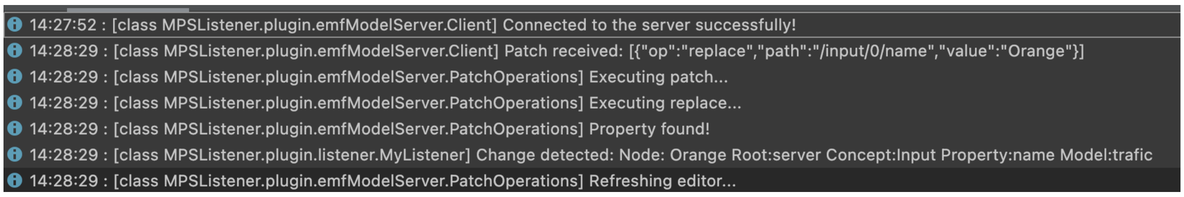
```
platform:/resource/TrafficSignals/TrafficSignals.statemachine
  State Machine server
    Input Go
    Input Hold
    Input Stop
    Transition fromRedtoYellow
    Transition fromYellowtoGreen
    Transition fromYellowtoRed
    Transition fromGreentoYellow
    State Red
    State Yellow
    State Green
```



```
state machine TrafficSignals

    input  Go
    input  Hold
    input  Stop

    << ... >>

    state  Red -> <no output>
    state  Yellow -> <no output>
    state  Green -> <no output>

    transition Red + Hold -> Yellow        fromRedtoYellow
    transition Yellow + Go -> Green        fromYellowtoGreen
    transition Yellow + Stop -> Red        fromYellowtoRed
    transition Green + Hold -> Yellow      fromGreentoYellow
```

(c) TrafficSignals.statemachine in EMF, the name of Input Wait is edited to Hold. The operation is propagated to the MPS side.

(d) TrafficSignals.statemachine in MPS, the edit operation to change the name of Input Wait to Hold is received on the MPS side and reflected in the model.



```
14:27:52 : [class MPSListener.plugin.emfModelServer.Client] Connected to the server successfully!
14:28:29 : [class MPSListener.plugin.emfModelServer.Client] Patch received: [{"op":"replace","path":"/input/0/name","value":"Orange"}]
14:28:29 : [class MPSListener.plugin.emfModelServer.PatchOperations] Executing patch...
14:28:29 : [class MPSListener.plugin.emfModelServer.PatchOperations] Executing replace...
14:28:29 : [class MPSListener.plugin.emfModelServer.PatchOperations] Property found!
14:28:29 : [class MPSListener.plugin.listener.MyListener] Change detected: Node: Orange Root:server Concept:Input Property:name Model:trafic
14:28:29 : [class MPSListener.plugin.emfModelServer.PatchOperations] Refreshing editor...
```

(e) Logs in MPS showing that the edit operation is successfully received and propagated in the TrafficSignals.statemachine

Figure 2.3: Example scenario of cross-platform (between Eclipse EMF and JetBrains MPS) real-time collaboration using BUMBLE-CE

3. Migration from HCL RTist to RTist in Code – Prototype

HCL RTist is an Eclipse-based modeling and development environment for creating complex, event-driven, real-time applications. RTist in Code is a next generation tool for creating stateful real-time C++ applications developed as an extension for Visual Studio Code (VS Code) and Eclipse Theia. It is based on the language server technology and began as an effort to provide support for other integrated development environments (IDEs) than Eclipse. This effort is based on the premise that developers may benefit from modeling in multiple IDEs in order to take advantage of the strengths of each and choose the one that is most appropriate for their purposes. Through the use of language server technology, language functionality is decoupled from the IDE being used by separating language-aware modules into separate processes and communicating via the Language Server Protocol (LSP).

RTist in Code is available for both Visual Studio Code and Eclipse Theia for the following reasons. First, both VS Code and Eclipse Theia share the same extension model, meaning that an extension developed for one can also run in the other. Moreover, they both rely on web-based technologies, so developers can access their functionality through a web browser. This is particularly useful when working remotely or across multiple devices. Furthermore, both VS Code and Eclipse Theia are popular IDEs on the market, which means that they have a significant user base.

While RTist in Code represents a promising solution, models defined in HCL RTist need to be translated into models that can be used in RTist in Code. This is due to the fact that models defined in HCL RTist apply the UML Real Time and CPPPropertySet profiles (the former defines the real-time specific extensions to UML, and the latter defines properties related to transforming models to C++ code). In contrast, the models in RTist in Code conform to the Art textual language. Art is an extension to C++ that provides high-level concepts not directly available in C++. The Art compiler converts these concepts into C++ code. Therefore, the models must be translated in order for them to be used in RTist in Code.

The automatic migration of models developed in HCL RTist for Eclipse into models that are compatible with RTist in Code for Visual Studio Code and Eclipse Theia serves several purposes. First, it eliminates the need for developers to manually migrate models, thereby saving time and effort. In addition, it ensures that the models conform to the Art textual syntax and may be used in RTist in Code, thereby reducing the risk of errors or issues arising during manual migration. Furthermore, it improves the overall process by providing an automated solution that can handle the migration of models in an efficient and cost-effective manner and supports the integration of HCL RTist with RTist in Code.

In BUMBLE, we are providing an approach that supports the automatic migration of models from HCL RTist to RTist in Code through a model exporter. This model exporter enables users to quickly and easily translate their .emx models defined in HCL RTist (Eclipse IDE) into .art models that can be used in RTist in Code (Eclipse Theia or Visual Studio Code). As a result, users do not need to recreate their models from scratch, hence saving them a lot of time and effort. Moreover, as the Art language can be subject to future changes, we provide a solution where the transformations from .emx to .art models can be regenerated with a minimal mapping effort from developers. The workflow of the approach is detailed in Figure 1.

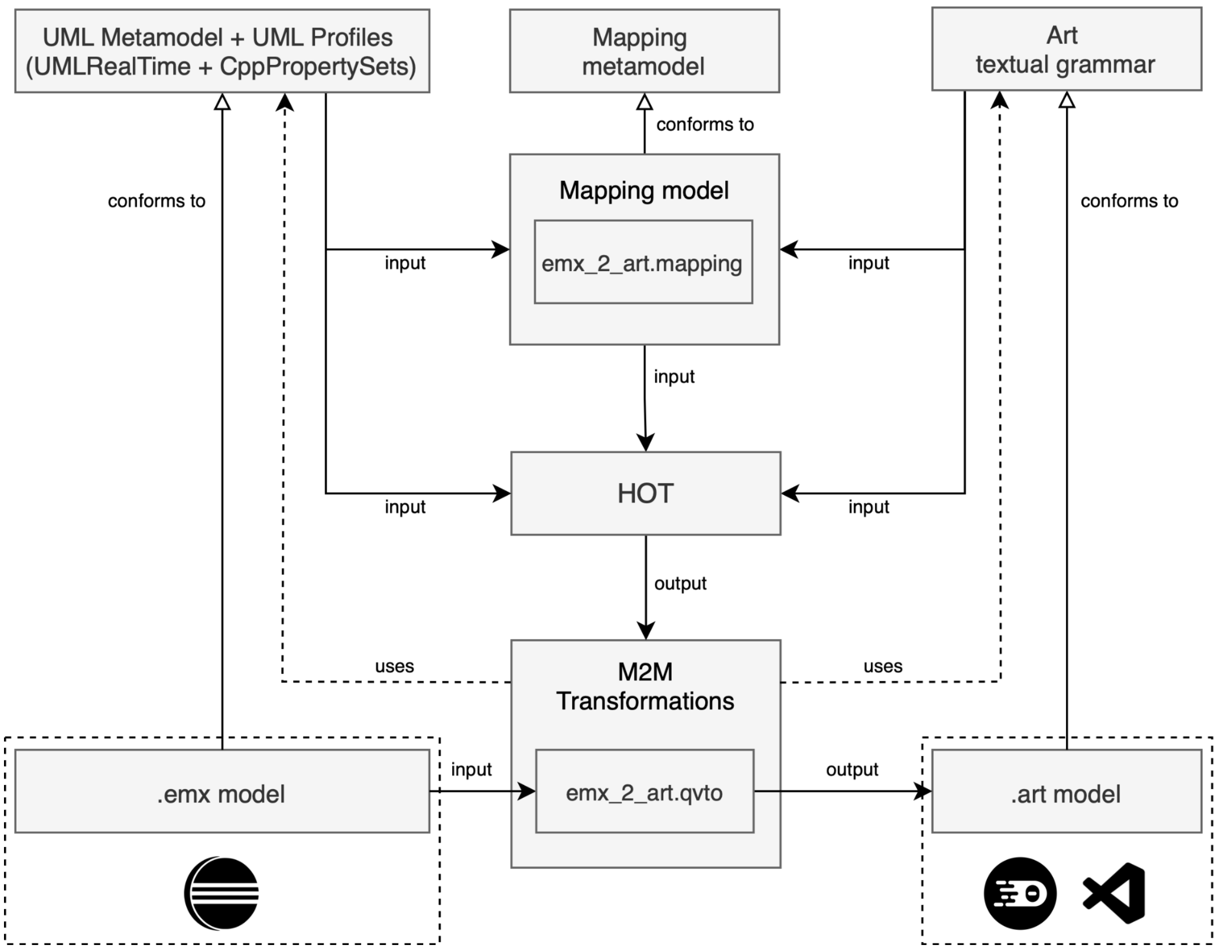


Figure 1: Model exporting approach workflow

Starting from the UML metamodel and profiles, and the Art textual grammar, we define the emx_2_art mapping model conforming to the mapping metamodel proposed in [LC23]. This model consists of a set of mapping rules between UML and profiles, and Art textual grammar, and is defined only once (unless the UML profiles or the Art textual grammar are modified). The mapping model is then given as input to higher order transformations (HOTs) that use the UML metamodel, UML profiles and Art textual grammar to resolve references to the mapped elements. The HOTs output a unidirectional model to model (M2M) transformation that takes as input .emx models (defined in HCL RTist in the Eclipse IDE) and outputs .art models (that can be used in RTist in Code in Eclipse Theia and VS Code).

Migration of models may also require several preferences to be input by the user. Among these preferences is the separation of the translated model into .art files. Rather than generating a single .art file for the whole model, model migration can be configured to place individual elements in separate .art files. This preference will be supported due to the fact that each client possesses a distinctive modeling style and it is generally preferred for the migrated models to closely resemble hand-written models. The separation of output models into separate files is also helpful in avoiding merge conflicts as there is a reduced likelihood of the same file being edited simultaneously by different contributors. Furthermore, the C++ generator in RTist in Code uses a thread pool to parallelize code generation, where each thread in the pool reads one source .art file and produces a set of .cpp (i.e., implementation) and .h (i.e., header) files. Translation of several small .art files in parallel on a machine with multiple cores is typically faster than translation of a few larger ones. Currently, we are investigating the full set of user preferences that should be supported. A settings and preferences menu will provide users with the ability to configure the model exporter in accordance with their needs.

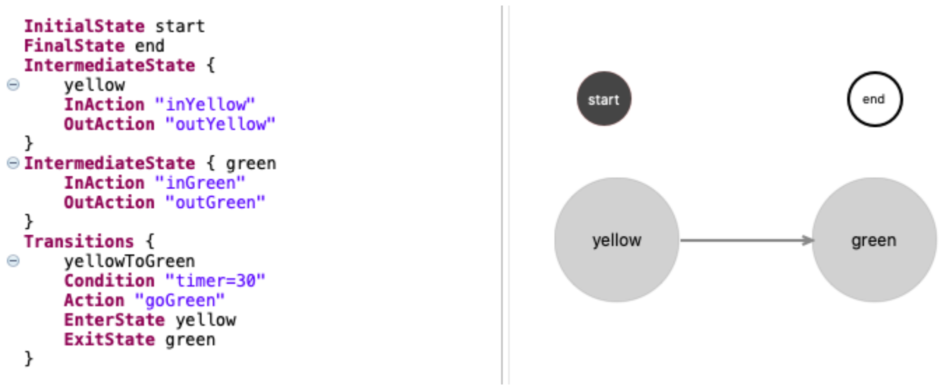
4. Integration of Multiple Open-source Frameworks for Metamodeling and Modeling

In this section, we summarize the integration of disjoint tools for metamodeling and modeling purposes. Note that the technical details of each integration are not provided here, where we instead focus on the benefits of the integrations. Technical solutions and descriptions can be found in other deliverables for WP3 and WP4.

4.1 Integration of Sirius and Xtext

Sirius and Xtext are both available under the Eclipse umbrella, but there is no ready-to-use out-of-the-box mechanism to seamlessly connect them. In BUMBLE, we have exploited model transformation techniques to link Xtext textual notations to Sirius graphical notations via annotated EMF metamodels. More specifically, we allow the generation of graphical languages from annotated EMF metamodels that are in turn generated from Xtext grammars (and vice versa).

The integration of these two frameworks allows the end-user to switch between graphical and textual editing of the same models, as depicted in Figure 4.1.1, and it is the ground for the generation of blended textual and graphical modeling environments in the Eclipse ecosystem.



```
InitialState start
FinalState end
IntermediateState {
    yellow
    InAction "inYellow"
    OutAction "outYellow"
}
IntermediateState { green
    InAction "inGreen"
    OutAction "outGreen"
}
Transitions {
    yellowToGreen
    Condition "timer=30"
    Action "goGreen"
    EnterState yellow
    ExitState green
}
```

Figure 4.1.1. Blended notations with Xtext and Sirius

4.2 Integration of EMF and MPS

EMF and MPS are the two disjoint core modeling environments addressed by BUMBLE. There is no connection out-of-the-box between the two. In BUMBLE, we have provided the pillars for bridging these two powerful (meta-)modeling environments. Via EMF, the end-user can define tree-based and graphical languages, as well as models conforming to them, and via MPS, she can describe textual languages and exploit the power of projectional manipulation of textual models via multiple views [AC22]. The possibility to visualise and edit the same information in these two platforms, otherwise disjoint, can greatly boost communication between stakeholders, who can freely select their preferred notation or switch from one to the other at any time. The framework's architecture combining EMF and

MPS is depicted in Figure 4.2.1. More details are provided in other deliverables related to multiple editors.

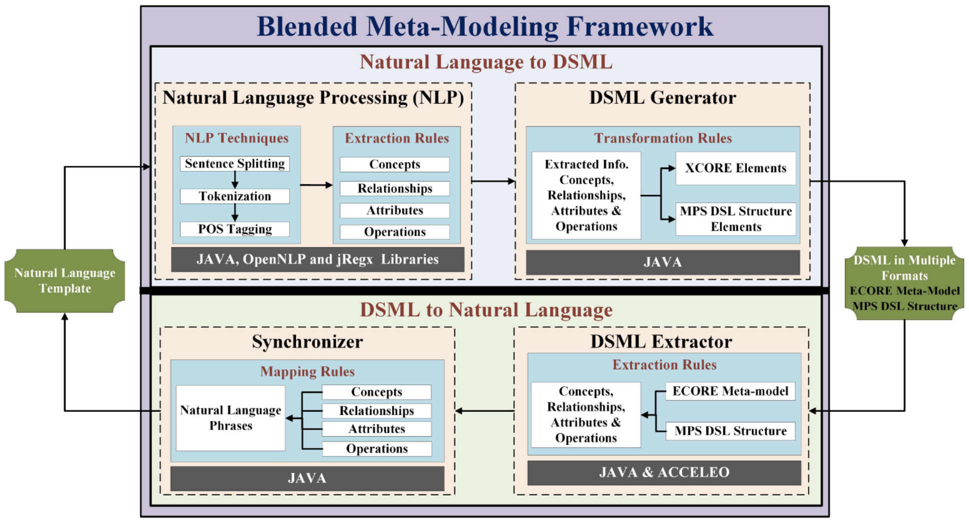


Figure 4.2.1. Framework architecture

4.3 Integration of Draw.io and TextX

Draw.io (soon to become Diagrams.net) is a standalone web-based environment for cross-platform graph drawing software developed in HTML5 and JavaScript. Its interface can be used to create informal diagrams such as flowcharts, wireframes, organizational charts, and network diagrams.

TextX is a standalone Python-based framework for the definition of Python-based grammars; TextX is based on Xtext but does not rely on the Eclipse ecosystem.
Draw.io is very useful and used for early design and communication of software systems via informal diagrams. Due to their notational freedom and effectiveness for communication, informal diagrams are often preferred over models with a fixed syntax and semantics as defined by a modeling language. However, precisely because of this lack of established semantics, informal diagrams are of limited use in later development stages for analysis tasks such as consistency checking or change impact analysis. In BUMBLE, we have provided an approach to reconciling informal diagramming and modeling such that architects can benefit from analysis based on the informal diagrams they are already creating [JC22] Our approach supports migrating from existing informal architecture diagrams in Draw.io to flexible and textual models in TextX, i.e., partially treating diagrams as models while maintaining the freedom of free-form drawing. Moreover, to enhance the ease of interacting with the flexible models, we provide support for their blended textual and graphical editing. The workflow is depicted in Figure 4.3.1.

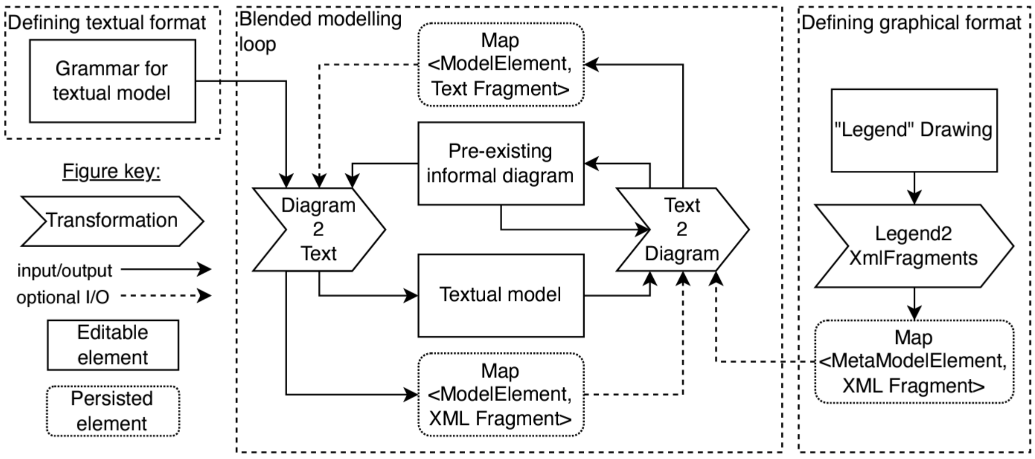


Figure 4.3.1. Workflow for transitioning from Draw.io drawings and TextX models and back

5. Integration of JSONware and Modelware via JSONSchemaDSL

JSONSchemaDSL is a model-driven approach to bridge two unrelated technical spaces (TS), namely JSONware and Modelware.

The JSONware TS includes JSON (JavaScript Object Notation) and JSON Schema. Modelware TS refers to software tools and platforms that are used to develop, maintain, and manage models as cornerstone artifacts of a (software) engineering process. In particular, JSONSchemaDSL leverages the software tools and platforms based on Eclipse Modeling Framework (EMF), like Xtext, Sirius, and GEMOC (see Figure 5.1).

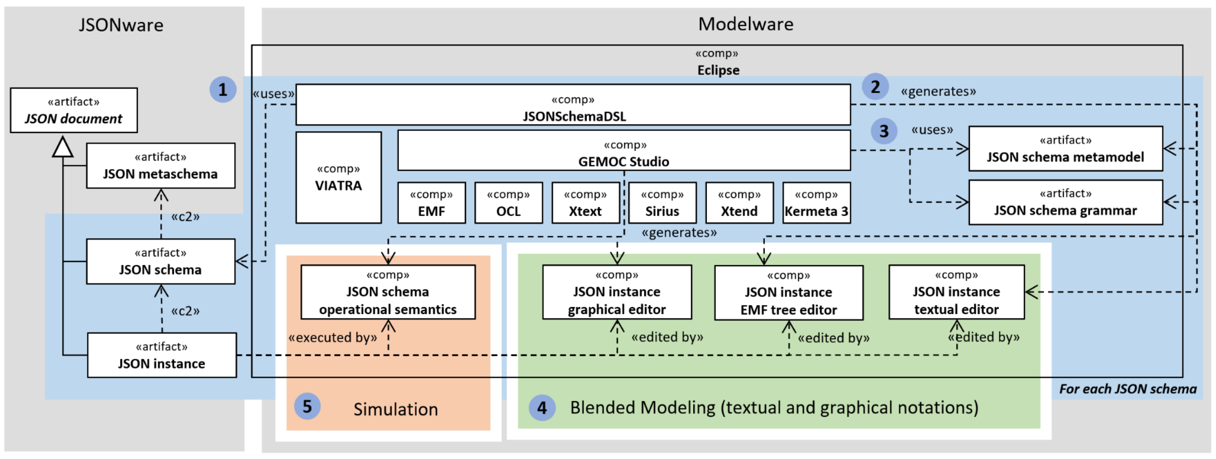


Figure 5.1: JSONSchemaDSL components

JSON schema is taken as input by JSONSchemaDSL (1), which generates (2), via a semi-automated process, the corresponding Ecore metamodel, and Xtext grammar. Thanks to the native capabilities of EMF and Xtext, tree-based and textual editors for JSON instance documents are automatically generated, and enriched by JSON schema-specific OCL constraints. The textual editor adopts the native JSON notation and is meant to be used by domain experts already acquainted with the JSON notation for their particular engineering activity. In addition, MDE experts can also further inspect or edit the actual structure of the in-memory representation of the JSON artifact in EMF using the generated tree-based editor.

A graphical concrete syntax for the given JSON schema is realized via Sirius, an Eclipse project which allows the generation of a graphical modeling workbench for EMF-based models.
Language engineers and domain experts are expected to agree on the graphical notation. A graphical editor is then generated for JSON instance documents. The Sirius-based editor offers the possibility to visualize and edit the content of a JSON instance document.

It is worth noting that all the editors mentioned above, i.e., tree-based, textual, and graphical, share the same in-memory representation based on EMF and can be opened in the same Eclipse IDE.
This capability is necessary for a blended modeling experience of JSON artifacts (4). Indeed, domain experts can seamlessly choose among three different editors to manipulate the same JSON Instance document, which, in turn, conforms both to the original JSON schema and the generated Ecore metamodel and Xtext grammar. Moreover, by preserving the native JSON notation, the domain experts can continue using any existing JSON document editor.

Finally, if a JSON schema is provided with executable operational semantics, any compliant JSON instance documents can be executed (5).
Executable operational semantics are domain-specific, i.e., different operational semantics are expected to be specific for a given JSON schema in order to make executable any valid instances of that schema. We did it for the Shipyard workflow language provided by Keptn tool[7] using the GEMOC Studio. GEMOC reuses metamodels/grammars (3) and provides generic components through Eclipse technologies developing, integrating, and using heterogeneous xDSLs via a language workbench. In particular, the GEMOC Studio integrates Kermeta 3, an action language used to implement the execution semantics of Ecore metamodels, as the ones generated by the JSONSchemaDSL.
Kermeta 3 is built on top of the Xtend, a dialect of Java, which compiles into readable Java-compatible source code. As a result, a Java-compliant interpreter can be generated for a given JSON schema.
The availability of executable semantics and interpreters for JSON-based xDSLs may help (i) domain experts in performing activities and (ii) tool providers to augment their tools with MDE technologies whenever JSON schemas are used in model-based DevOps processes.

It is worth noting that the JSONSchemaDSL is not a "one size fits all" approach. It covers domain-independent steps, like representing the JSON metaschema in Ecore and domain-specific ones. The latter must be replicated for each JSON schema.

Details on technical implementations of JSONSchemaDSL can be found in related publications [CG21], [CH21] and [CB22].

JSONSchemaDSL has been conceived to be compatible by construction with software tools and platforms used in the JSONware and EMF TSs.

The integration with tools and platforms can be faced from two main user viewpoints.

- **End-User viewpoint/JSON expert/MDE-agnostic**. This user can seamlessly use JSONware and Modelware tools. The Xtext-based editors generated by the JSONSchemaDSL approach reuse the native JSON notation. As such, a JSON document (e.g., a JSON schema or

[7] https://keptn.sh/

schema instance) can be copy-pasted from/to JSON editors (e.g., Visual Studio Code) and the Xtext-based editors.

- **Tool Experts.** A bird's eye view of the technical realization is given in Fig. X. In JSONware TS, JSON documents can be manipulated through a variety of existing tools for JSON (e.g., GSON[8]) and JSON Schema-specific ones[9], e.g., validators. JSONSchemaDSL is implemented on top of Eclipse-based technologies and provides an EMF-based Java API to manipulate a JSON document programmatically as a EMF-model conforming to a generated schema metamodel. From an MDE expert’s perspective, JSON documents can now be managed as EMF-based ones, allowing the reuse of model management platforms like Eclipse Epsilon to implement model-based functionalities and integrate external tools, with a preference for Java-based tools.

6. Modelix and MPS

The Modelix platform uses several general purpose frameworks to achieve a cloud-based, multi-user modeling environment. In this document, we only give a high-level description of these integrations. For some aspects, more details can be found in the WP3 and WP5 deliverables.

6.1 Modelix Deployment, Authorization, and User Management

Docker, helm, and Kubernetes are used to automate the creation of a Modelix platform. Once a Modelix platform has been instantiated, the Kubernetes cluster automatically deploys new pods with MPS and Projector when a user opens a workspace. A caching mechanism has been created so that a user who does not have an assigned pod will receive an unbound pod. This means the user does not have to wait long for a new MPS instance to be started, at the cost of some resource overhead.

Authentication is currently managed by a Keycloak instance, which is configured through a reverse proxy to add the user profile to the request for the MPS pods, or redirect the user through a log-in page. The log-in page can be connected to several authentication end-points, such as ADFS, OAuth, or a log-in page that allows Keycloak itself to verify the user’s account.

6.2 Version & Dependency Management

Modelix currently supports the creation of workspaces, which determine the version of MPS that is used, along with the location of the language and model dependencies that need to be loaded in the collaborative environment. These languages and model dependencies are automatically downloaded and built (if necessary) from different locations:

- (Maven) artifact repository
- Git repository (by specifying a git tag, HEAD, or commit ID)
- Workspace uploads (ZIP archives)

This mechanism allows for specifying which language plugin version is to be used in a particular workspace, similar to setting up a development/modeling environment on a local machine which has MPS installed. A next step includes exporting the state of the modeling environment back to a git

[8] google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back (github.com) https://github.com/google/gson
[9] Here a set of tools to for JSON schema-related activities https://json-schema.org/implementations.html

branch. This would allow traditional DevOps and Quality Gates to be connected to the collaborative environment.

The modeling infrastructure requires the specific versions to be specified. At the time of writing, MPS only partially supports defining the version of a language/plugin. This means it does not have strong support yet for checking backward compatibility of modeling languages. To circumvent this shortcoming, one either should allow that only strict dependency versions are used, or that no strong guarantees are given about the compatibility of language/plugin versions. Special care needs to be taken with changing API's and data structures, as this may eventually lead to corruption of the modeling environment.

6.3 Integration with other Front End Technologies

Modelix can be used as a centralized model server with the MPS Projector front-end, or through an API to query model information. This has been further described in deliverable D3.5.

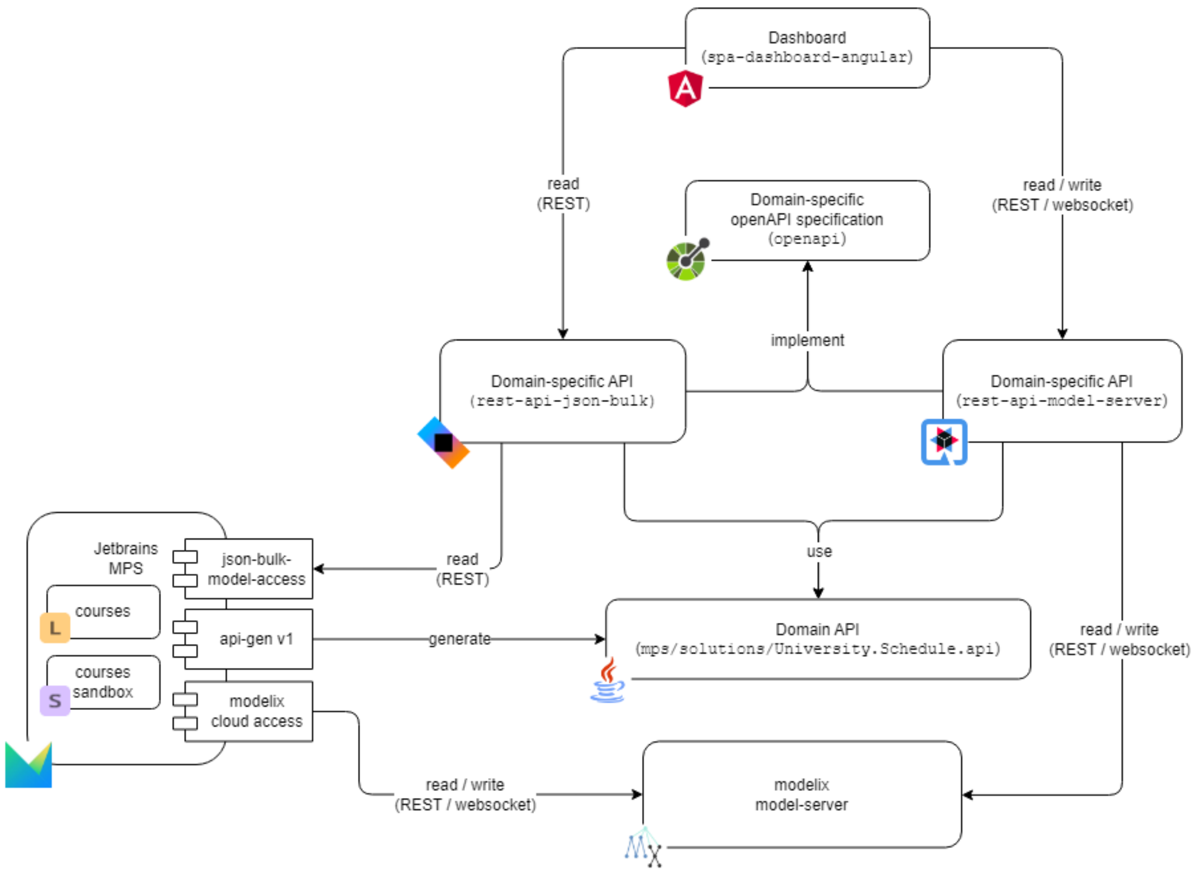


Figure 6.3.1: Modelix System Architecture from https://github.com/modelix/modelix-samples

The architecture shown in Figure 6.3.1 explains how in addition to the general Modelix Model API, different API's can be used for different front end technologies. An example Single Page Application using React and an OpenAPI service specification for an MPS language allow querying from and submitting model data to the model server of Modelix.

Some deployment options are shown in Figure 6.3.2. In addition to the API approaches described above, it is possible to deploy MPS Server with Web Edit Kit to achieve a similar interaction. This approach is more applicable to linking top-level specifications together, whereas the exposing of the MPS DSL editor is more applicable to special-purpose, powerful editors with a rich syntax.This leads to highly tailored, specific purpose web wizards, hiding the more complex interactions with DSML editors.

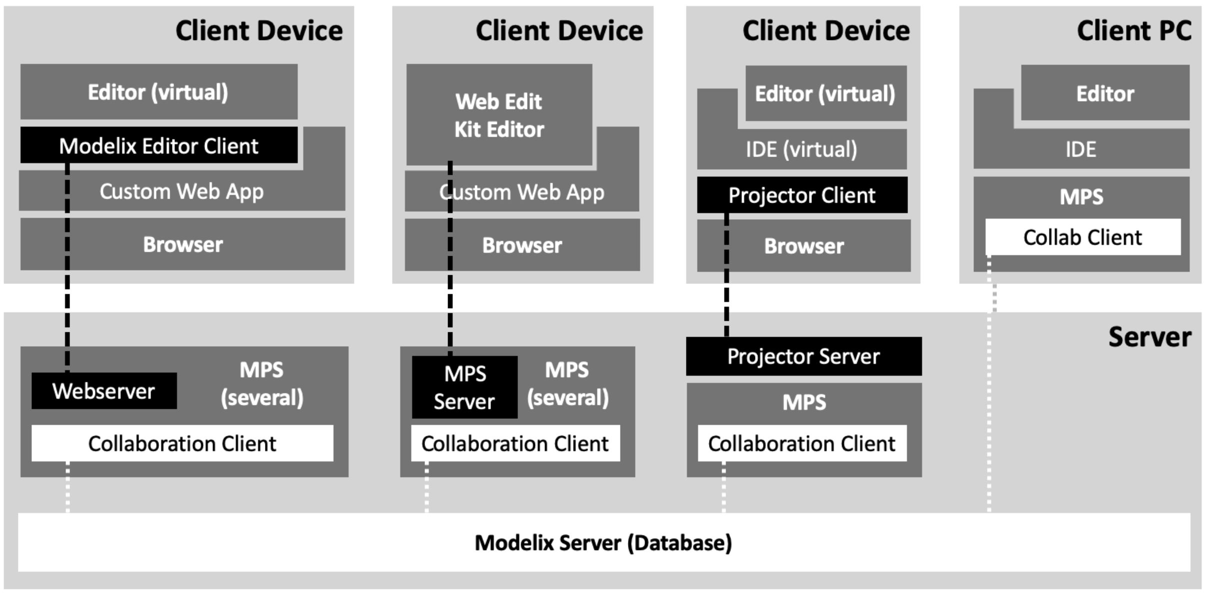


Figure 6.3.2: Deploying MPS with MPSServer and Web Edit Kit editor.

6.4 Integration with DClare

Modelix provides a centralized modeling environment. As plugins can be installed in the workspace, it is also possible to deploy a DClare client, which allows connecting to a decentralized modeling environment. This approach is being explored in the BUMBLE Dutch Demonstrator.

7. SuperModels + MPS

In UC7, we created a blended modeling environment (ME) that combines the strengths of SuperModels and MPS. It's tentatively called SuperME. That allows us to keep our investments in SuperModels and at the same time tap into the established ecosystem of the more mature MPS ME.

SuperModels and MPS are deeply integrated, not meerly mechanically combined. As depicted in Figure 7.1 below, SuperModels serves as frontend and MPS serves primarily as backend assuming the responsibility of model storage (persistence). DSML users can employ all aspects of existing SuperModels DSMLs like diagrammatic editors, model checks and generators. At the same time, they can employ all aspects of MPS DSMLs like editors, model checks and generators. We selected JetBrains RD as an interfacing technology that allows immediate inter-process communication between MPS (JVM process) and SuperModels (.NET process).

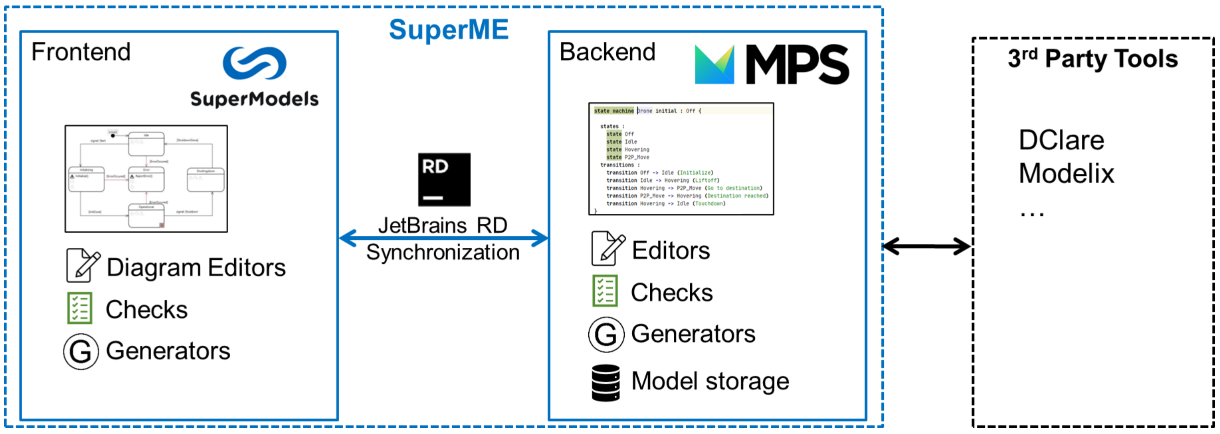


Figure 7.1: Overview of SuperModels and MPS integration

7.1 Generic Part - RService

The SW stack used to implement SuperME has a generic part and a SuperModels-specific part. Let's first look at the generic part that we called RService (see Figure 7.1.1).

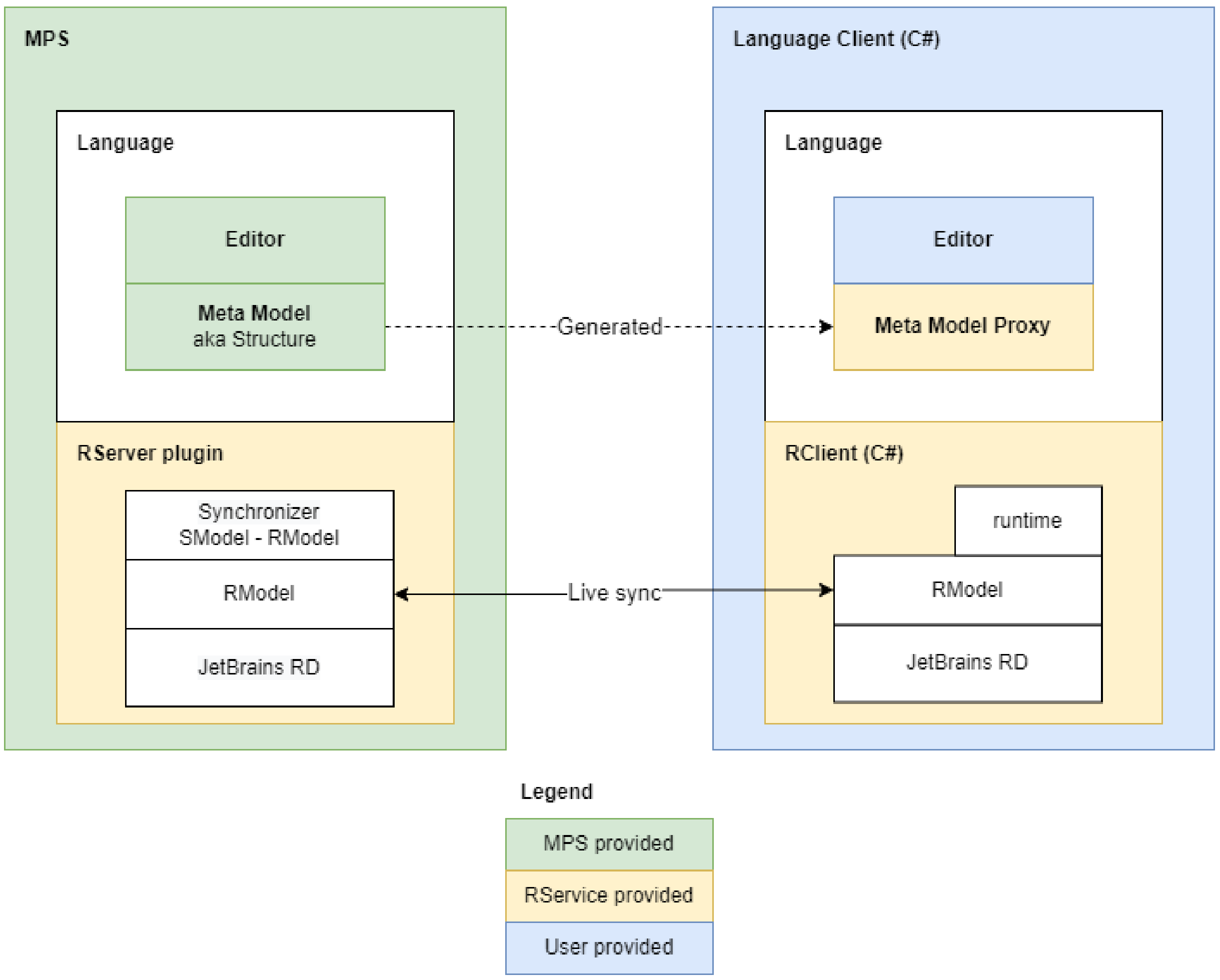


Figure 7.1.1: SuperME SW stack - generic part

RService is built with the JetBrains Reactive Distributed (RD) communication framework and thus employs a client-server architecture. RD allows a server and a client to communicate via remote procedure calls (RPCs) but also via shared data. In essence, the server and the client keep a copy of the shared data, and RD takes care to synchronize them. If either side introduces changes, RD will reactively propagate only those changes to the other side. RD allows for either side to subscribe to changes that it cares about. RD communication interface is specified in a DSL in terms of RPCs and/or structure of the shared data.

In our case, we want the server to provide access to MPS models of any MPS language. So we chose to define the communication interface on the abstract model level, meaning that the shared data is in the form of ASTs. We decided to keep our communication interface as close as possible to the MPS' SModel API since it's a familiar API in the MPS community. Hence we called it RModel. Consequently, we called our model server RServer and our model client RClient.

RServer runs from within MPS and is packaged as an MPS plugin. It provides access to MPS models via the RModel interface by synchronising MPS models to and from its local copy of the RModel data structure.

RClient is a code generator and runtime library that allows to export MPS metamodels into C# classes that provide a type-safe wrapper around the RModel API.

7.2 SuperModels-specific Part

Let's now see how the generic RService is applied in the integration of MPS and SuperModels (see Figure 7.2.1).

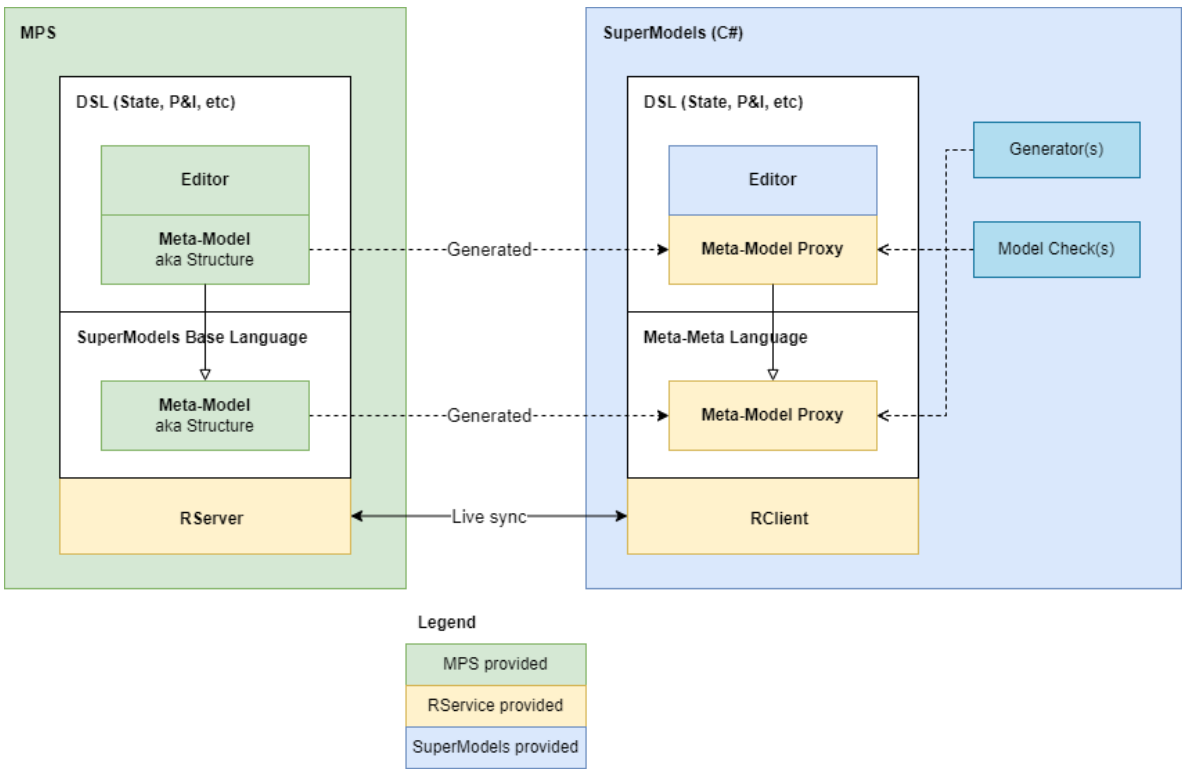


Figure 7.2.1: SuperME SW stack - SuperModels specific part

As mentioned above, we decided to allocate the model storage responsibility (also called persistence) to MPS. This means that models expressed in SuperModels DSMLs must be hosted on MPS and accessed via RServer. This implies that by necessity the structure aspect of SuperModels DSMLs should be implemented in MPS. We implemented the SuperModels meta meta model in MPS as SuperModels Base Language (only the structure aspect). Then, any SuperModels DSML in MPS is an extension of SuperModels Base Language.

This architecture allows for SuperModels editors to show and modify the AST stored in MPS and thus to behave similar to 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 without the need to change them. Said differently RService is applied to synchronise SuperModels specific ASTs between model server in MPS and model client in SuperModels.

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 within and across DSMLs. So we can link between models from both SuperModels and MPS.

8. Summary

During the BUMBLE project, a vast set of (meta-)modeling tools have been integrated into different tooling environments for a variety of purposes. Theses purposes encompass enabling certain features like blended modeling or real-time collaboration, allowing a DSML user to flexibly change the tooling environment within a heterogeneous tool chain, migrating modeling to web and cloud environments, and automating error-prone and time-consuming work on transferring model information between different tooling environments. In the deliverable at hand, we sketched these integrations and referred for further, more detailed information to other BUMBLE deliverables.

References

[LC23] Latifaj, M., Ciccozzi, F., & Mohlin, M. Higher-Order Transformations for the Generation of Synchronization Infrastructures in Blended Modeling. Frontiers in Computer Science, 4, 166.

[AC22] Anwar, M. W., & Ciccozzi, F. (2022, April). Blended Metamodeling for Seamless Development of Domain-Specific Modeling Languages across Multiple Workbenches. In *2022 IEEE International Systems Conference (SysCon)* (pp. 1-7). IEEE.

[JC22] Jongeling, R., Ciccozzi, F., Cicchetti, A., & Carlson, J. (2022). From Informal Architecture Diagrams to Flexible Blended Models. In *European Conference on Software Architecture* (pp. 143-158). Springer, Cham.

[CG21] A. Colantoni, A. Garmendia, L. Berardinelli, M. Wimmer, and J. Bräuer, "Leveraging Model-Driven Technologies for JSON Artefacts: The Shipyard Case Study," 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2021, pp. 250-260, doi: 10.1109/MODELS50736.2021.00033.

[CH21] A. Colantoni, B. Horváth, Á. Horváth, L. Berardinelli, and J. Bräuer, "Towards Continuous Consistency Checking of DevOps Artefacts," 2021 ACM/IEEE International Conference on Model

Driven Engineering Languages and Systems Companion (MODELS-C), 2021, pp. 449-453, doi: 10.1109/MODELS-C53483.2021.00069.

[CB22] A. Colantoni, L. Berardinelli, A. Garmendia, and M. Wimmer, "Towards Blended Modeling and Simulation of DevOps Processes: The Keptn Case Study. In ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3550356.3561597