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

D3.1 Architecture Description for BUMBLE Eclipse Platforms—Version 2

BUMBLE

Blended Modeling for Enhanced Software and Systems Engineering

Contributors

Jörg Holtmann	University of Gothenburg
Jan-Philipp Steghöfer	University of Gothenburg
Weixing Zhang	University of Gothenburg
Mattias Mohlin	HCL
Malvina Latifaj	Mälardalen University
Robbert Jongeling	Mälardalen University
Johan Fredriksson	Saab

Reviewers

Ivano Malavolta	Vrije Universiteit Amsterdam
Federico Ciccozzi	Mälardalen University

Project Acronyms

<ACR>	<Acronyms>
BUMBLE	Blended Modeling for Enhanced Software and Systems Engineering
MDE	Model-Driven Engineering
DSML	Domain-Specific Modeling Language
UML	Unified Modeling Language
EMF	Eclipse Modeling Framework
UML-RT	UML for Real-time
rtUML	Real-time UML
SysML	Systems Modeling Language
xtUML	Executable UML
CNL	Controlled Natural Language
ALF	Action Language for Foundational UML
XML	eXtensible Markup Language
GMF	Graphical Modeling Framework
SoC	System-On-Chip
NoC	Network-On-Chip
ECU	Engine Control Unit
MPS	Meta Programming System
PapyrusIC	Papyrus Industry Consortium
PapyrusRT	Papyrus for Real-time
OEM	Original Equipment Manufacturer
AST	Abstract Syntax Tree

Table of Contents

1. Introduction

This document describes the architectures of the solutions BUMBLE proposed for the Eclipse technology space. These architectures can be separated into two target platforms: the architecture for the Eclipse Rich Client Platform (RCP) and the architecture for client-/server-based solutions that use the underlying Eclipse technology. Chapter 2 introduces these two more specific target platforms and explains how the BUMBLE use cases and the requirements identified in D2.2 relate to them. Chapter 3 then introduces four use cases in more detail:

- UC1, driven by MDU, is the canonical use case used in BUMBLE to illustrate basic concepts using simple languages for the modeling of state machines;
- UC2, driven by HCL, focuses on combining textual and graphical modeling of sophisticated state machines in HCL RTist;
- UC3, driven by Volvo, addresses blended modeling for vehicular architectures using EAST-ADL; and
- UC6, driven by Saab, aims to allow blended editing and consistency checking of SysML models and related program code.

In comparison to the version 1 of this deliverable, we particularly updated the description of UC2 with an alternative realization based on a client/server architectural style. Furthermore, we updated all sections with the current status quo and added a conclusion.

The table below shows additional use cases that also use the Eclipse technology space. However, in this deliverable version, we focus on the four use cases above as representative examples. All three use cases only focus on the Eclipse technology space and do not also consider JetBrains MPS as underlying DSML core technology. Examples for use cases that consider MPS are provided in D3.2. Furthermore, two of the three examples do not only require solutions that work within the Eclipse RCP, but also require client-server solutions in order to allow blended modeling in editors such as VS Code and Eclipse Theia that rely on language servers as the backing for the editors.

Use Case	Description	Lead Partner	MPS	EMF	Includes client/server aspects
UC1	**Software Open-Source Blended Modeling**	**MDU**	**X**	**X**	
UC2	**Combined Textual and Graphical Modeling of State Machines in HCL RTist**	**HCL**		**X**	**X**
UC3	**Vehicular Architectural Modeling in EAST-ADL**	**Volvo**		**X**	**X**
UC5	Reactive and Incremental Transformations across DSMLs	MVG	X	(X)	

UC6	**Blended Editing and Consistency Checking of SysML Models and Related Program Code**	**Saab**		**X**	
UC10	Development Process of Low-Level Software	Unibap		X	
UC11	Multi-Aspect Modeling for Highly Configurable Automotive Test Beds Ready for Smart Engineering Demands	AVL		X	X

2. The Eclipse Technology Space as Base DSML Technology

Within the Eclipse technology space, a number of common concepts are used as introduced in Section 2.1. The concrete solution based on these common concepts can then either be realized in the Eclipse Rich Client Platform (RCP) as discussed in Section 2.2 or based on a client/server architectural style as discussed in Section 2.3. We also show how the existing technologies support the main BUMBLE requirements in Section 2.4 and which extensions are necessary.

2.1. Architecture Overview

Independent of the architectural style, the architecture for the Eclipse technology space follows a model-view-controller pattern in which a *model source* holds information about the models that are viewed or edited and *graphical and textual editors* provide the controllers and view. Depending on the target platform and the concrete view, there can also be a translation layer between the model and the view and controller parts. This is explained in more detail in the two subsequent sections for each of the architectural styles. This abstract architecture is shown in Figure 1.

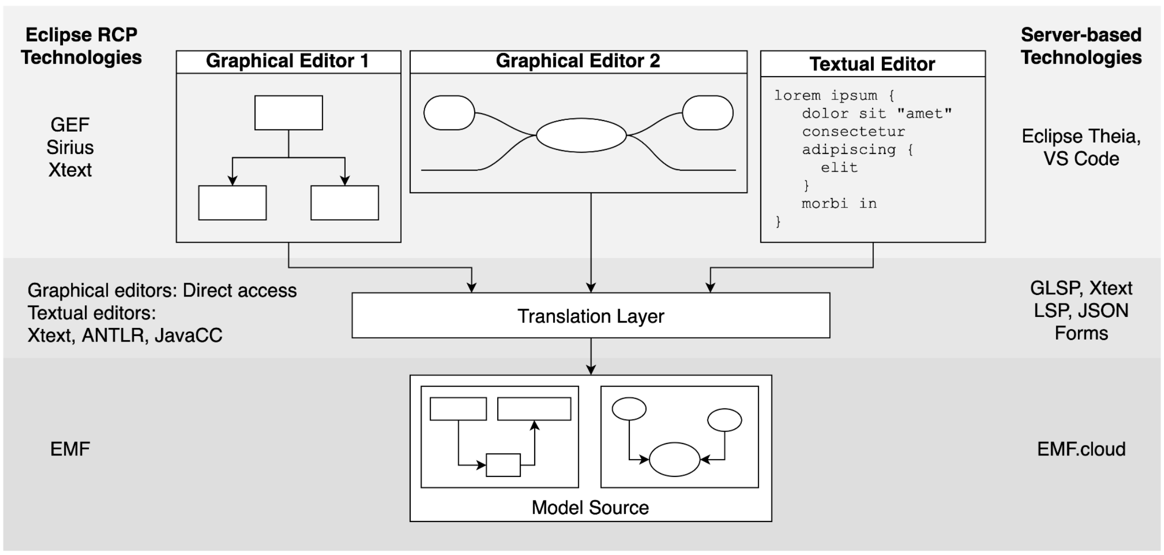


Figure 1. An abstract view of the architecture within the Eclipse technology space with different technologies depending on the chosen architectural style (Eclipse RCP or client/server).

In the Eclipse world, the base technology to represent, query, and modify any model at runtime is the Eclipse Modeling Framework (EMF)[1]. It is based on a highly generic meta-model (Ecore) that can be used to specify domain-specific modeling languages. The Unified Modeling Language (UML) implementation used in Eclipse and in editors like Eclipse Papyrus is, e.g., specified based on Ecore. That means that the M2 meta-model provided by Ecore is used to specify the M1 meta-model for UML which is in turn used to create M0 model instances. An M0 model instance is always represented as an EMF model tree with a clearly defined root note. EMF ensures that a model tree conforms to its M1 meta-model. Therefore, the single source of truth that the EMF model tree represents always conforms to the meta-model.

The EMF model tree is stored in memory. Eclipse offers a resource set and editing domain mechanism that provides scoping for EMF model trees. It is therefore possible to load the same model in different scopes and manipulate it independently. Synchronization happens on save and reload, whereas blended modeling editors should synchronize on any change of the model in any representation. Conversely, if two editors use the same editing domain, they can access a model within the same scope. That means that a change made by one editor can be immediately picked up by the other editor – at least, if the view is updated accordingly. Simply put, if one editor changes the EMF model tree (i.e., the single source of truth), the other editor can "see" these changes immediately. Publish-subscribe is usually used to implement such an update mechanism.

EMF also provides a transaction concept that enables a limited form of concurrent editing. Write accesses to the EMF model tree can be wrapped in a transaction that protects the model tree from concurrent writes. Resolving these conflicts is, however, left to the editors that triggered the change.

It is possible to load multiple M0 model instances based on different M1 meta-models at the same time with EMF. These model instances can then cross-reference each other. EMF ensures the integrity of these references.

In many cases, EMF model trees can be accessed directly, e.g., by graphical editors. The editor can retrieve the EMF model tree of a resource (usually a file) directly from EMF. Depending on the concrete file format, EMF has built-in deserialisation capabilities (e.g., for XMI files). In some cases, however, an additional translation layer is needed. The most prominent example of this are textual editors that work with a specific notation different from XMI. In this case, a parser needs to deserialize the text file into an EMF model tree. A common technology used for this purpose is the language engineering framework Xtext[2]. In the context of client-server solutions, additional translations need to be made to serialize the contents of the EMF model tree for transfer over HTTP.

2.2. Eclipse RCP

The Eclipse Rich Client Platform is the foundation of the Eclipse IDE and other, specialized tool workbenches. It uses a plugin architecture based on the OSGi standard to provide a modular environment in which features can be added dynamically as plugins, also called bundles. Popular modeling tools for the Eclipse RCP include Eclipse Papyrus and Polarsys Capella. These tools are based on more foundational technologies such as EMF mentioned above as well as GMF and Sirius

[1] https://www.eclipse.org/modeling/emf/
[2] https://www.eclipse.org/Xtext/

as technologies that support the implementation of graphical editors. The latter technologies are specific to the Eclipse RCP and tightly coupled with the underlying abstractions for graphical user interfaces.

The advantage of RCP solutions is that the end user receives an application that has all relevant features bundled and can therefore be used out of the box. The developer can create these "products" based on any set of compatible features. Eclipse provides a system called P2 that ensures that the installed features and all prerequisites work as expected.

2.3. Client/Server Architectural Styles based on Eclipse Technologies

The Eclipse ecosystem has also brought forward a number of technologies that are independent of Eclipse RCP or have, over time, found a new life outside of the confines of the rich client platform. In particular, Eclipse Theia is a web-based IDE, similar to VS Code, that can run in the browser. Like VS Code, Eclipse Theia uses language servers as the foundation of its editors. There are also solutions relying on language servers that enable using EMF outside of the Eclipse IDE, for example, in VS Code or Eclipse Theia.

Importantly, however, technologies like EMF.cloud[3] allow the use of Eclipse modeling technologies in such an environment. This increases interoperability with existing modeling environments, provides a transition path for organizations that want to evolve their modeling environments from Eclipse RCP applications, and makes the powerful tools that are already available in the Eclipse ecosystem available to client/server solutions. Note, however, that the transitioning from the Eclipse RCP to a language server based architecture is even smoother if the language server is implemented in Java, since in that case developers can use EMF as is.

In terms of the architecture in Figure 1, the client/server architectural style requires a more involved translation layer. That is, while all editors in the RCP can share an in-memory representation of the viewed or edited model and access it via the EMF API, a client/server solution requires translating the model into a representation that can be transferred over the network in both directions. The Language Server Protocol (LSP) has been designed to facilitate this exchange of information. Clients and servers implement it to exchange information about a model and enable features such as syntax highlighting and code completion. The Eclipse Graphical Language Server Platform (GLSP)[4] extends this mechanism to also accommodate graphical editors on the client side. Some viewers or editors also use JSON as an exchange format, e.g., the tree view in VS Code.

2.4. Existing Support with respect to BUMBLE Features

In the following, we briefly describe the current capabilities of the Eclipse Technology Space for the five high-level BUMBLE features identified in Deliverable D2.2. As appropriate, we will reference the core requirements as well as the relevant use cases. Furthermore, we exemplarily illustrate some of the current capabilities throughout the following subsections by referencing Figure 2. An overview of the current capabilities is provided in Table 1. The BUMBLE Core Requirements (BCx) w.r.t. the BUMBLE features are specified in D2.2 as follows:

[3] https://www.eclipse.org/emfcloud/
[4] https://www.eclipse.org/glsp/

- BC1: It must be possible to define multiple concrete syntaxes / representations for a single DSML model definition, including relevant views or editors conforming to the concrete syntaxes / representations.
- BC2: A DSML user must be able to select a preferred concrete syntax / representation for a DSML model instance. A DSML developer must define a default concrete syntax / representation.
- BC3: In case multiple syntaxes exist for a (single element of a) DSML model definition, all concrete syntaxes / representations must be updated in accordance with any changes that have been performed by means of using one of those syntaxes.
- BC4: In case multiple syntaxes exist for a (single element of a) DSML model definition, it must be possible that certain elements may not be relevant or visible in one or more specific abstract and concrete syntaxes. Semantics of (an element of) a DSML model definition that is considered in multiple abstract and concrete syntaxes must (be enforced to) be/remain the same.

Table 1. Overview of the support for the five main BUMBLE features in the Eclipse Technology Space.

BUMBLE Feature	**Out-of-the-Box Support in the Eclipse Technology Space**
Blended Syntaxes & Modeling (B)	The Eclipse RCP partially supports BC1 and BC2 with its facilities to generate editors for DSMLs, but there is no existing technology that directly supports BC3 and BC4 without further development since changes in one editor are not synchronized continuously to another. The client/server architectural style does not provide editor generators or other blended modeling capabilities and thus does not fulfill any of the requirements.
Collaborative Modeling (C)	The Eclipse RCP is single-user oriented, but provides some extensions to support certain forms of collaboration. While there are projects that provide support for collaborative programming (e.g., the Saros project[5]) no such support currently exists for any of the common model editors. Eclipse Connected Data Objects (CDO)[6] supports persisting changes to a database and propagating these changes to different clients, but is implemented on the level of the abstract syntax (EMF). Even the client/server architectural style does not support such collaboration out of the box. While the underlying technologies are in theory capable of multi-user support, they only support single user access at this point[7].
Evolution (E)	Evolution of the meta-model is partially supported by a number of model transformation tools that allow automatic transformation of models to accommodate changes in the meta-model.

[5] https://www.saros-project.org/
[6] https://www.eclipse.org/cdo/
[7] https://eclipsesource.com/blogs/2021/02/25/the-emf-cloud-model-server/

Traceability (T)	Traceability across models is supported out of the box if such references are defined in the meta-models. For the Eclipse RCP, more capable solutions are available in fulfillment of BC6, but the client/server architectural style lacks capable traceability tools.
Model Non-Conformance (N)	Non-conformance is relevant for textual editing of models. Since all technologies for textual editing of models in the Eclipse technology space are parser-based, non-conformance needs to be resolved manually by the user.

2.4.1. Blended Syntaxes & Modeling (B)

As demonstrated in UC1, Eclipse RCP technologies provide some support for blended modeling out-of-the-box. Apart from the graphical editor presented in UC1, Eclipse also has different views and editors that access the same underlying model, e.g., the outline view and the properties editor. In many cases, changes in these views and editors are directly reflected in the model and vice versa. For instance, the properties editor allows changing attribute values in a UML class in Papyrus. The outline view allows calling refactoring commands for methods in a Java file. Any change that is made is reflected in the model. On the other hand, updates in the model are also visible in these views, e.g., when the code structure changes, the outline view changes accordingly. Listeners allow propagating these changes, as illustrated in the top of Figure 2. In addition, there is powerful support for defining modeling syntaxes and generating editors for them. For example, Figure 2 illustrates the definition of both a textual and graphical concrete syntax specification based on one metamodel as well as their instantiation based on one model. Together, these capabilities partially address BC1 and BC2. Considering the client-/server architectural style, even though language servers and other technologies exist, they are less mature than their counterparts on the Eclipse RCP side and, e.g., do not yet support generating graphical editors for DSMLs. For this architectural style, BC1 and BC2 are thus not fulfilled.

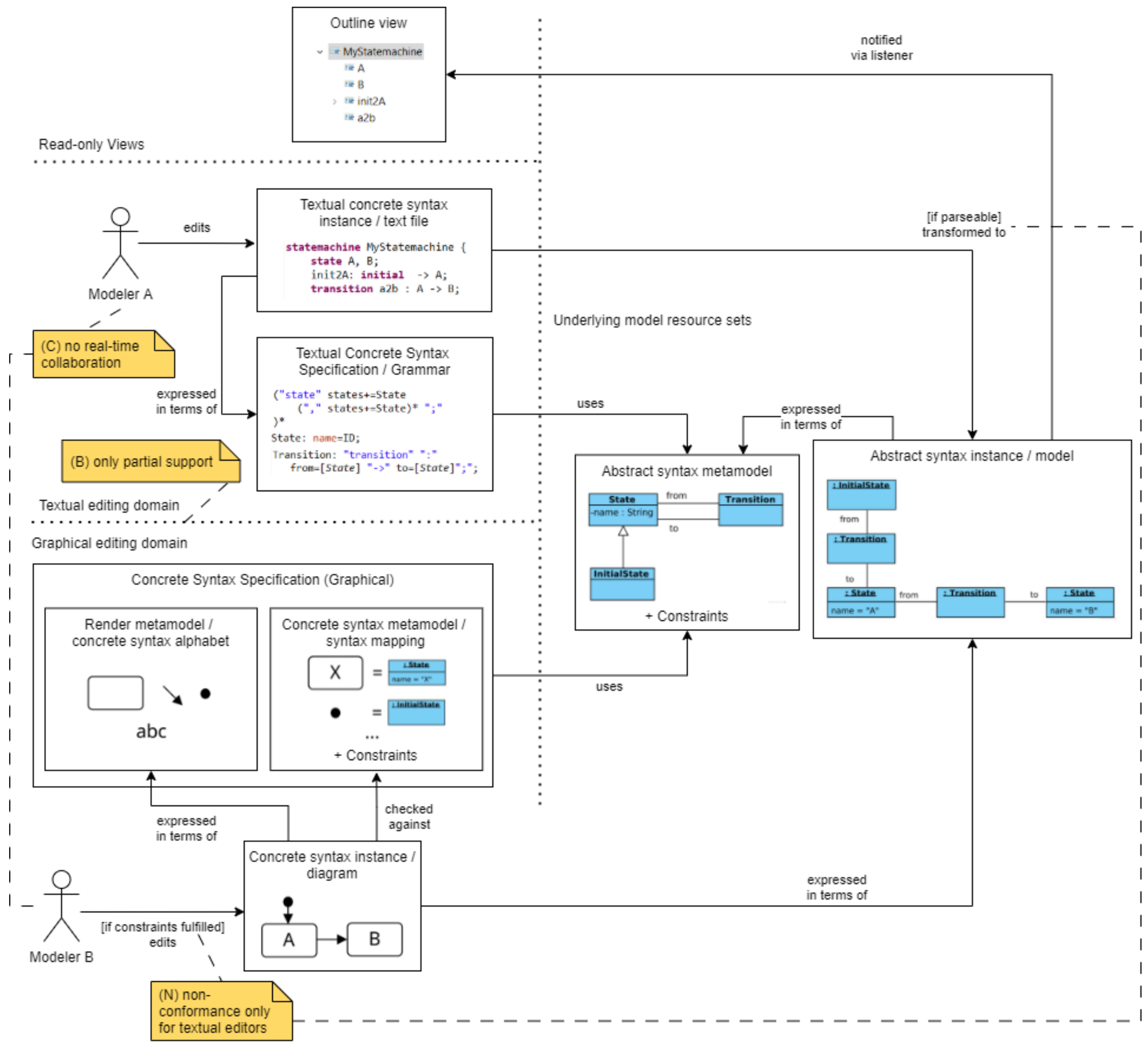


Figure 2. Functional principles of the Eclipse RCP architectural style regarding the BUMBLE features (B), (C), and (N) [figure partially based on Nachreiner et al.[8]]

Within the Eclipse RCP, different views can access the same underlying model (e.g., an editor and the outline view, cf. Figure 2), but this is not blended modeling where different concrete syntaxes are involved. The same is true for the client/server architectural style. The editor presented in UC1 synchronizes the views on save, but not continuously in the spirit of BC3 and BC4. Likewise, there are currently no existing blended solutions for the client/server architectural style, even though the foundational technologies exist. Thus, BC3 and BC4 are not fulfilled.

[8] L. Nachreiner; A. Raschke; M. Stegmaier; M. Tichy: CouchEdit: A Relaxed Conformance Editing Approach. In 2nd Modelling Language Engineering and Execution Workshop, 2020

2.4.2. Collaborative Modeling (C)

The BUMBLE core requirement BC5 addressing this feature states: “It should be possible to support real-time collaboration between multiple DSML users. This means that - independent of which concrete syntax the DSML users have chosen - changes by an individual DSML user are instantly visible to all other DSML users that have viewing/reading and/or editing/writing rights to the considered (collection of) DSML model instance(s).” (cf. Deliverable D2.2)

The built-in capabilities of the technologies for Eclipse RCP do not provide collaborative modeling capabilities that go beyond collaboration via version control. For example, consider Modeler A and Modeler B in Figure 2 editing in two different editors: Despite the figure indicating a synchronization via the model, the editors of the Eclipse RCP are not intended to provide such collaborative real-time modeling. While the SAROS project[9] offers a collaborative code editor, it is limited to text files. The CDO Model Repository does offer multi-user access to EMF models along with change propagation and transactional access. However, CDO operates on the EMF level (i.e., on the abstract syntax level) and therefore needs to be connected to a translation layer as illustrated in Figure 1 to support blended modeling. BC5 is therefore partially addressed in the Eclipse RCP.

In principle, client/server solutions support accessing the same model by different clients. However, in practice, current solutions in the Eclipse technology space are not capable of dealing with concurrent modifications of the same model by different clients out-of-the-box since they are missing the necessary conflict resolution mechanisms. Therefore, BC5 is only partially fulfilled for the client/server architectural style.

An overview of existing technologies for collaborative modeling and their capabilities and shortcomings as relevant for the BUMBLE project can be found in D5.1.

2.4.3. Evolution (E)

The BUMBLE core requirement BC9 addressing this feature states: “It should be possible to deploy a new version of a DSML model definition by means of automatically migrating existing instances of that DSML model definition. In conjunction with that, cross-references to other DSML model definitions and instances must be migrated automatically.” (cf. Deliverable D2.2)

There are a number of model transformation technologies that can be used to automatically migrate models to new versions of a meta-model (e.g., Eclipse QVTO, Eclipse Henshin). If the meta-model evolution is itself expressed as a model transformation, it is even possible to derive the transformations of the models automatically from these higher-order transformations. While EMF as the underlying technology for both architectural styles in the Eclipse technology space does not support model non-conformance out of the box, these transformation capabilities can still partially address the requirement expressed in BC9.

2.4.4. Traceability (T)

The BUMBLE core requirement BC6 addressing this feature states: “It should not be impossible to integrate BUMBLE-based DSML environments in larger (non DSML technology based) applications that enable (real-time or non real-time) collaboration between users of that larger application context.” (cf. Deliverable D2.2)

[9] https://www.saros-project.org/

In terms of traceability, EMF as the underlying technology supports references between models it manages. This applies to both the Eclipse RCP as well as the client/server architectural styles. EMF can store and resolve references across models and automatically loads referenced models when necessary. Changes in the models that affect the cross-referenced models are automatically resolved. However, support for this kind of traceability is dependent on the meta-models explicitly defining such cross-references.

If it is necessary to trace across the boundaries of EMF (e.g., from a model to a source code file) or to establish relationships between models whose meta-models do not provide cross-references, additional technologies are necessary. Tools like Yakindu Traceability[10] or Eclipse Capra[11] can be used to establish traceability in the Eclipse RCP. However, such technologies are currently not available for the client/server architectural style.

In terms of traceability across model versions, EMF offers a number of tools for diffing and merging of models in the Eclipse RCP. Notably, EMF Compare[12] provides these capabilities for the Eclipse RCP. However, no such tools are available for the client/server architectural style.

Overall, BC6 is thus partially addressed in the Eclipse RCP architectural style and not addressed in the client/server architectural style.

2.4.5. Model Non-conformance (N)

EMF as the underlying technology for model handling in the Eclipse technology space does not support non-conforming models out of the box. The standard tree-based or graphical editors that are provided by Eclipse and common technologies like Eclipse GMF or Eclipse Sirius do not allow creating non-conformant models either since they limit the users' ability to create model elements to what the meta-model explicitly allows. Sometimes, a concrete syntax specification and a graphical editor even add additional constraints to forbid certain undesired modeling states. In Figure 2, this is reflected in the bottom left-hand part by the guard “[if constraints fulfilled]” on the “edits” flow, where the constraints refer to the constraints of both the abstract syntax metamodel and the concrete syntax metamodel.

This strict conformance to the meta-model is no longer given in text-based editing of models. There, the user can freely enter text. If these textual editors are based on Eclipse Xtext, the underlying parser will fail to construct the abstract syntax tree for the model and therefore not construct a suitable in-memory representation (cf. guard “[if parseable]” of the “transformed to” flow in the upper right-hand side part of Figure 2). Instead, the user will see error messages which need to be resolved manually. If the text needs to be transformed into a valid model before saving (e.g., because the textual representation is only an intermittent format and the model is serialized differently as in UC3), the user will not be able to save their changes before resolving the errors.

This is a limitation of parser-based approaches as further elaborated in Deliverable D3.2. Section 3.1 also discusses this limitation in the context of UC1. We have no current plans to address this limitation for the Eclipse technology space.

[10] https://www.itemis.com/en/yakindu/traceability/
[11] https://eclipse.org/capra
[12] https://www.eclipse.org/emf/compare/

3. BUMBLE Extensions to the Eclipse Technology Space

In the following, we describe the architectures of three use cases:

1. UC1: The Canonical State Machine Use Case
2. UC2: Combined Textual and Graphical Modeling of State Machines in HCL RTist
3. UC3: Vehicular Architectural Modeling in EAST-ADL
4. UC6: Blended Editing and Consistency Checking of SysML Models and Related Program Code

These descriptions show how the Eclipse Technology Space is used and extended within the BUMBLE project. The use cases cover both architectural styles and make use of many common frameworks in the Eclipse ecosystem. For each use case, we also provide a discussion of how they relate to the high-level core requirements.

3.1. Architectural Description of UC1 (Canonical State Machine Use Case)

This use case covers a public show case for the BUMBLE technologies. It provides blended modeling for a canonical state machine DSML. The focus is not on the expressiveness of the state machines, but rather on providing a vendor-neutral common baseline that showcases the possibility to generate at least two model specific notations, one graphical and one textual, and related editors.
The following describes a prototypical implementation of this scenario with unmodified, out-of-the-box Eclipse technologies for the Eclipse RCP architectural style. In particular, this prototype highlights the shortcomings of the existing solutions and justifies the need for additional effort in BUMBLE.

3.1.1. Eclipse Xtext and Eclipse Sirius

Eclipse Xtext and Eclipse Sirius, are two open-source frameworks for the development of textual and graphical model editors respectively. They are both based on the Eclipse Modeling Framework (EMF), which allows for an out-of-the-box synchronization between these two frameworks. In this work, we describe the process of integrating these two frameworks, the capabilities, and drawbacks. Figure 3 provides an overview of how the synchronization takes place.

Both Xtext and Sirius contain a ResourceSet. A ResourceSet is a collection of Resources, where the latter represent an in-memory model of the physical file system (i.e., the text file). An Xtext ResourceSet contains the *Xtext Resource*, which includes a parser, linker and serializer that supports loading the model from the text file (i.e.,parsing) and saving the model in the text file (i.e., serializing). A Sirius ResourceSet contains two resources; a *SemanticResource* which in our case is an *XtextResource*, and a *DiagramResource* that contains all the graphical information such as shape, size, colour, position etc. The diagram elements reference their semantic counterparts and to synchronize between the two, Sirius uses the CanonicalEditPolicy that automatically updates the Diagram Resource upon changes in the Xtext Resource that is located in the Sirius ResourceSet.

It is important to highlight that the Xtext Resource located in the Xtext ResourceSet and the Xtext Resource located in the Sirius ResourceSet, are two separated resources. If the user makes changes in the Xtext editor, these changes are reflected only in the in-memory model (i.e., Xtext Resource in the Xtext ResourceSet). However, if the user decides to save these changes, the text file will be modified. When the Xtext Resource in the Sirius ResourceSet loads from the text file, the CanonicalEditPolicy updates the Diagram Resource.

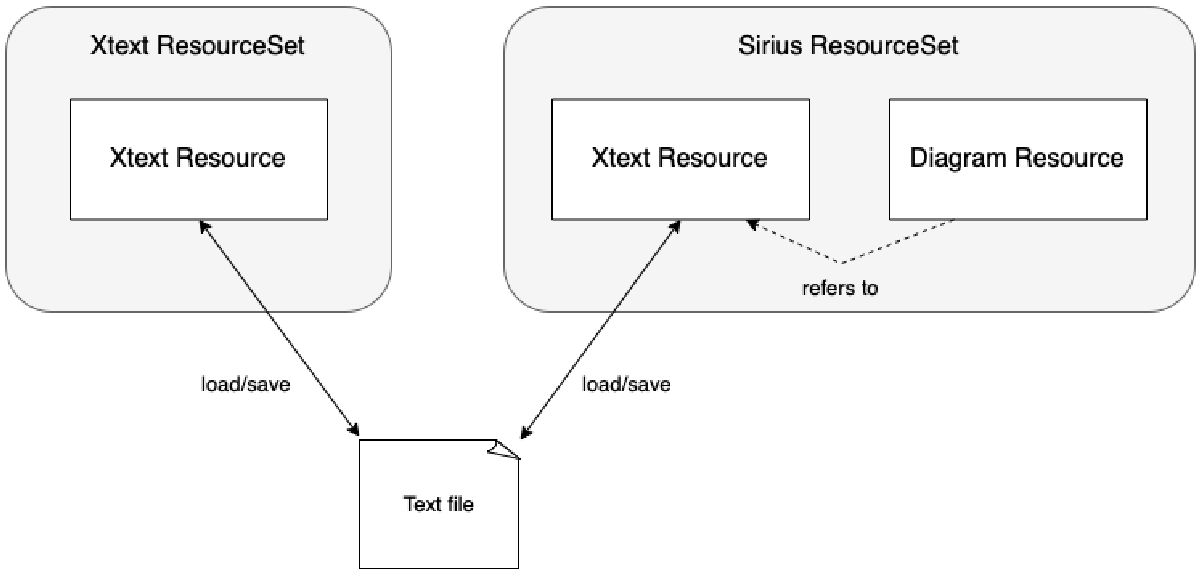


Figure 3: Out-of-the-box synchronization for Xtext and Sirius

3.1.2. Implementation

This prototype is based on a domain model that describes basic concepts about State Machines. The workflow of defining our prototype is as follows.

We start off by defining a grammar in Xtext that is used to describe State Machines. This grammar defines four concepts used to describe state machines; StateMachine (root element), InitialState, FinalState, State, and Transitions, and is detailed in Listing 1.

```
StateMachine:
        'InitialState' initialstate = InitialState
    'FinalState' finalstate = FinalState
        ('IntermediateState' '{' states+=State ( "," states+=State)* '}' )*
        'Transitions' '{' transitions+=Transitions ( "," transitions+=Transitions)* '}' ;
InitialState:
        name = ID ;
FinalState:
        name = ID ;
State:
        name = ID
        'InAction' InAction = STRING
        'OutAction' OutAction = STRING ;
Transitions:
        name = ID
        'Condition' condition = STRING
        'Action' action = STRING
        ('EnterState' enterstate=[State])?
        ('InitialStateTransition' initialstatetransition=[InitialState])?
        ('ExitState' exitstate=[State])?
        ('FinalStateTransition' finalstatetransition=[FinalState])? ;
```

Listing 1: Xtext grammar describing State Machines

After defining the grammar, we generate the Xtext artifacts. From this process, among others, we obtain the generated Ecore metamodel of the defined grammar and we register it in the Package Registry. The generated Ecore metamodel is detailed in Figure 4.

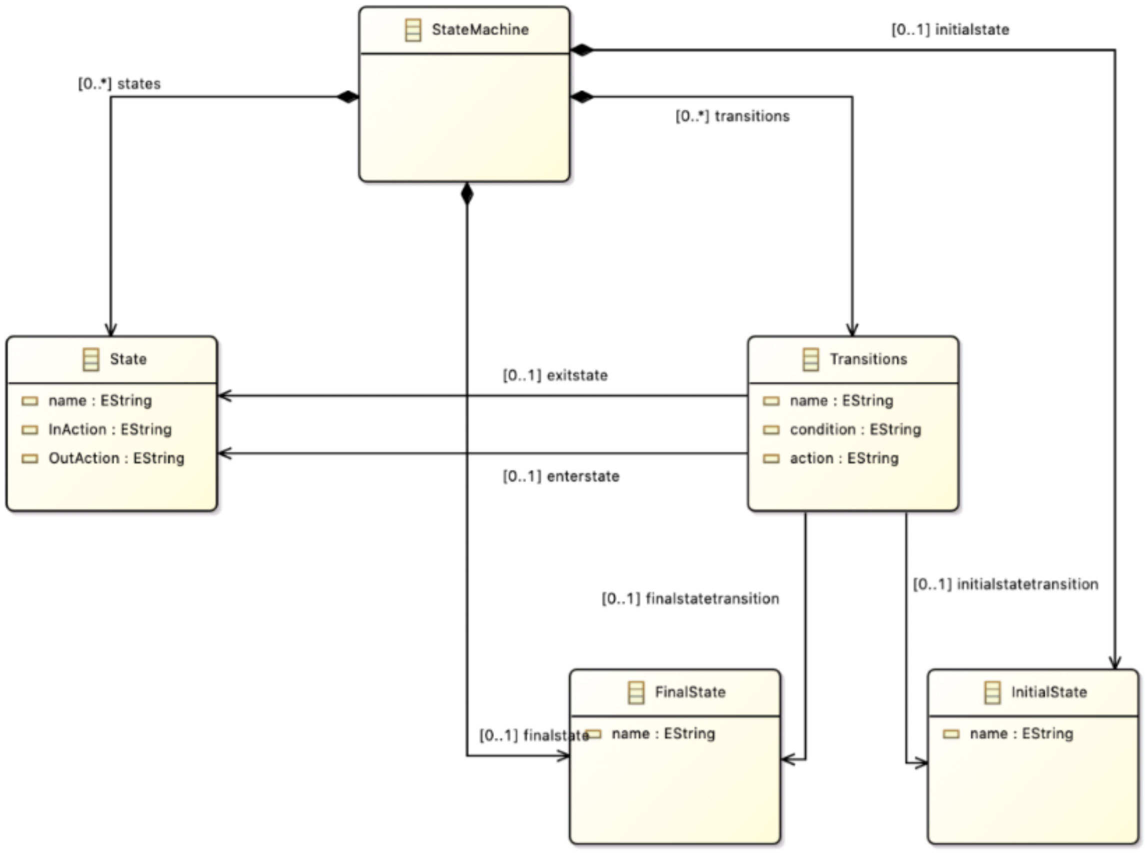
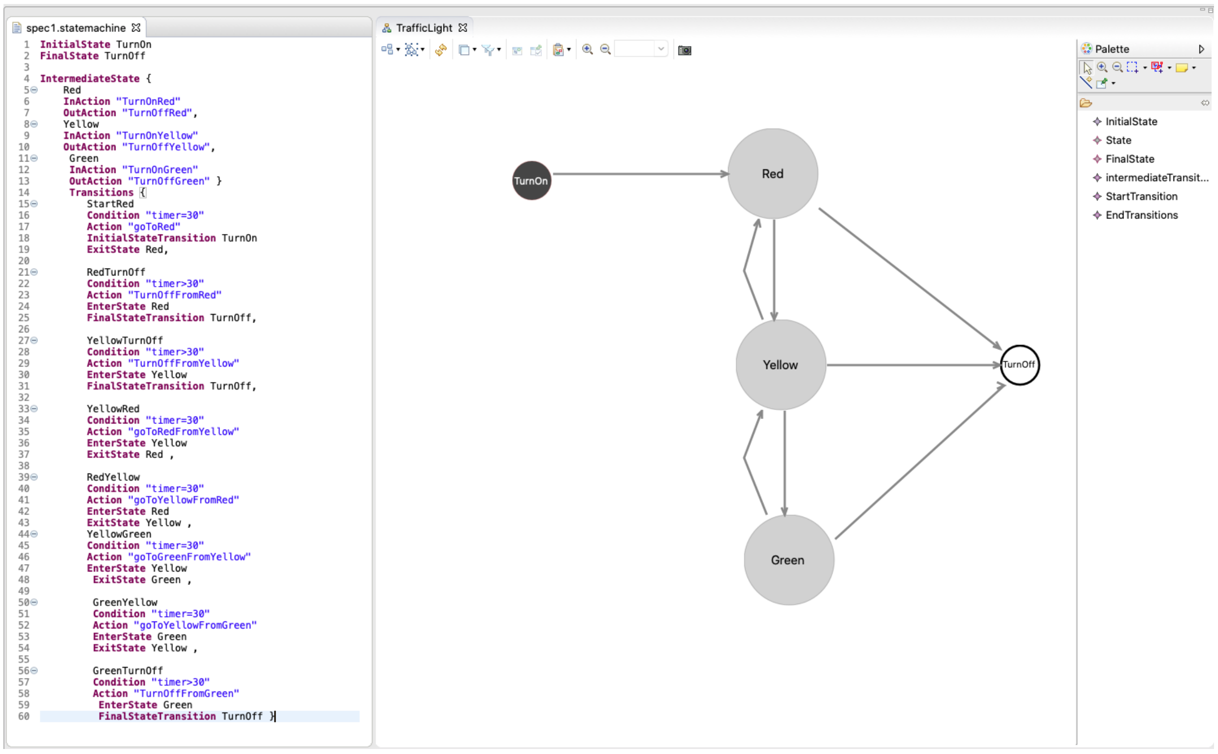


Figure 4: Generated Ecore metamodel

In a new runtime instance, using Sirius perspective, we create a new Viewpoint Specification Project, and add a new representation to the viewpoint. The representation can either be a diagram description, an edition table description, a cross table description, a tree description, or a sequence diagram description, but in our prototype we use the diagram description. After defining the diagram description, in the MANIFEST.MF file, we add the plug-in that defines the StateMachine metamodel and associate the metamodel to the diagram representation. This enables the graphical representation of instances of StateMachine by the diagram. In the Domain Class property of the diagram, we specify the root element and then in the default layer we start adding new diagram elements that correspond to model elements. More specifically, we define node elements used to display the State, InitialState and FinalState and element-based edges to display the Transitions. Moreover, we specify the style for each diagram element, in order to define the way they are graphically represented in the diagram. However, the current implementation can only display an existing model. In order to enable the creation of new model elements from the graphical editor using a palette, we create a new section and add Node Creation and Edge Creation elements to it. In addition, we define the action that will be executed by each element.

To finalize our prototype, we create a new Project, including an Xtext file that details a TrafficLight model, and a .aird file where we add this model and create a new representation for it. Figure 5 provides an overview of the two editors side-by-side.



```
InitialState TurnOn
FinalState TurnOff

IntermediateState {
    Red
    InAction "TurnOnRed"
    OutAction "TurnOffRed",
    Yellow
    InAction "TurnOnYellow"
    OutAction "TurnOffYellow",
     Green
     InAction "TurnOnGreen"
     OutAction "TurnOffGreen" }
     Transitions {
        StartRed
        Condition "timer=30"
        Action "goToRed"
        InitialStateTransition TurnOn
        ExitState Red,

        RedTurnOff
        Condition "timer>30"
        Action "TurnOffFromRed"
        EnterState Red
        FinalStateTransition TurnOff,

        YellowTurnOff
        Condition "timer>30"
        Action "TurnOffFromYellow"
        EnterState Yellow
        FinalStateTransition TurnOff,

        YellowRed
        Condition "timer=30"
        Action "goToRedFromYellow"
        EnterState Yellow
        ExitState Red ,

        RedYellow
        Condition "timer=30"
        Action "goToYellowFromRed"
        EnterState Red
        ExitState Yellow ,
        YellowGreen
        Condition "timer=30"
        Action "goToGreenFromYellow"
        EnterState Yellow
         ExitState Green ,

         GreenYellow
         Condition "timer=30"
         Action "goToYellowFromGreen"
         EnterState Green
         ExitState Yellow ,

         GreenTurnOff
         Condition "timer>30"
         Action "TurnOffFromGreen"
          EnterState Green
          FinalStateTransition TurnOff }
```

Figure 5: Prototype of integrating Xtext and Sirius

3.1.3. Relation to BUMBLE Features

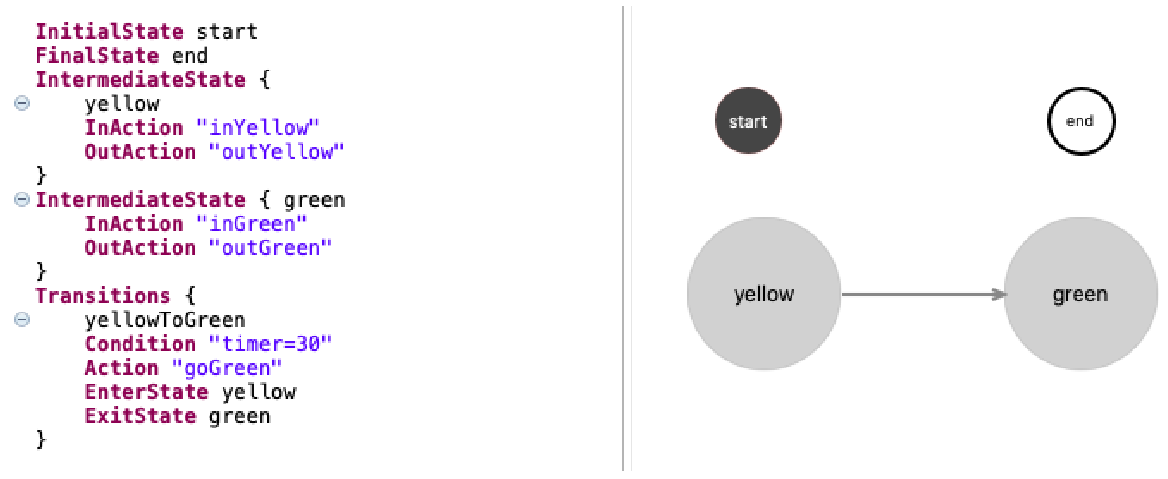
The integration of Xtext and Sirius is relatively easy to achieve by following the aforementioned workflow. For a single model, the user can instantiate both graphical and textual editors and switch between them. Moreover, the changes done in one editor, are propagated to the other upon save. However, this out-of-the box synchronization has the following downsides to it.

Synchronization on-demand

The synchronization between the two representations happens on demand and not on-the-fly, thus to propagate changes from one editor to the other the user needs to first save these changes. A possible solution would be for Sirius and Xtext to share the same resource, but the latter leads to further issues. A change in the textual editor results in Xtext removing the old AST and inserting a new one, based on the changes. Being that the AST/resource that is shared between Xtext and Sirius, serves as the semantic model for Sirius, when the AST is removed, the CanonicalEditPolicy updates the diagram model and removes the notational elements. When the new AST is inserted, the diagram model is recreated based on a default mapping, but all user customizations are lost.

Propagating changes with syntax errors

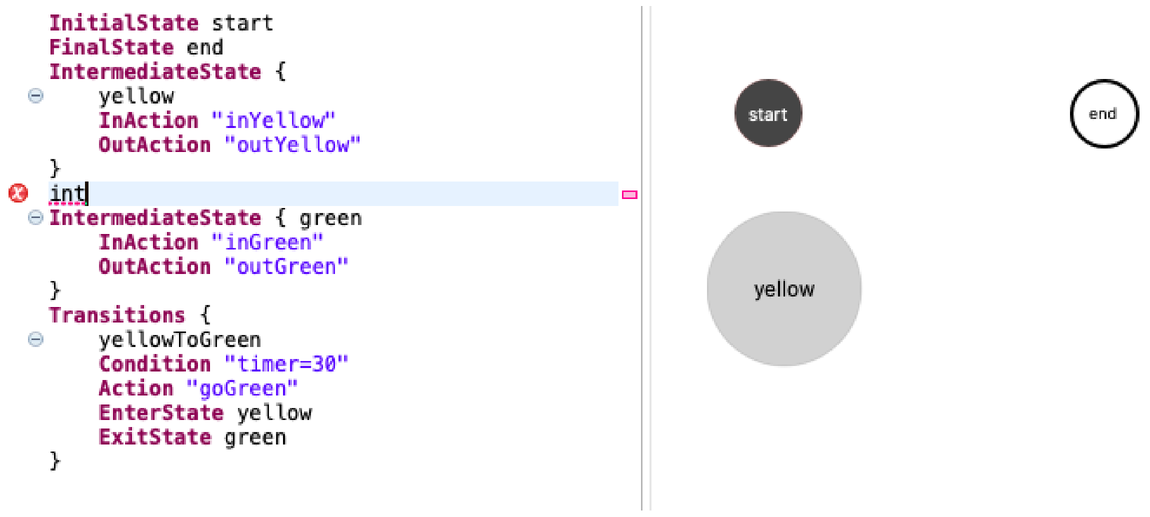
When using the textual editor, the user might write an expression that leads to syntax errors and save the file. Upon saving these file changes in the textual editor, the diagram in the graphical editor gets refreshed, and the elements defined after the syntax error are no longer available in the diagram. Figure 6 details the elements present in the textual and graphical editor before a syntax error.



```
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 6: State of the editors before the syntax error

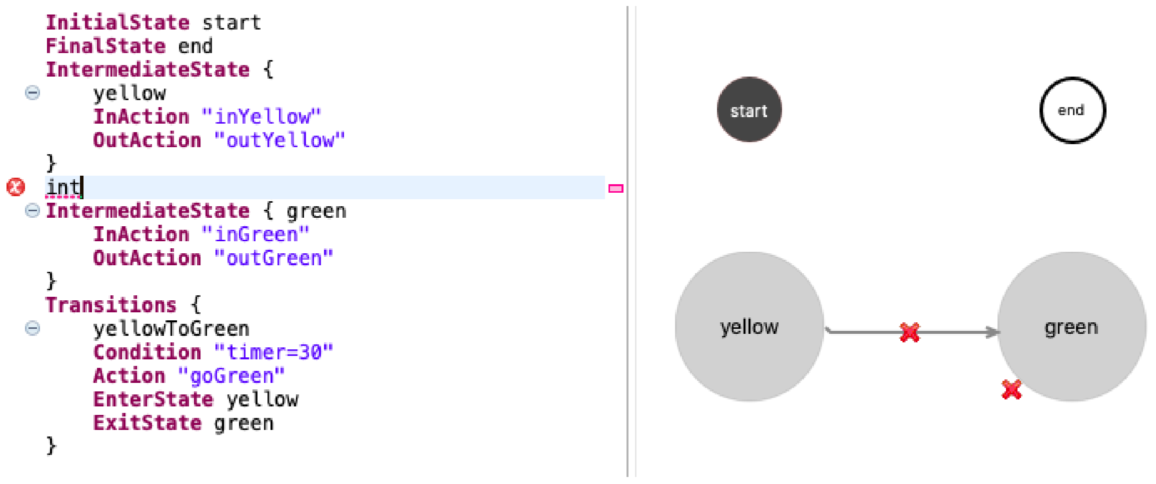
Figure 7 details the syntax error in the textual editor, and the remaining elements in the graphical editor after saving the file changes in the textual editor and refreshing the graphical editor.



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

Figure 7: State of the editors after the syntax error with an automatic refresh strategy

As it can be seen, all the elements defined after the syntax error (i.e., State “green” and Transition “yellowTogreen”), are no longer present in the graphical editor. Even if the user was to undo the changes, or fix the syntax errors, the diagram information cannot be recovered. A possible solution to this is changing the refresh strategy. At this point, the diagram gets refreshed upon saving the model in the textual editor. However, if we uncheck this option, and save the model in the textual editor that has syntax errors, we do not lose the elements in the graphical editor, but instead get a warning regarding the elements that will be lost if we decide to manually refresh the diagram. This provides the opportunity to fix the errors before refreshing the diagram and risk losing the notational information.



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

Figure 8: State of the editors after the syntax error without an automatic refresh strategy

Concurrent dirty states

A possible scenario is to modify elements in both graphical and textual editors without saving the changes in either of them. Being that both editors use their own memory instance of the model unless they try to save the changes, we encounter no issues.

However, if we first try to save the changes in the graphical editor and click on the textual editor to refresh, we get a warning that the file has been changed (from the graphical editor). The textual editor is now aware of that, because the file changes have been saved. If we decide to ignore the file changes, and then try to save the modified elements in the textual editor, we get an update conflict because the file has been changed on the file system (save operation in the graphical editor) and we can either choose to overwrite it or not. A similar thing happens if we try to save the changes in the textual editor first. To summarize, the user is always forced to choose to save the changes made on one editor only, losing the ones made on the other one. In the future, we will investigate whether other EMF persistency solutions like CDO can mitigate this issue.

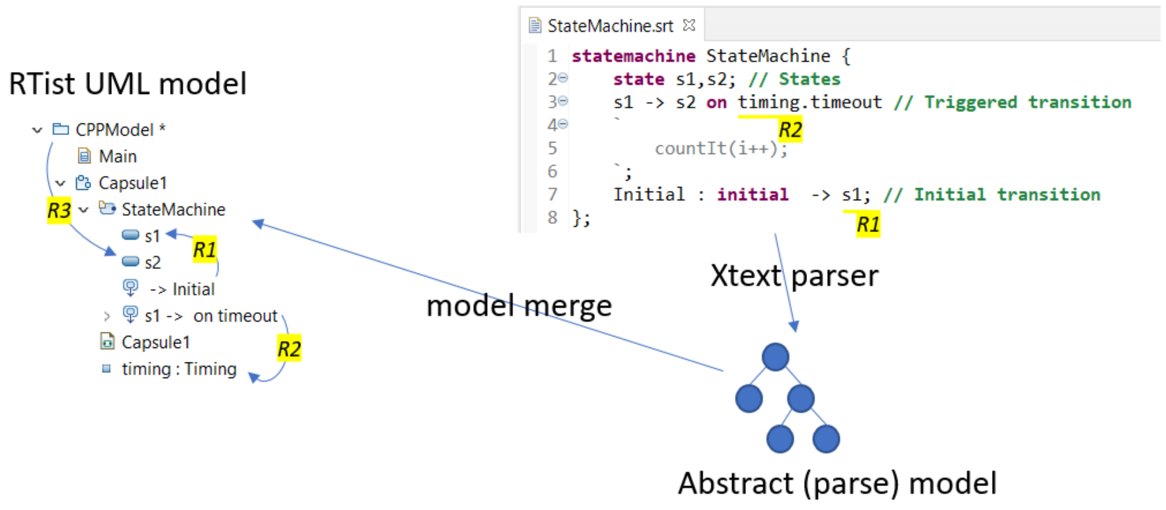
Possible solutions would be to have on-the-fly synchronization or in the case of on-demand synchronization to restrict the user from making changes to one editor if there are unsaved changes in the other editor.

3.2. Architectural Description of UC2 (Combined Textual and Graphical Modeling of State Machines in HCL RTist)

Two separate solution architectures have been developed for this use case. The first architecture is based on the Eclipse RCP, which is the most straightforward approach (given that HCL RTist is a modeling tool based on Eclipse). In parallel to working on this solution, we have also developed a second solution architecture which is based on the Language Server technology. Both these solution architectures are presented below.

3.2.1. Architecture Based on the Eclipse RCP

The picture below shows the functional principle for the use case implementation in HCL RTist:



```
statemachine StateMachine {
    state s1,s2; // States
    s1 -> s2 on timing.timeout // Triggered transition
    `
        countIt(i++);
    `;
    Initial : initial  -> s1; // Initial transition
};
```

Figure 9: Functional principle of the support for textual state machines in HCL RTist

3.2.1.1. Parsing with Xtext

A textual syntax for state machines has been implemented (shown in the above picture as the StateMachine.srt text file). The grammar for this syntax is defined using Xtext. While Xtext can automatically generate an EMF model to use as meta model for the defined syntax, this approach cannot be used for this use case since the requirement is to integrate the solution into the existing HCL RTist modeling tool which uses the standard UML meta model. Fortunately, Xtext also allows to use any existing EMF meta model by importing it into the grammar definition:

```
grammar com.hcl.xtools.dsl.statemachine.StatemachineRT with
org.eclipse.xtext.common.Terminals
import "http://www.eclipse.org/uml2/5.0.0/UML" as uml
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

The mapping from the concrete state machine syntax to the abstract syntax (i.e. the UML meta model) can in most cases be expressed directly in the Xtext parser rules. For example:

```
StateMachine returns uml::StateMachine:
	"statemachine" (name=NAME)? "{"
	region+=Region
	"}" ";"
;
```

Here “name” and “region” are features on the UML StateMachine meta class that we can use for storing necessary data while parsing.

But in some cases the mapping isn’t that straight-forward. For example:

```
InitialTransition returns uml::Transition:
	((name=NAME | ownedComment+=RedefineOrExclude
redefinedTransition=[uml::Transition|NAME]) ":")? "initial" "->"
target=[uml::Vertex|QNAME_OR_HISTORY]
	(effect=Effect)?
	";"
;
```

In the syntax there is not enough information for setting up the "source" feature which is mandatory for all UML transitions. Fortunately Xtext is very configurable and allows us to write our own Java code for controlling how to create the proper UML representation from the textual state machine syntax. We have used these two customization features:

1. Store information as comments while parsing (see "ownedComment" in the example above). Then in the Xtext AST factory we convert such comments into the proper UML representation.
2. Use custom code in the Xtext Linker for creating additional information based on resolved links in the parse model. For the above example we can identify the initial transition based on the fact that "target" is bound, but "source" is not. We can then create an Initial pseudo state and use that as the source of the transition, to ensure the created model conforms to UML rules.

3.2.1.2. Model merge

The abstract parse model is managed by the Xtext parser and cannot be directly inserted into the main HCL RTist model. It is necessary to clone the parse model, and merge the clone into the main model. The trigger for doing this is based on an Xtext model listener. It gets called whenever the textual state machine is modified.

The merge itself is based on a name and structure based comparison between the cloned parse model and the existing HCL RTist model. The HCL RTist model is then updated according to the changes detected in the parse model.

3.2.1.3. Managing model references

There are three kinds of model references, shown in the picture above as R1, R2 and R3.

- **R1** Reference within the parse model
  For example, the reference to state "s1" from the initial transition.
- **R2** Reference from the parse model to the HCL RTist model
  For example, the reference to port "timing" from the triggered transition.
- **R3** Reference from the HCL RTist model to the parse model
  For example, a dependency (not shown in the picture) from package "CPPModel" to state "s2".

References of the kind R1 and R2 are set-up by the Xtext linker based on names used in the textual syntax. An Xtext scope provider has been implemented to perform this linking according to desired name resolution rules. In many cases (especially for references of kind R2) it is necessary for the scope provider to use the HCL RTist model element that is the context for the textual state machine (a capsule). This is accomplished by treating the state machine text file (.srt) as a fragment file. A fragment file is referenced in another (parent) model file, and it's hence possible to find out to which part of an HCL RTist model a certain textual state machine belongs.

References of the kind R3 require that the Xtext parse model element have stable and predictable EMF URIs. This is not the case by default, where elements may get new URIs each time the text is parsed. To address this, an Xtext fragment provider (IFragmentProvider) has been implemented. The fragment is the last part of the element URI, and by default this part of the URI is not stable. The implemented fragment provider ensures that fragments are based on fully qualified names, which is the most stable way of identifying an element in a textual syntax. References of the kind R3 will hence use a fully qualified name to reference elements inside the textual state machine. As long as such elements are not renamed, and none of its container elements also are not renamed, the reference will not become broken, even after reparsing the text.

3.2.1.4. Synchronization between model notations

The textual representation of a state machine is another model notation that needs to be updated if the state machine model changes. This works in a very similar way to how the graphical notation (i.e. a statechart diagram) gets updated. The textual state machine editor uses a resource change adapter to intercept notifications when the state machine model changes, and triggers a serialization of the state machine to get a new textual representation.

Note that the mapping between concrete syntax and abstract syntax is always ambiguous in the sense that multiple concrete syntaxes yield the same abstract syntax representation. For example, comments and whitespace in the concrete syntax are not present in the abstract syntax. This means that serialization will "normalize" the syntax, i.e. the abstract syntax is mapped into one particular concrete syntax that is equivalent, but usually not identical, to the original concrete syntax typed by the user.

3.2.1.5. Eclipse plugin architecture

The picture below shows the new Eclipse plugins developed for supporting textual state machines in HCL RTist, and the open-source and commercial plugins they depend on.

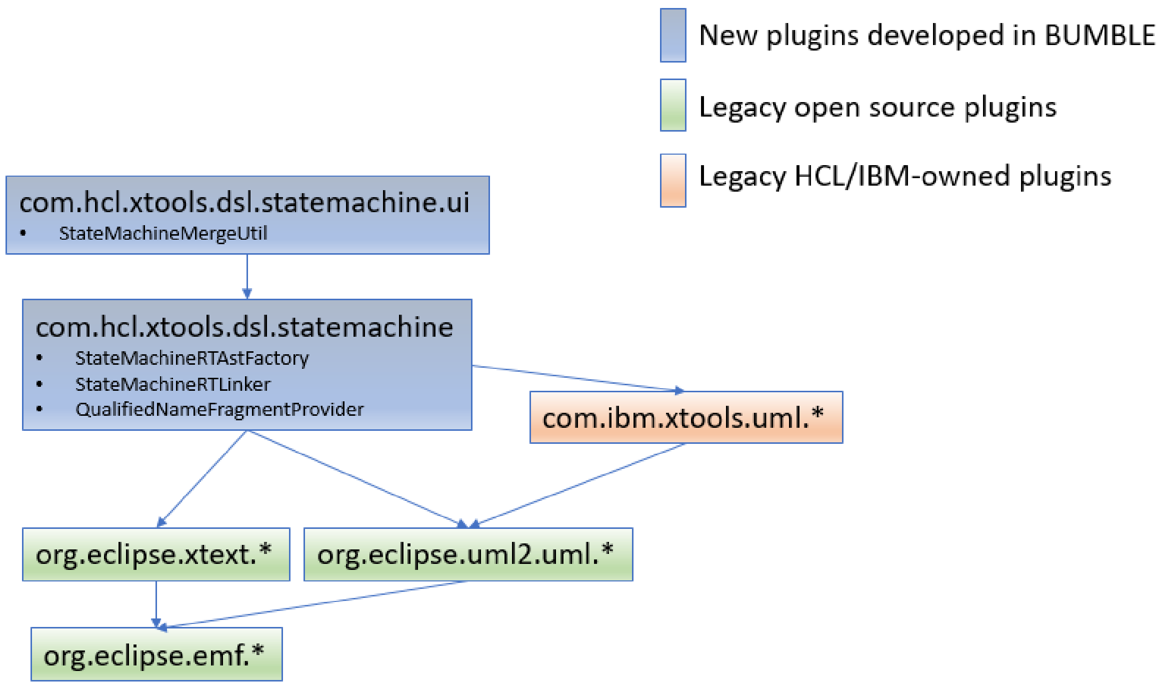

New plugins developed in BUMBLE
Legacy open source plugins
Legacy HCL/IBM-owned plugins
com.hcl.xtools.dsl.statemachine.ui
• StateMachineMergeUtil
com.hcl.xtools.dsl.statemachine
• StateMachineRTAstFactory
• StateMachineRTLinker
• QualifiedNameFragmentProvider
com.ibm.xtools.uml.*
org.eclipse.xtext.*
org.eclipse.uml2.uml.*
org.eclipse.emf.*

Figure 10: Eclipse plugins with some key classes

In the picture, some key classes that implement the functionality described above are mentioned:

- **StateMachineMergeUtil**
  Merges the Xtext parse model into the HCL RTist model. See Model merge.
- **StateMachineRTAstFactory** and **StateMachineRTLinker**
  Modifies and augments the Xtext parse model so it becomes a valid UML model. See Parsing with Xtext.
- **QualifiedNameFragmentProvider**
  Ensures that Xtext parse model elements get stable and predictable EMF URIs based on their qualified names. See Managing model references.

3.2.1.6. Relation to BUMBLE Features

The features described above are all related to the BUMBLE feature “Blended Syntaxes & Modeling (B)”. The Xtext generated editor essentially becomes another view on the model, in a similar way to how other Eclipse views (such as Project Explorer, Properties view, diagrams etc) are. Just like other views it allows a subset of the model to be viewed and edited (the subset defined by the implemented syntax).

The main difference with this textual view is that it internally uses a different model (the parse model created by Xtext), but that thanks to the implemented Xtext customizations this model is made compatible and compliant with the UML model so that the user gets the feeling of working with a single model through multiple views and notations.

3.2.2. Architecture based on the Language Server Technology

The promise of the language server technology is to enable the decoupling of the language functionality from the IDE being used. This is done by breaking out the language-aware modules into a separate application which runs in a separate process than the IDE and communicates with it using the language server protocol (LSP). The LSP is standardized and supported by many IDEs, including the Eclipse IDE; we chose to explore this solution in the context of two other IDEs, namely Visual Studio Code (VS Code) and Eclipse Theia. These IDEs were chosen based on their popularity in the market, and on the fact that they share the same extension model (i.e., an extension developed for VS Code can also run in Eclipse Theia). Another reason is that both VS Code and Eclipse Theia are implemented via web technologies, allowing developers to access their functionalities via a web browser.

To avoid confusion with the previously presented solution, we will refer to HCL RTist as "RTist in Code" for this solution. We chose to implement the language server in Java so that components from HCL RTist can be easily reused for RTist in Code. The picture below shows the high-level architecture of RTist in Code:

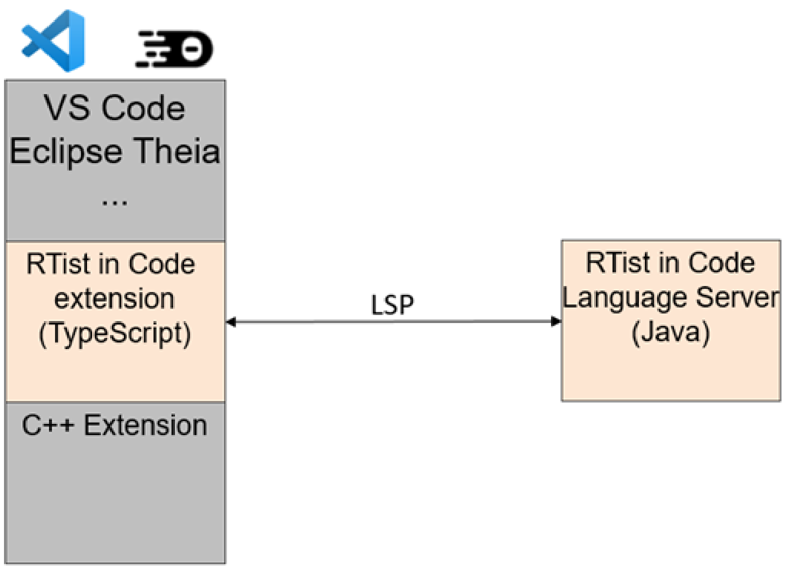

VS Code
Eclipse Theia
...
RTist in Code extension (TypeScript)
C++ Extension
LSP
RTist in Code Language Server (Java)

Figure 11: High-level architecture of RTist in Code

Note that just like RTist makes use of CDT for C/C++ support in Eclipse, RTist in Code makes use of a C++ extension for Visual Studio Code / Eclipse Theia. There are more than one such extension available, but for now we have assumed Microsoft´s C/C++ extension is used. In the future also other C++ extensions, for example "clangd", will be supported too.

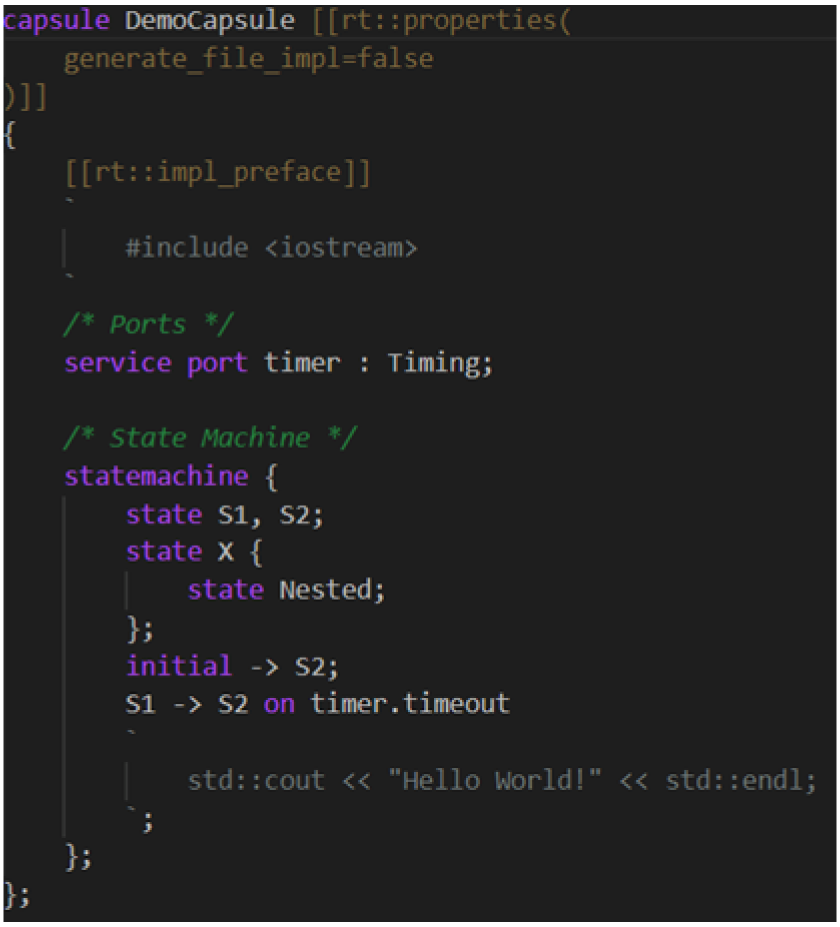
3.2.2.1. Using Xtext in a Language Server

Just like for RTist, we use Xtext for implementing the parser in RTist in Code. However, while in RTist this parser only needs to cover state machines, it needs to cover the whole UML-RT language in RTist in Code. The reason is that we cannot reuse the modules of RTist used for creating the part of the model that contains the state machines. The Language Server Protocol is designed for textual languages and VS Code and Eclipse Theia are text-centric IDEs, encouraging the editing of models using a textual notation.

Since it's natural to think about a language in terms of its syntax, we chose to give a new name to this new implementation of the modeling language, even if the concepts are identical from UML-RT. We call the language *Art*, and in addition to providing a textual syntax for well-known UML-RT concepts such as capsules, protocols and events, it also provides a few additional capabilities:

- The ability to embed C++ code snippets in specific areas of the model (Art uses C++ as action and expression language).
- The ability to annotate model elements with typed properties, in order to augment the model with data that can be used by tools that operate on the model (e.g.,code generators, semantic validators). By having a generic property syntax we can keep the Art language small, simple and resilient to future changes.

The picture below shows an example of a model conforming to the Art language.



```
capsule DemoCapsule [[rt::properties(
    generate_file_impl=false
)]]
{
    [[rt::impl_preface]]
    `
        #include <iostream>
    `

    /* Ports */
    service port timer : Timing;

    /* State Machine */
    statemachine {
        state S1, S2;
        state X {
            state Nested;
        };
        initial -> S2;
        S1 -> S2 on timer.timeout
        `
            std::cout << "Hello World!" << std::endl;
        `;
    };
};
```

Figure 12: Example of Art model

The example defines a capsule containing a timer port and a simple state machine. Grey text enclosed in backticks are embedded C++ code snippets. Backtick was chosen as the delimiter to avoid the need for escaping characters in the embedded code snippet (backtick is an unusual character in C++ code). Orange text enclosed in double square brackets are properties. Properties can have values of boolean, enumeration, integer and string type, and since a code snippet at Art level is nothing but a string, properties can also store code snippets. This is shown in the example above for the property "rt::impl_preface".

Note that the textual state machine syntax from the Eclipse RCP solution is reused in the Art language.

Xtext provides the necessary glue code for integrating the generated parser, and related language features such as content assist etc, with the LSP.

3.2.2.2. Graphical Diagrams with GLSP

The Graphical Language Server Platform (GLSP) is a technology built on the same core idea of LSP, but for graphical languages rather than textual ones. GLSP consists of a server part where the language implementation is realized, and a UI part that runs in the IDE and renders the diagrams provided by the server. Hence the architecture fits nicely with the high-level architecture shown in Figure 11 above.

Currently, three types of diagrams are supported for the Art language: class diagrams, structure diagrams, and state diagrams. Class diagrams show relationships (such as inheritance) between capsules, classes, and protocols. Structure diagrams show the internal structure of a capsule, i.e., which parts it contains and how they are connected. State diagrams show the state machine behavior of a capsule or class.

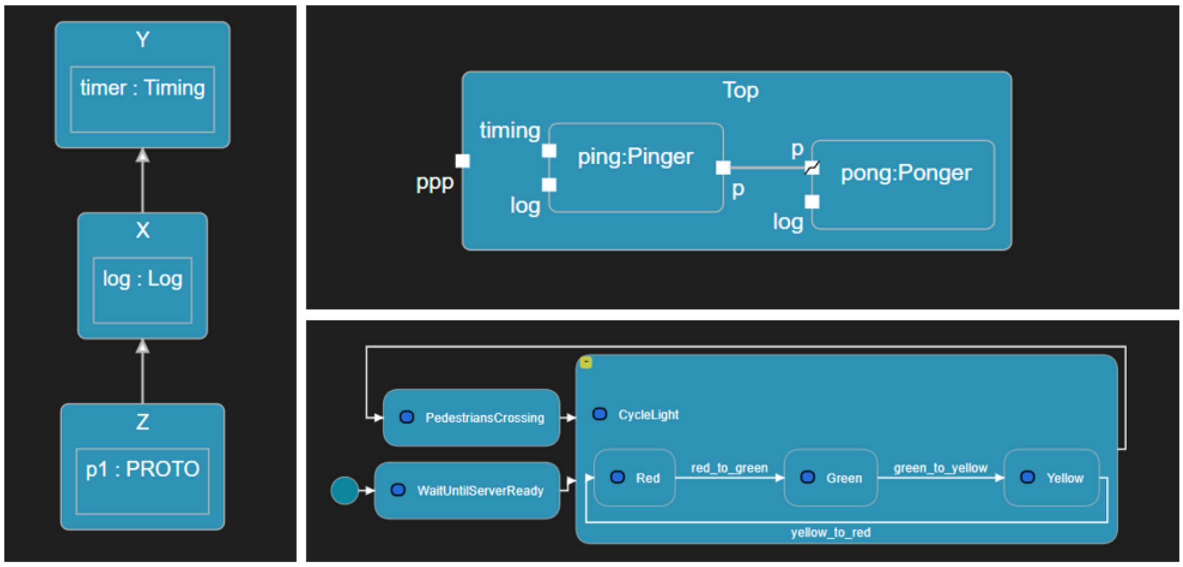


Figure 13: Art diagrams implemented with GLSP

The diagrams refresh automatically when the underlying Art model is changed. This is implemented by registering a listener which gets notified whenever an Art file is modified. On each notification to this listener it sets a 1.2 second timer, and when this timer times out the diagrams that show anything from the modified Art file are refreshed. The use of a timer avoids too frequent diagram updates after every keystroke, and ensures a responsive editing experience.

Currently Art models are mostly modified by editing the textual Art file, and the diagrams are therefore mostly read-only. However, GLSP provides the ability to build proper editors and as long as all graphical edit commands can be mapped to Art model updates, it is possible to build rich editors using GLSP. RTist in Code allows model elements to be renamed from diagrams. This graphical editing action is mapped to a “refactor-rename” operation where the model element, and all references to it, are updated to a new name.

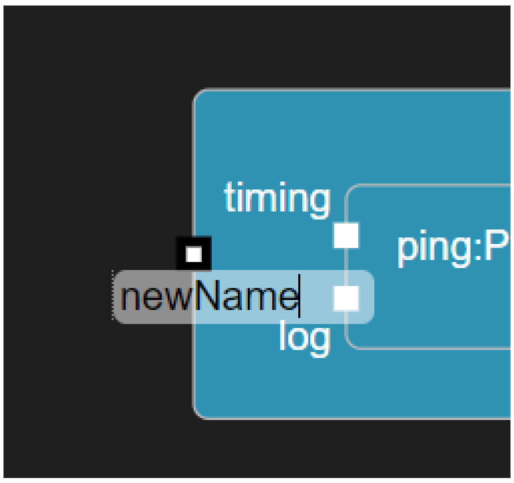


Figure 13: Renaming a port from within a diagram

3.2.2.3. Language Server Modules

We can break down the language server of RTist in Code into three main modules, as shown in the picture below. Differently from the solution architecture based on the Eclipse RCP, these modules are plain JAR files, and not plugins. The whole language server is a plain Java application that loads these three JAR files.

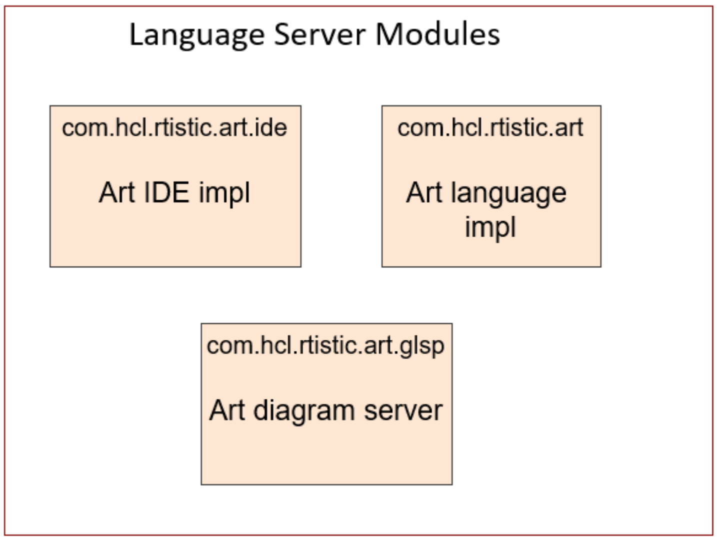


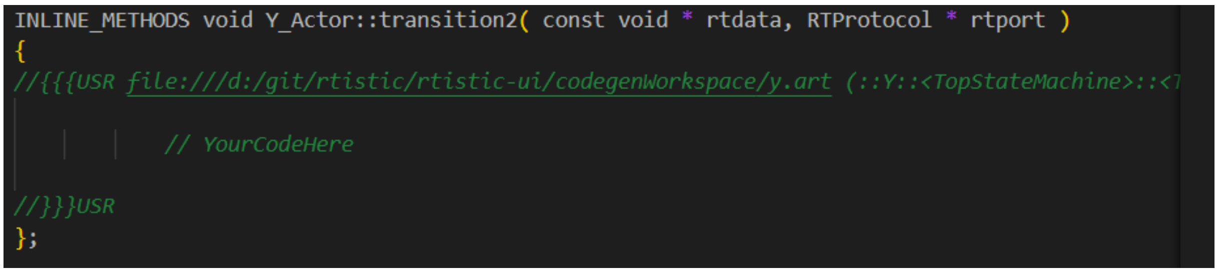
Figure 13: Language server modules

- **com.hcl.rtistic.art**
  This component contains the Xtext generated parser and other language features that are not dependent on the IDE. For example, semantic validation, support for code formatting etc.
- **com.hcl.rtistic.art.ide**
  This component contains functionality that interacts with the IDE. For example, it provides commands that can be invoked from the IDE extension using APIs provided by the LSP. It also contains the glue code provided by Xtext for implementing the LSP.

- **com.hcl.rtistic.art.glsp**
  This component contains the server-side code of GLSP and the implementation of the graphical diagrams for the Art language.

3.2.2.4. Generated Code as Another View of the Model

Just like diagrams update automatically 1.2 seconds after an Art file has been modified, we also trigger the generation of C++ code at the same time. This is a major difference with RTist where code generation is always the result of an explicit user action. The main reason for implementing this incremental and automatic code generation functionality is to let the generated C++ code act as another view of the model. It is a view that just like diagrams currently is mostly read-only, but since generated code contains certain code blocks enclosed in special comments, it is possible to let the user modify such code blocks and automatically synchronize such changes back to the Art file.



```
INLINE_METHODS void Y_Actor::transition2( const void * rtdata, RTProtocol * rtport )
{
//{{{USR file:///d:/git/rtistic/rtistic-ui/codegenWorkspace/y.art (::Y::<TopStateMachine>::<T

            // YourCodeHere

//}}}USR
};
```

Figure 14: Editable code blocks in generated C++ files

3.2.2.5. Relation to BUMBLE Features

The features described above are all related to the BUMBLE feature “Blended Syntaxes & Modeling (B)”. Contrary to the Eclipse RCP solution where the model is stored in XML files that are independent of any concrete model view, with this solution there is no stand-alone model representation and instead the textual Art files are used for storing the model. This means that textual editing of Art files is the main way in which the model is created and updated. This is a solution that fits well with the text-centric approach of IDEs such as Visual Studio Code and Eclipse Theia.

Editing from other views, such as graphical diagrams and generated C++ code is possible, but requires an unambiguous mapping of each editing action to corresponding updates of the Art file. To support a higher degree of editing freedom, something that in particular may be desired in the graphical diagrams, it will be necessary to store additional graphical information in files next to the Art files.

3.3. Architectural Description of UC3 (Vehicular Architectural Modeling in EAST-ADL)

The *Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL)*[13] is a DSML for the specification of automotive embedded systems, which is applied at Volvo Technology AB. The Eclipse-RCP-based tool suite EATOP[14,15] provides tree and form editors to

[13] http://www.east-adl.info/
[14] https://www.eclipse.org/eatop/
[15] https://bitbucket.org/east-adl/east-adl/

enable the tool-based specification of EAST-ADL models based on EMF. Furthermore, a graphical notation similar to class diagrams exists that is able to depict the hierarchies of EAST-ADL models. In the BUMBLE Use Case 3 and w.r.t. BUMBLE feature (B), we want to complement these existing editors with a textual notation and a seamless switching and synchronization between the textual representation and the tree-/form-based editors, that is, blended EAST-ADL modeling.

Besides the editors, EATOP provides the (de-)serialization from/into a special persistence format called EAXML. EAXML is a customized variant of the conventional EMF persistence format XMI and, beyond custom XML tags, preserves the order of the persisted elements according to their tree-based representation (cf. core stakeholder requirement C3.6 in BUMBLE deliverable D2.2).

In Section 3.3.1, we describe the architecture of the new editor for the textual notation and its relationship to the architecture of the existing EATOP editors in the context of the Eclipse RCP. Subsequently, we describe the architecture for the new textual editor in the context of a client/server architectural style, which enables the application of the editor in VS Code. In Section 3.3.2, we present the architecture of an approach that automates the evolution of the textual editor in the case of EAST-ADL language evolutions. Finally, we establish a relationship to the BUMBLE features.

3.3.1. Architectural Description for the Eclipse RCP

Figure 11 depicts, among other things, selected EATOP plugins that are relevant to the activities for the BUMBLE Use Case 3 w.r.t. the BUMBLE feature (B). The EATOP tree- and form-based editors are represented by the plugins `o.e.eatop.examples.[editor/explorer/actions]`. The customized (de-)serialization from/into EAXML is represented by the plugin `o.e.eatop.serialization`. The plugins `o.e.eatop.geastadl` and `o.e.eatop.eastadl22` provide the actual DSML metamodels `geastadl.ecore` and `eastadl22.ecore`, respectively. These metamodels describe the language concepts of EAST-ADL in the version 2.2.

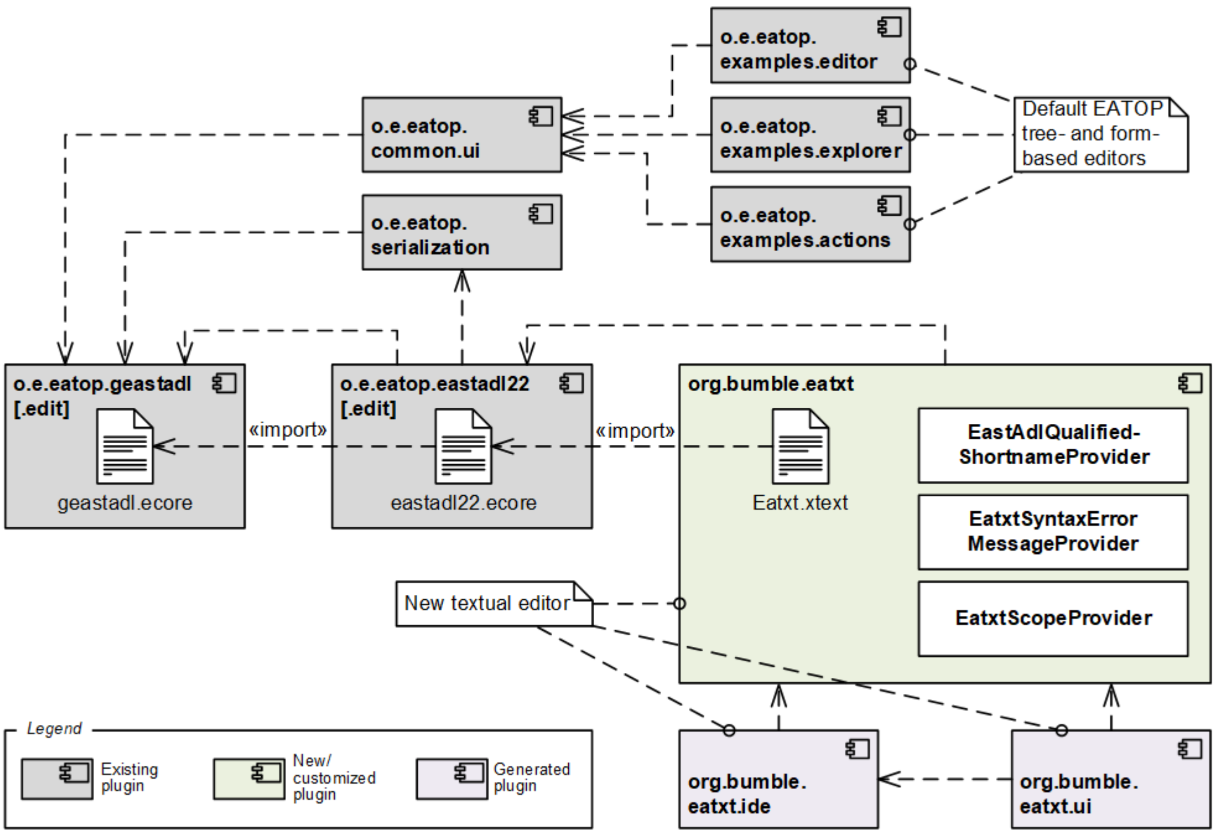


Figure 11: Architecture for the textual editor for EAST-ADL models and its relationships to EATOP

In the first stage, the University of Gothenburg (GU) conceived a textual notation for EAST-ADL in collaboration with Volvo Technology AB. For this purpose, we proposed different variants of the language and refined it in multiple stages to one variant that was favored by Volvo's engineers. In the second stage, we implemented a textual editor for this notation based on Xtext, which we call *EATXT*. Xtext generates different plugins that make up the EATXT editor, namely `org.bumble.eatxt`, `org.bumble.eatxt.simplified.ide`, and `org.bumble.eatxt.ui` as depicted in Figure 11. Furthermore, Xtext allows the generation of grammars based on existing Ecore metamodels, which we exploited for our use case. In Figure 11, the resulting generated grammar `Eatxt.xtext` as part of the plugin `org.bumble.eatxt` imports the EAST-ADL language concepts from `eastadl22.ecore` and associates them with a textual concrete syntax.

Beyond the pure generation of these plugins, we had to customize certain features in the plugin `org.bumble.eatxt` due to several reasons:

1. We had to adapt the grammar as part of `Eatxt.xtext` to achieve the textual notation as favored by Volvo. In the case that changes in the EAST-ADL metamodel `eastadl22.ecore` occur, we automated these adaptations with a grammar optimization plugin (cf. Section 3.3.3).
2. By default, Xtext assumes that for any metamodel/grammar that there is a mandatory attribute `name` which is used as the unique identifier of all the model elements. However, in EAST-ADL, there is an optional `name` attribute but a different mandatory and uniquely identifying attribute called `shortName`. Thus, we had to make the Xtext framework aware of this attribute instead of the default behavior, which we do in the class `EastAdlQualifiedShortnameProvider` (cf. Figure 11).

3. As the whitespace-aware language feature of our textual EAST-ADL notation is no default Xtext behavior, Xtext's default error messages for potential parsing errors are typically not very meaningful to the user of the editor. Furthermore, the EAST-ADL metamodel restricts certain String attributes to be in a certain format w.r.t. regular expressions. If the user of the textual notation does not adhere to such a format, the error message of the underlying metamodel is not meaningfully translated to the resulting Xtext error message. To provide meaningful error messages to the user in such cases, we customize the corresponding error messages in the class `EatxtSyntaxErrorMessageProvider` (cf. Figure 11).
4. As EAST-ADL uses the attribute `shortName` and not the attribute `name` as a unique model element identifier (see above), we also had to adapt the default Xtext behavior regarding scoping for cross-references between model elements of an EAST-ADL model (e.g., references to types). In this context, we let Xtext's auto-completion feature let propose the `shortName`s of other model elements, which we implemented in the class `EatxtScopeProvider` (cf. Figure 11).

We are currently working on the blending between our EATXT and the other EATOP editors via a synchronization in the EAXML persistence format.

3.3.2. EATXT in VS Code Applying the Language Server Protocol

As mentioned in Section 2.3 and also described for a different use case in Section 3.2.2, the Language Server Protocol (LSP) enables transferring solutions relying on Eclipse RCP / EMF to modern web-based IDEs like VS Code. In order to provide the textual EAST-ADL editing capabilities also for engineers favoring VS Code, we hence applied the LSP approach to transfer EATXT to this IDE.

Figure 12 describes this realization. On the server side as part of the Eclipse plugin `org.bumble.eatxt` (cf. last section), we provide the Java class `RunServer` that spawns an instance of the class `LanguageServerImpl` provided by the plugin `o.e.xtext.ide`. On the client side, we developed the VS Code extension `eatxt-vscode` (being also fully compatible with Eclipse Theia). Beyond the EATXT language configuration, this extension provides the TypeScript module `Extension`. This module spawns an instance of the module `LanguageClient` (provided by the VS Code package `vscode-languageclient`) when an EATXT file is opened in the runtime workspace of VS Code.

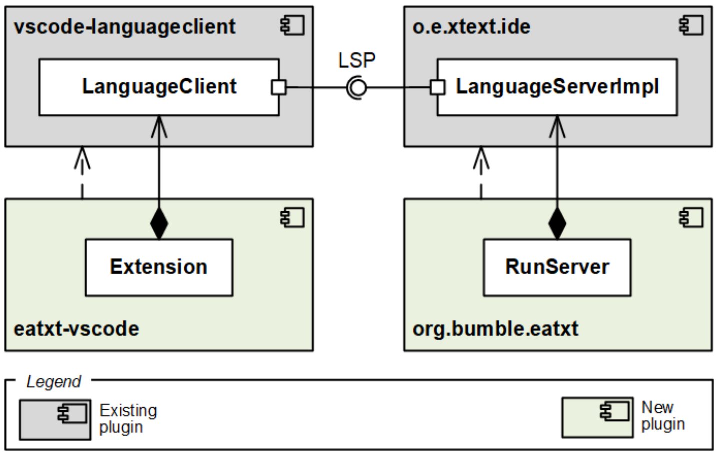


Figure 12: Architecture for bringing the EATXT editor to VS Code and Eclipse Theia via LSP

3.3.3. EATXT Grammar Optimization

The language's concrete grammar is what we call grammars (like our EATXT grammar, cf. center of Figure 13), generated from Ecore metamodels (like EAST-ADL's metamodel, cf. left-hand side of Figure 13). The generated grammar is always user-unfriendly and difficult to use, so we coordinated the EATXT language design with Volvo to make many adjustments to the grammar that is initially generated by the Xtext framework from the EAST-ADL metamodel (cf. Section 3.3.1).

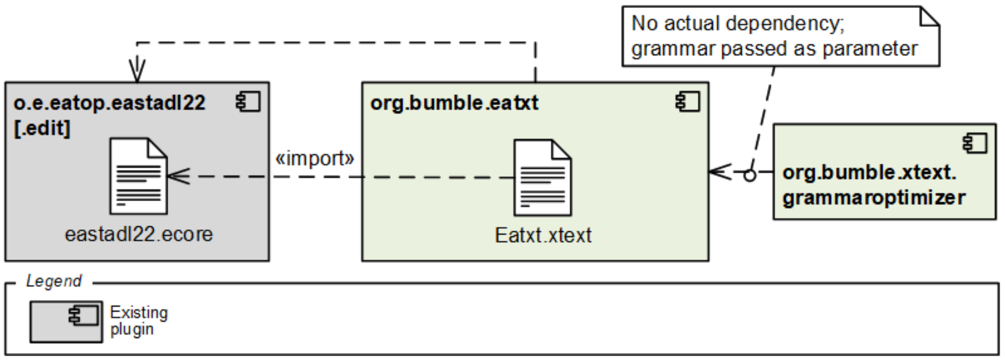

No actual dependency; grammar passed as parameter
o.e.eatop.eastadl22 [.edit]
eastadl22.ecore
«import»
org.bumble.eatxt
Eatxt.xtext
org.bumble.xtext. grammaroptimizer
Legend
Existing plugin

Figure 13: Architecture for the post-processing of the EATXT grammar

Sometimes the size of a grammar is large, e.g., the generated grammar of EAST-ADL has about 3000 lines of text. In such a large grammar definition file, it is obviously cumbersome to do repetitive operations such as moving the shortName attribute to the front of a production rule and applying the same operation to all the production rules. To efficiently incorporate these adaptations to the grammar in case the EAST-ADL metamodel evolves, we developed a post-processing plugin called `org.bumble.xtext.grammaroptimizer` (cf. right-hand side of Figure 13), which automates these grammar modifications. It will play a huge role in the evolution of the metamodel. The grammar depends on the metamodel, and the grammar needs to be changed as the metamodel evolves which is usually in the form of regenerating the grammar from the evolved metamodel. Therefore, the originally optimized grammar will be overwritten by the newly generated grammar, which results in the need to optimize it again. Apparently, GrammarOptimizer automates the whole grammar modification work again.

The GrammarOptimizer uses Regular Expressions to find parts of the grammar that is initially generated by Xtext and to replace these parts with the corresponding parts of our modified grammar. In this context, the grammar is passed as a parameter, so that no actual dependency between the plugins exists. We opted for post-processing the grammar instead of directly modifying the Xtext grammar generator, because the latter approach would have led to code that would be very specific to each Xtext version and thereby would have to be changed on every Xtext version increment. In contrast, the application of conventional Regular Expressions guarantees the stability of our code.

We describe the functional principle of the plugin in the BUMBLE Deliverable D3.4.

3.3.4. Relation to BUMBLE Features

Blended Syntaxes & Modeling (B)

As a prerequisite to achieve this BUMBLE feature, we developed a new textual syntax and editor called EATXT, which complements the existing EAST-ADL tree-/form-based notations and their

EATOP editors. The EATXT editor can be applied both within the Eclipse RCP and as part of a client-/server-based architectural style. The latter enables to edit EATXT files in cloud IDEs like VS Code and Eclipse Theia.

In order to extend the BUMBLE Use Case 3 in such a way that it supports blended EAST-ADL modeling, we are currently working on both the Eclipse RCP and the client-/server-based architectural style. One the one hand, in the Eclipse RCP architectural style that the EATOP tool suite uses (cf. Section 3.3.1), we want to achieve blending between the EATXT editor and the other EATOP editors through a synchronization in the EAXML persistence format. However, with this architectural style, we will probably have the same issues as the ones that we discovered in the canonical use case 1 (cf. Section 3.1.3).

Evolution (E)

As mentioned above, our design of the grammar for EATXT deviates strongly from the grammar that Xtext initially generates from the EAST-ADL metamodel. Thus, together with the actual design of the EATXT grammar, we directly co-conceived and co-implemented an automatic grammar optimizer to efficiently cope with potential EAST-ADL metamodel evolutions.

3.4. Architectural Description of UC6 (Blended Editing and Consistency Checking of SysML Models and Related Program Code)

At Saab, SysML is the modeling language chosen for designing complex system architectures and designing system behavior. There are a number of different tools that provide a graphical interface for viewing and creating models in SysML.

A system model is created to describe the decomposition of the system into so-called “system components” that are then assigned to software or hardware components. Moreover, the intended functionality of those components are modeled via behavioral diagrams (e.g., state machines) and internal block diagrams. The software components are implemented in various ways, among which C++ is one.

3.4.1. Model and Code Matcher

We have set up a bridge between C++ code and the XML description of the system described in SysML. A matcher tool implemented in Java indexes both the code and the SysML model and finds probable links between them. These links can be improved by adding additional heuristics for guiding the tool in creating links. Links can also be invalidated. The matcher tool indexes can be stored persistently. The matcher has been deliberately IDE agostic to be able to augment with several different models and modeling tools.

The Matcher assumes that there are architectural rules in both models (SysML and C++) that can be matched. It creates two tuples for matching. From the SysML model elements it creates a tuple for each model element € consisting of €:<structure $S_€$, element type $€_T$, element name $€_N$ and element stereotype $€_T$>. For each code element C it creates a tuple C<Structure S_C, code type C_T and code name C_N>. The matcher has a set of conditions <c_1,c_2,c_3>. These conditions are related to structure, type and name.

The first condition c_1 compares the structure S_C and $S_€$ to make sure that both elements belong to the same architectural entity. The structure is defined as the relation between hierarchy in the model, and repository and directory structure in the code, The second condition c_2 compares the pair $<€_T,€_S>$ with the code type C_T to make sure that the correct level of design is compared, E.g., in SysML both a component and an interface may be of the type "block", however they are stereotyped differently.

The third condition c_3 compares the names C_N and $€_N$. Because names are sometimes abbreviated, or harmonized in either model or code, a dictionary is used for widening the name comparison. Some abbreviations can be rule based, others need to be explicit.

If all three conditions are fulfilled the matcher will create a trace between the model and the code, The traces (links) that are created are bi-directional and will be used in several different ways. The first step has been to indicate the consistency of each system component. Basic KPIs, showing the percentage of consistency between the model and code are presented in an html page.

Further work is done in order to use the links to blend the SysML diagrams from the SysML tool of choice (currently Rhapsody) together with the Software IDE (currently Jetbrains CLion). The selected code should show the appropriate SysML diagram is the CLion IDE. To show the Rhapsody diagrams in CLion, we are planning on running Rhapsody in headless state, and dynamically retrieve diagrams based on the links created by the matcher.

By blending the notations and showing the links continuously it is easier for both software engineers, and systems engineers to understand the gaps in the consistency between the model and its implementation that emerge throughout the evolution of the system.

Figure 14 shows the overall architecture of the matcher, CLion and Rhapsody.

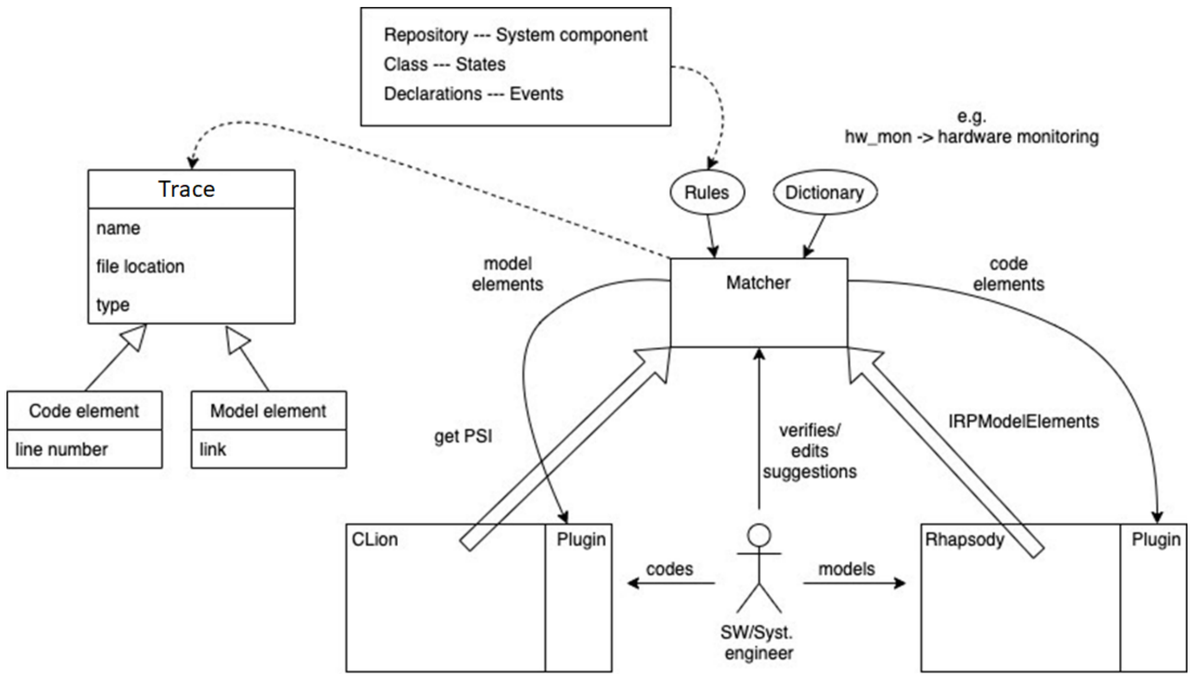


Figure 14. Overall architecture of the matcher, CLion and Rhapsody

3.4.2. Relation to BUMBLE Features

Blended Syntaxes & Modeling (B)

As discussed the Blending will be performed by showing SysML diagrams in the code IDE or vice versa. While working with code the linked SysML diagrams should be shown.

Traceability (T)

The main purpose of the proposed solution with a matcher is to automatically create links between models based on heuristics. In our (Saab) case we have created a set of rules and conditions based on our architectural guidelines.

Evolution (E)

It is obvious that the links help in understanding the evolution of the system. The regression of the links them-selves may also be used for creating information of “evolution-rate”. As elements emerge or disappear in the system model (matched or not) is an indication of how fast the system model is evolving, and vice versa.

4. Conclusion

In the context of the Eclipse technology space, this deliverable presents the software architectures of approaches that realize four out of seven BUMBLE use cases based on the Eclipse technology space. The presented approaches build upon the Eclipse RCP and the client/server architectural style. We also relate each of the described use cases to the corresponding BUMBLE features that they provide.